The LispKit Manual

Volume 2 (Sources)

Peter Henderson
University of Stirling

Geraint A. Jones
Oxford University

Simon B. Jones
University of Stirling

Technical Monograph PRG-32(2)

Oxford University Computing Laboratory
Programming Research Group
8-11, Keble Road
Oxford OX1 3QD
England

Peter Henderson [α]
John Hughes [γ]
Geraint A. Jones [β]
Simon B. Jones [α]
Stephen Murrell [γ]

[α] Department of Computing Science
University of Stirling
Stirling FK9 4LA
Scotland

[β] Jesus College
Oxford OX1 3DW
England

[γ] Wolfson College
Oxford OX2 6UD
England

Parts of this monograph were typeset on a Monotype Lasercomp phototypesetter at Oxford University Computing Service.

The LispKit Lisp manual: II Sources

This volume is the second of a pair which together constitute the LispKit Lisp manual. The first volume consists of a brief introduction to the language and to some of the existing implementations of the system, together with a note about each of the software components of the system. This, the companion volume, contains copies of all of the LispKit Lisp programs and libraries as they were being distributed by the authors at the time of going to press, together with the text of an implemenation of the virtual machine which supports the LispKit system. The listings of LispKit Lisp which follow are, substantially, what you would obtain by passing each of the sources in a distribution copy of the LispKit system through the pretty-printer which is provided in the system - although we confess to ironing out one or two of the more excentric layouts!

The LispKit system is distributed, free of charge, by the authors for copies of the documentation, copies of the LispKit system, or for any information about LispKit, contact:

Peter Henderson*, Geraint Jones or Simon Jones*

Oxford University Computing Laboratory
Programming Research Group
8-11, Keble Road
Oxford OX1 3QD
Oxford (0865) 54141

* address from October 1983

Department of Computing Science
University of Stirling
Stirling FK9 4LA
Stirling (0786) 3171

Contents of Volume II

CHECK LSO

```
( letrec
   check
   ( comment
      quote
      ( ( Variable definition and scope checker )
         ( P Henderson and S B Jones PRG Oxford August 1982 ) ) )
   ( check lambda ( p ) ( checkexpr p ( quote NIL ) ( quote NIL ) ) )
   ( checkexpr
      lambda
      ( p e loc )
      ( if
         ( atom p )
         ( ok p e loc )
      ( if
         ( eq ( head p ) ( quote quote ) )
         ( quote NIL )
      ( if
         ( eq ( head p ) ( quote lambda ) )
         ( checkexpr
            ( head ( tail ( tail p ) ) )
            ( append ( head ( tail p ) ) e )
            loc )
      ( if
         ( eq ( head p ) ( quote let ) )
         ( checklet ( tail p ) e loc )
      ( if
         ( eq ( head p ) ( quote letrec ) )
         ( checkletrec ( tail p ) e loc )
      ( if
         ( isop ( head p ) )
         ( checkeach ( tail p ) e loc )
         ( checkeach p e loc ) ) ) ) ) ) ) )
   ( ok
      lambda
      ( name e loc )
      ( if
         ( eq e ( quote NIL ) )
         ( cons ( cons name loc ) ( quote NIL ) )
      ( if
         ( eq name ( head e ) )
         ( quote NIL )
         ( ok name ( tail e ) loc ) ) ) )
   ( isop
      lambda
      ( op )
      ( member
         op
         ( quote
            ( chr
```

```
            atom
            head
            tail
            leq
            eq
            cons
            add
            sub
            mul
            div
            rem
            if ) ) ) )
( member
   lambda
   ( x l )
   ( if
      ( eq l ( quote NIL ) )
      ( quote F )
   ( if
      ( eq x ( head l ) )
      ( quote T )
      ( member x ( tail l ) ) ) ) )
( append
   lambda
   ( l1 l2 )
   ( if
      ( eq l1 ( quote NIL ) )
      l2
      ( cons ( head l1 ) ( append ( tail l1 ) l2 ) ) ) )
( vars
   lambda
   ( l )
   ( if
      ( eq l ( quote NIL ) )
      ( quote NIL )
      ( cons ( head ( head l ) ) ( vars ( tail l ) ) ) ) )
( checklet
   lambda
   ( p e loc )
   ( append
      ( checkexpr
         ( head p )
         ( append ( vars ( tail p ) ) e )
         loc )
      ( checklist ( tail p ) e loc ) ) )
( checkletrec
   lambda
   ( p e loc )
   ( let
      ( append
         ( checkexpr ( head p ) newe loc )
         ( checklist ( tail p ) newe loc ) )
      ( newe append ( vars ( tail p ) ) e ) ) )
( checklist
   lambda
```

```
    ( l e loc )
    ( if
      ( eq l ( quote NIL ) )
      ( quote NIL )
      ( append
        ( checkexpr
          ( tail ( head l ) )
          e
          ( cons ( head ( head l ) ) loc ) )
        ( checklist ( tail l ) e loc ) ) ) )
( checkeach
  lambda
  ( l e loc )
  ( if
    ( eq l ( quote NIL ) )
    ( quote NIL )
    ( append
      ( checkexpr ( head l ) e loc )
      ( checkeach ( tail l ) e loc ) ) ) ) )
```

```
( letrec
   ( lambda
      ( keyboard )
      ( stream
         ( c ( head keyboard ) ( lambda ( x ) x ) ( tail keyboard ) ) ) )
   ( stream lambda ( s ) ( cons s ( quote NIL ) ) )
   ( c
      lambda
      ( n f l )
      ( if
         ( eq n ( quote 0 ) )
         ( f ( head l ) )
         ( c
            ( sub n ( quote 1 ) )
            ( lambda ( x ) ( f ( ( load_code ( head l ) ) x ) ) )
            ( tail l ) ) ) ) ) )
```

```
( lambda
  ( i )
  ( cons
    newline
  ( cons
    ( quote Editor )
    ( sequence
      ( head i )
      ( cons
        ( quote ready )
        ( append
          ( edit_output ( edit ( head i ) ( untilend ( tail i ) ) ) )
          ( cons newline ( quote ( Exit editor ) ) ) ) ) ) ) ) ) )
```

```
( letrec
   ( convert ( cons ( quote 2 ) ones ) )
   ( convert
      lambda
      ( l )
      ( cons
         ( head l )
         ( convert
            ( normalise
               ( quote 2 )
               ( cons ( quote 0 ) ( mult ( tail l ) ) ) ) ) ) ) )
   ( ones cons ( quote 1 ) ones )
   ( mult
      lambda
      ( l )
      ( cons
         ( mul ( quote 10 ) ( head l ) )
         ( mult ( tail l ) ) ) )
   ( normalise
      lambda
      ( c l )
      ( let
         ( if
            ( eq
               ( div e c )
               ( div ( add e ( quote 9 ) ) c ) )
            ( cons
               ( add d ( div e c ) )
               ( normalise
                  ( add c ( quote 1 ) )
                  ( cons ( rem e c ) x ) ) )
            ( carry
               c
               ( cons
                  d
                  ( normalise ( add c ( quote 1 ) ) ( cons e x ) ) ) ) )
         ( d head l )
         ( e head ( tail l ) )
         ( x tail ( tail l ) ) ) )
   ( carry
      lambda
      ( c l )
      ( let
         ( cons
            ( add d ( div e c ) )
            ( cons ( rem e c ) x ) )
         ( d head l )
         ( e head ( tail l ) )
         ( x tail ( tail l ) ) ) ) )
```

```
( letrec
   ( lambda
      ( kb )
      ( cons
         ( quote Editor )
      ( cons ( quote ready ) ( edit ( head kb ) ( tail kb ) ) ) ) )
   ( comment
      quote
      ( ( S-expression editor )
         ( P Henderson and S B Jones PRG Oxford August 1982 ) ) )
   ( edit lambda ( f c ) ( editloop c f ( quote NIL ) f ) )
   ( editloop
      lambda
      ( c f s pf )
      ( letrec
         ( if
            ( eq ( first t ) exit )
            ( quote ( Exit editor ) )
         ( if
            ( eq ( first t ) continue )
            rest
            ( cons ( second t ) rest ) ) )
         ( rest
            editloop
            ( tail c )
            ( third t )
            ( fourth t )
            ( fifth t ) )
         ( 1 editstep ( head c ) f s pf ) ) )
   ( editstep
      lambda
      ( c f s pf )
      ( if
         ( atom c )
         ( if
            ( eq c ( quote end ) )
            ( tuple exit nothing f s pf )
         ( if
            ( eq c ( quote h ) )
            ( if
               ( atom f )
               ( tuple error error f s pf )
               ( tuple
                  continue
                  nothing
                  ( head f )
                  ( cons
                     ( lambda ( x ) ( cons x ( tail f ) ) )
                     s )
                  ( head f ) ) )
         ( if
            ( eq c ( quote t ) )
```

```
      ( if
        ( atom f )
        ( tuple error error f s pf )
        ( tuple
          continue
          nothing
          ( tail f )
          ( cons
            ( lambda ( x ) ( cons ( head f ) x ) )
            s )
          ( tail f ) ) )
    ( if
      ( eq c ( quote u ) )
      ( if
        ( atom s )
        ( tuple error error f s pf )
        ( tuple
          continue
          nothing
          ( ( head s ) f )
          ( tail s )
          ( ( head s ) f ) ) )
    ( if
      ( eq c ( quote p ) )
      ( tuple prompt ( dump ( quote 2 ) f ) f s pf )
    ( if
      ( eq c ( quote top ) )
      ( let
        ( tuple continue nothing topf ( quote NIL ) topf )
        ( topf top f s ) )
    ( if
      ( eq c ( quote undo ) )
      ( tuple continue nothing pf s f )
      ( tuple error error f s pf ) ) ) ) ) ) ) )
( if
  ( eq ( head c ) ( quote c ) )
  ( let
    ( if
      ( eq m ( quote NIL ) )
      ( tuple error error f s pf )
      ( tuple
        continue
        nothing
        ( subst m ( shead ( stail ( tail c ) ) ) )
        s
        f ) )
    ( m match ( shead ( tail c ) ) f ) )
( if
  ( eq ( head c ) ( quote r ) )
  ( tuple continue nothing ( shead ( tail c ) ) s f )
( if
  ( eq ( head c ) ( quote p ) )
  ( if
    ( eq ( shead ( tail c ) ) ( quote all ) )
    ( tuple prompt f f s pf )
```

```
            ( tuple
              prompt
              ( dump ( shead ( tail c ) ) f )
              f
              s
              pf ) )
          ( tuple error error f s pf ) ) ) ) ) )
    ( shead lambda ( x ) ( if ( atom x ) x ( head x ) ) )
    ( stail lambda ( x ) ( if ( atom x ) x ( tail x ) ) )
    ( match
      lambda
      ( p s )
      ( if
        ( eq p ( quote NIL ) )
        ( quote NIL )
      ( if
        ( atom p )
        ( cons ( cons p s ) ( quote NIL ) )
      ( if
        ( atom s )
        ( quote NIL )
        ( append
          ( match ( head p ) ( head s ) )
          ( match ( tail p ) ( tail s ) ) ) ) ) ) ) )
    ( append
      lambda
      ( x y )
      ( if
        ( eq x ( quote NIL ) )
        y
        ( cons ( head x ) ( append ( tail x ) y ) ) ) ) )
    ( assoc
      lambda
      ( x a )
      ( if
        ( eq a ( quote NIL ) )
        x
      ( if
        ( eq ( head ( head a ) ) x )
        ( tail ( head a ) )
        ( assoc x ( tail a ) ) ) ) ) )
    ( subst
      lambda
      ( a p )
      ( if
        ( atom p )
        ( assoc p a )
        ( cons
          ( subst a ( head p ) )
          ( subst a ( tail p ) ) ) ) ) )
    ( dump
      lambda
      ( n x )
      ( if
        ( atom x )
```

```
      x
   ( if
      ( eq n ( quote 0 ) )
      ( quote * )
      ( cons
         ( dump ( sub n ( quote 1 ) ) ( head x ) )
         ( dump n ( tail x ) ) ) ) ) )
( top
   lambda
   ( f s )
   ( if
      ( eq s ( quote NIL ) )
      f
      ( top ( ( head s ) f ) ( tail s ) ) ) )
( tuple
   lambda
   ( a b c d e )
   ( cons a ( cons b ( cons c ( cons d e ) ) ) ) )
( first lambda ( x ) ( head x ) )
( second lambda ( x ) ( head ( tail x ) ) )
( third lambda ( x ) ( head ( tail ( tail x ) ) ) )
( fourth lambda ( x ) ( head ( tail ( tail ( tail x ) ) ) ) )
( fifth lambda ( x ) ( tail ( tail ( tail ( tail x ) ) ) ) )
( prompt quote Prompt )
( error quote Error )
( exit quote Exit )
( continue quote Continue )
( nothing quote Nothing ) )
```

```
( let
   ( lambda ( input ) ( ( expert ( head input ) ) ( tail input ) ) )
   ( expert
      lambda
      ( database )
      ( letrec
         ( lambda
            ( kb )
            ( sysq kb ( quote init ) ( quote NIL ) ( quote NIL ) ) )
         ( sysq
            lambda
            ( kb stq kn v )
            ( if
               ( eq stq ( quote end ) )
               ( quote NIL )
               ( let
                  ( sys kb ( sect nstq database ) kn v nstq )
                  ( nstq if ( eq stq ( quote * ) ) v stq ) ) ) )
         ( sys
            lambda
            ( kb st kn v stq )
            ( if
               ( eq st ( quote NIL ) )
               ( sysq kb stq kn v )
               ( let
                  ( if
                     ( eq key ( quote goto ) )
                     ( sysq kb ( head arg ) kn v )
                  ( if
                     ( eq key ( quote say ) )
                     ( append
                        ( ch arg v )
                        ( cons
                           newline
                           ( sys kb ( tail st ) kn v stq ) ) )
                  ( if
                     ( eq key ( quote note ) )
                     ( sys
                        kb
                        ( tail st )
                        ( newkn kn ( head arg ) ( head ( tail arg ) ) )
                        v
                        stq )
                  ( if
                     ( eq key ( quote getval ) )
                     ( sys
                        kb
                        ( tail st )
                        kn
                        ( lookup ( head arg ) kn stq ) )
                  ( if
                     ( eq key ( quote enq ) )
```

```
            ( enq
              kb
              ( tail st )
              kn
              ( lookup ( head arg ) kn )
              ( head arg )
              ( ch ( tail arg ) v )
              stq )
         ( if
           ( eq key ( quote ask ) )
           ( enq
             kb
             ( tail st )
             kn
             ( quote NIL )
             ( head arg )
             ( ch ( tail arg ) v )
             stq )
         ( if
           ( eq key ( quote if ) )
           ( if
             ( match ( head arg ) v )
             ( sysq kb ( head ( tail arg ) ) kn v )
             ( sys kb ( tail st ) kn v stq ) )
         ( if
           ( eq key ( quote ifn ) )
           ( if
             ( match ( head arg ) v )
             ( sys kb ( tail st ) kn v stq )
             ( sysq kb ( head ( tail arg ) ) kn v ) )
           ( cons key ( quote ????? ) ) ) ) ) ) ) ) ) )
         ( key head ( head st ) )
         ( arg tail ( head st ) ) ) ) )
( ch
  lambda
  ( s v )
  ( if ( eq ( head s ) ( quote * ) ) v s ) )
( match
  lambda
  ( a b )
  ( if
    ( atom a )
    ( or ( eq a ( quote ? ) ) ( eq a b ) )
    ( unless
      ( atom b )
      ( and
        ( match ( head a ) ( head b ) )
        ( match ( tail a ) ( tail b ) ) ) ) ) )
( lookup
  lambda
  ( x kn )
  ( if
    ( eq kn ( quote NIL ) )
    ( quote NIL )
  ( if
```

```
      ( eq ( head ( head kn ) ) x )
      ( tail ( head kn ) )
      ( lookup x ( tail kn ) ) ) ) )
( enq
   lambda
   ( kb st kn lup nam str stq )
   ( if
      ( eq lup ( quote NIL ) )
      ( append
         str
         ( sys
            ( tail kb )
            st
            ( newkn kn nam ( head kb ) )
            ( head kb )
            stq ) )
      ( sys kb st kn lup stq ) ) )
( sect
   lambda
   ( q d )
   ( if
      ( eq q ( head ( head d ) ) )
      ( tail ( head d ) )
      ( sect q ( tail d ) ) ) )
( newkn
   lambda
   ( kn x v )
   ( if
      ( eq kn ( quote NIL ) )
      ( cons ( cons x v ) ( quote NIL ) )
   ( if
      ( eq x ( head ( head kn ) ) )
      ( cons ( cons x v ) ( tail kn ) )
      ( cons ( head kn ) ( newkn ( tail kn ) x v ) ) ) ) ) ) ) )
```

```
( ( init
    ( say Kelloggs Delegging Machine ( Model A ) service routine )
    ( goto presbut ) )
  ( tagain ( say Try again: ) ( goto presbut ) )
  ( presbut
    ( ask q Is the button pressable? )
    ( if yes legoff )
    ( if no isbut ) )
  ( isbut
    ( enq qisbut Is there a button? )
    ( if yes dedcoy )
    ( note butret tagain )
    ( if no newbut ) )
  ( newbut
    ( say Fit a new button assembly ( part no £765/wy/35454y/z2 ) )
    ( note qisbut yes )
    ( note button new )
    ( getval butret )
    ( goto * ) )
  ( dedcoy
    ( getval qdedcoy )
    ( if done infkel )
    ( note qdedcoy done )
    ( say Remove any dead coypu or empty cornflake packets from the )
    ( say button assembly )
    ( goto tagain ) )
  ( infkel
    ( say Inspect the ablative distending grommet for signs of infestation )
    ( say by kelp or other southern Atlantic seaweeds --- )
    ( enq qinfkel is it clear? )
    ( if no solsoak )
    ( getval button )
    ( if new dealer )
    ( note butret tagain )
    ( goto newbut ) )
  ( solsoak
    ( note qinfkel yes )
    ( say Soak in Kelloggs Patent Marine Solvent for three days )
    ( goto tagain ) )
  ( pagain ( say Press the button again ) ( goto legoff ) )
  ( legoff
    ( ask q Do all the legs fall off? )
    ( if yes isclen )
    ( if no anylof ) )
  ( anylof
    ( getval qalof )
    ( if yes insflegs )
    ( ask qalof Do any of the legs fall off? )
    ( if no anylegs )
    ( note butquery no )
    ( if yes insflegs ) )
  ( anylegs
```

```
    ( enq qanylegs Are there any legs? )
    ( if no nolegs )
    ( note butquery yes )
    ( if yes insflegs ) )
  ( nolegs
    ( note qanylegs yes )
    ( say Fix new legs ( part no £7576/e/7yeyr/5ww/p ) )
    ( goto pagain ) )
  ( insflegs
    ( getval despr )
    ( if yes noglue )
    ( say Inspect the non-departed legs: )
    ( enq nailedon Have they been nailed on? )
    ( if yes remnails )
    ( if no lookglue ) )
  ( remnails
    ( note nailedon no )
    ( say Remove all nails from the legs )
    ( goto pagain ) )
  ( lookglue
    ( note despr yes )
    ( enq isglue Is there any sign of glue? )
    ( if yes remglue )
    ( if no noglue ) )
  ( remglue
    ( note isglue no )
    ( say Soak legs in paradi-chloro-phenyl-pentanoic acid )
    ( goto pagain ) )
  ( noglue
    ( getval butquery )
    ( if no tryone )
    ( getval button )
    ( if new tryone )
    ( note butret pagain )
    ( goto newbut ) )
  ( tryone
    ( getval qone )
    ( if done trytwo )
    ( note qone done )
    ( say Replace the pelargonium extraction unit )
    ( say ( part no £-2081070112gf-yu4 ) )
    ( goto pagain ) )
  ( trytwo
    ( getval qtwo )
    ( if done dealer )
    ( note qtwo done )
    ( say Try inserting a secondary positron acidifier in the benign )
    ( say matriculating quadrille )
    ( goto pagain ) )
  ( dealer
    ( say OK, I give up )
    ( say Go to your nearest authorised dealer )
    ( goto end ) )
  ( isclen
    ( ask q Is the machine clean and shiny? )
```

```
    ( if yes nofault )
    ( if no trysoap ) )
( nofault
    ( say Your Kelloggs Delegging Machine ( Model A ) )
    ( say is now in perfect working order )
    ( goto end ) )
( trysoap
    ( getval qsoap )
    ( if done tryboil )
    ( note qsoap done )
    ( say Sponge the machine with warm soapy water. )
    ( say taking care not to wet the expeditionary telephone emulator )
    ( say or the cold air intake conduit )
    ( goto isclen ) )
( tryboil
    ( getval qboil )
    ( if done filthy )
    ( note qboil done )
    ( say Boil the machine for 30 mins in deionised water )
    ( goto isclen ) )
( filthy
    ( say Your machine is incurable filthy, I can do nothing for it )
    ( goto end ) ) )
```

```
( run_and_halt ( quote NIL ) )
```

IDENTITY **LSO**

```
( lambda ( x ) x )
```

```
( letrec
   ( i ( quote 0 ) )
   ( i
      lambda
      ( n )
      ( cons n ( i ( add ( quote 1 ) n ) ) ) ) )
```

```
( letrec
  ( lambda
    ( kb )
    ( append
      ( quote ( LispKit interpreter: type end to finish ) )
      ( interact
        kb
        ( quote ( patience ) )
        ( quote ( ( quote 50 ) ) )
        ( quote NotDefined ) ) ) )
  ( comment
    quote
    ( ( LispKit lisp interpreter )
      ( Geraint Jones. PRG Oxford )
      ( Last changed 29 March 1983 ) ) )
  ( interact
    lambda
    ( kb global_n global_v it )
    ( letrec
      ( cons newline ( cons ( quote > ) line ) )
      ( line
        if
        ( eq tag ( quote exit ) )
        ( quote ( Exit interpreter ) )
        ( if
          ( eq tag ( quote message ) )
          ( append
            message
            ( interact ( tail kb ) new_global_n new_global_v it ) )
          ( if
            ( eq tag ( quote evaluation ) )
            ( append
              ( print expression )
              ( interact ( tail kb ) global_n global_v expression ) )
            ( if
              ( eq tag ( quote restore ) )
              ( interact
                ( append
                  file
                  ( let
                    ( if
                      ( eq ( head t ) ( quote > ) )
                      ( tail t )
                      t )
                    ( t tail kb ) ) )
                global_n
                global_v
                it )
              ( cons
                ( quote Error )
                ( interact ( tail kb ) global_n global_v it ) ) ) ) ) )
      ( tag 1 result ) )
```

```
( message 2 result )
( expression 2 result )
( file 2 result )
( new_global_n 3 result )
( new_global_v 4 result )
( result loop ( head kb ) global_n global_v it )
( print
  letrec
  ( lambda ( s ) ( first patience ( flatten s ) ) )
  ( first
    lambda
    ( n l )
    ( if
      ( eq l ( quote NIL ) )
      ( quote NIL )
    ( if
      ( eq n ( quote 0 ) )
      ( let
        ( list stop stop stop )
        ( stop chr ( quote 46 ) ) )
      ( cons
        ( head l )
        ( first ( sub n ( quote 1 ) ) ( tail l ) ) ) ) ) ) )
  ( patience evaluate ( quote patience ) global_n global_v it )
  ( flatten
    lambda
    ( s )
    ( letrec
      ( if
        ( isfunction s )
        ( symbol ( quote function ) ( flatten ( showfunction s ) ) )
      ( if
        ( iscode s )
        ( symbol ( quote co'e ) ( flatten ( showcode s ) ) )
      ( if
        ( atom s )
        ( list s )
        ( cons
          open
          ( append ( flatten ( head s ) ) ( tailpart r ) ) ) ) ) )
      ( symbol
        lambda
        ( type rep )
        ( cons
          open
        ( cons
          point
        ( cons type ( cons point ( tailpart rep ) ) ) ) ) )
      ( tailpart
        lambda
        ( r )
        ( if
          ( eq ( head r ) ( quote NIL ) )
          ( list close )
        ( if
```

```
                ( eq ( head r ) open )
                ( tail r )
                ( cons point ( append r ( list close ) ) ) ) ) )
            ( r flatten ( tail s ) )
            ( open chr ( quote 40 ) )
            ( close chr ( quote 41 ) )
            ( point chr ( quote 46 ) ) ) ) ) ) )
( loop
  lambda
  ( command global_n global_v it )
  ( letrec
    ( if
      ( atom command )
      ( if
        ( eq command ( quote end ) )
        exit
      ( if
        ( eq command ( quote save ) )
        ( save global_n global_n global_v )
      ( if
        ( eq command ( quote vars ) )
        ( message global_n global_n global_v )
        ( evaluation expression ) ) ) )
    ( if
      ( eq keyword ( quote def ) )
      ( if
        ( eq name ( quote Error ) )
        error
      ( if
        ( atom name )
        ( message
          ( cons name ( quote ( defined ) ) )
          ( cons name global_n )
          ( cons value global_v ) )
        error ) )
    ( if
      ( eq keyword ( quote cancel ) )
      ( if
        ( member name global_n )
        ( message
          ( cons name ( quote ( cancelled ) ) )
          ( remove name global_n global_n )
          ( remove name global_n global_v ) )
        error )
    ( if
      ( eq keyword ( quote save ) )
      ( save ( tail command ) global_n global_v )
    ( if
      ( eq keyword ( quote restore ) )
      ( restore ( tail command ) )
      ( evaluation expression ) ) ) ) )
    ( keyword head command )
    ( name head' ( tail' command ) )
    ( value head' ( tail' ( tail' command ) ) )
    ( expression evaluate command global_n global_v it )
```

```
        ( remove
          lambda
          ( a t l )
          ( if
            ( eq a ( head t ) )
            ( tail l )
            ( cons
              ( head l )
              ( remove a ( tail t ) ( tail l ) ) ) ) ) )
        ( exit list ( quote exit ) )
        ( message
          lambda
          ( m n v )
          ( list ( quote message ) m n v ) )
        ( evaluation lambda ( e ) ( list ( quote evaluation ) e ) )
        ( error message ( quote ( Error ) ) global_n global_v )
        ( save
          lambda
          ( l global_n global_v )
          ( letrec
            ( message
              ( list
                ( cons
                  ( quote restore )
                  ( deflist l global_n global_v ( quote NIL ) ) ) ) )
              global_n
              global_v )
            ( deflist
              lambda
              ( l n v d )
              ( if
                ( eq n ( quote NIL ) )
                d
                ( deflist
                  l
                  ( tail n )
                  ( tail v )
                  ( if
                    ( member ( head n ) l )
                    ( cons
                      ( list
                        ( quote def )
                        ( head n )
                        ( head v ) )
                      d )
                    d ) ) ) ) ) )
        ( restore lambda ( f ) ( list ( quote restore ) f ) ) ) )
    ( evaluate
      letrec
      ( lambda
        ( e global_n global_v it )
        ( letrec
          ( eval e n v )
          ( n cons ( cons ( quote it ) global_n ) ( quote NIL ) )
          ( v
```

```
        cons
        ( cons it ( listeval global_v n v ) )
        ( quote NIL ) ) ) )
( eval
  lambda
  ( e n v )
  ( letrec
     ( if
        ( atom e )
        ( associate e n v )
     ( if
        ( eq keyword ( quote quote ) )
        ( letrec
           ( secure text1 )
           ( secure
              lambda
              ( l )
              ( let
                 ( if
                    ( atom l )
                    l
                    ( cons
                       ( if
                          ( atom h )
                          ( if ( reserved h ) ( quote Error ) h )
                          ( secure h ) )
                       ( secure ( tail l ) ) ) )
                 ( h head l ) ) ) ) )
     ( if
        ( eq keyword ( quote atom ) )
        ( atom argument1 )
     ( if
        ( eq keyword ( quote head ) )
        ( if
           ( indivisible argument1 )
           ( quote Error )
           ( head argument1 ) )
     ( if
        ( eq keyword ( quote tail ) )
        ( if
           ( indivisible argument1 )
           ( quote Error )
           ( tail argument1 ) )
     ( if
        ( eq keyword ( quote cons ) )
        ( cons
           ( if ( reserved argument1 ) ( quote Error ) argument1 )
           argument2 )
     ( if
        ( eq keyword ( quote eq ) )
        ( eq argument1 argument2 )
     ( if
        ( eq keyword ( quote leq ) )
        ( arithmetic ( leq argument1 argument2 ) )
     ( if
```

```
      ( eq keyword ( quote add ) )
      ( arithmetic ( add argument1 argument2 ) )
   ( if
      ( eq keyword ( quote sub ) )
      ( arithmetic ( sub argument1 argument2 ) )
   ( if
      ( eq keyword ( quote mul ) )
      ( arithmetic ( mul argument1 argument2 ) )
   ( if
      ( eq keyword ( quote div ) )
      ( arithmetic
         ( if
            ( eq argument2 ( quote 0 ) )
            ( quote Error )
            ( div argument1 argument2 ) ) )
   ( if
      ( eq keyword ( quote rem ) )
      ( arithmetic
         ( if
            ( eq argument2 ( quote 0 ) )
            ( quote Error )
            ( rem argument1 argument2 ) ) )
   ( if
      ( eq keyword ( quote if ) )
      ( if argument1 argument2 argument3 )
   ( if
      ( eq keyword ( quote lambda ) )
      ( makefunction text1 text2 n v )
   ( if
      ( eq keyword ( quote let ) )
      ( let
         ( eval text1 newn newv )
         ( newn cons ( variables definitions ) n )
         ( newv cons ( listeval ( values definitions ) n v ) v ) )
   ( if
      ( eq keyword ( quote letrec ) )
      ( letrec
         ( eval text1 newn newv )
         ( newn cons ( variables definitions ) n )
         ( newv
            cons
            ( listeval ( values definitions ) newn newv )
            v ) )
   ( if
      ( eq keyword ( quote chr ) )
      ( if ( number argument1 ) ( chr argument1 ) ( quote Error ) )
   ( if
      ( eq keyword ( quote exec ) )
      ( if
         ( atom argument1 )
         ( quote Error )
         ( makecode argument1 ) )
      ( letrec
         ( if
            ( isfunction applicand )
```

```
                ( eval body newn newv )
              ( if
                ( iscode applicand )
                ( apply function arguments )
                ( quote Error ) ) )
              ( body tail ( head function ) )
              ( newn
                cons
                ( head ( head function ) )
                ( head ( tail function ) ) )
              ( newv cons arguments ( tail ( tail function ) ) )
              ( function tail applicand )
              ( applicand eval ( head e ) n v ))))) ))))) ))))) ))))) ) )
        ( keyword head e )
        ( arguments listeval texts n v )
        ( argument1 head' arguments )
        ( argument2 head' ( tail' arguments ) )
        ( argument3 head' ( tail' ( tail' arguments ) ) )
        ( arithmetic
          lambda
          ( result )
          ( if
            ( and ( number argument1 ) ( number argument2 ) )
            result
            ( quote Error ) ) )
        ( definitions assoclist' ( tail' texts ) )
        ( texts tail e )
        ( text1 head' texts )
        ( text2 head' ( tail' texts ) ) ) )
  ( listeval
    let
    ( lambda ( l n v ) ( map ( e n v ) l ) )
    ( e
      lambda
      ( n v )
      ( lambda ( x ) ( eval x n v ) ) ) )
  ( associate
    letrec
    ( lambda
      ( a n v )
      ( if
        ( eq n ( quote NIL ) )
        ( quote NotDefined )
      ( if
        ( member a ( head n ) )
        ( locate a ( head n ) ( head v ) )
        ( associate a ( tail n ) ( tail v ) ) ) ) )
    ( locate
      lambda
      ( a n v )
      ( if
        ( atom v )
        ( quote Error )
      ( if
        ( eq a ( head n ) )
```

```
            ( head  v )
            ( locate  a  ( tail  n )  ( tail  v ) ) ) ) ) )
      ( variables
         let
         ( lambda  ( d )  ( map  v  d ) )
         ( v  lambda  ( b )  ( head  b ) ) )
      ( values
         let
         ( lambda  ( d )  ( map  v  d ) )
         ( v  lambda  ( b )  ( tail  b ) ) )
      ( indivisible
         lambda
         ( c )
         ( or  ( atom  c )  ( or  ( isfunction  c )  ( iscode  c ) ) ) ) ) )
```

```
( letrec
   compile
   ( comment
      quote
      ( ( Compiler generating closures from lambda expressions )
        ( Geraint Jones. PRG Oxford. 25 March 1983 ) ) )
   ( compile
      lambda
      ( e )
      ( cons ( comp e ( quote NIL ) RTN_seq ) ( quote NIL ) ) ) )
```

```
( letrec
   ( lambda
      ( i )
      ( append
         ( quote ( LispKit Librarian ) )
         ( librarian ( head i ) ( freevars ( head i ) ) ( tail i ) ) ) )
   ( librarian
      lambda
      ( e u i )
      ( letrec
         ( if
            ( eq missing ( quote NIL ) )
            ( write ( bind_operators e u ) i )
            ( cons
               newline
               ( append
                  missing
                  ( if
                     ( eq ( head i ) ( quote end ) )
                     ( write e ( tail i ) )
                     ( if
                        ( eq ( head i ) ( quote abort ) )
                        ( quote NIL )
                        ( librarian e' u' ( tail i ) ) ) ) ) ) ) )
         ( missing difference u operators )
         ( e' head next )
         ( u' tail next )
         ( next bind e u ( head i ) )
         ( write
            lambda
            ( e i )
            ( cons
               newline
               ( append
                  ( quote ( Type anything to print result ) )
                  ( sequence ( head i ) ( cons e ( quote NIL ) ) ) ) ) ) ) ) )
   ( bind
      lambda
      ( e u a )
      ( letrec
         ( cons
            ( if
               ( eq defs ( quote NIL ) )
               e
               ( cons ( quote letrec ) ( cons e defs ) ) )
            ( difference u' ( map ( lambda ( d ) ( head d ) ) a ) ) )
         ( defs
            filter
            ( lambda ( d ) ( member ( head d ) u' ) )
            a )
         ( u'
            close
```

```
        ( lambda
          ( v )
          ( reduce
            ( lambda
              ( d l )
              ( if
                ( member ( head d ) v )
                ( ( tail d ) l )
                l ) )
            a'
            v ) )
        u )
      ( a'
        map
        ( lambda
          ( d )
          ( cons
            ( head d )
            ( lambda ( l ) ( union ( freevars ( tail d ) ) l ) ) ) ) )
        a ) ) )
  ( bind_operators
    lambda
    ( e u )
    ( letrec
      ( if
        ( eq u ( quote NIL ) )
        e
        ( cons
          ( quote let )
          ( cons
            e
            ( append
              ( map
                ( define ( quote 1 ) )
                ( intersection u monadic_ops ) )
              ( append
                ( map
                  ( define ( quote 2 ) )
                  ( intersection u diadic_ops ) )
                ( map
                  ( define ( quote 3 ) )
                  ( intersection u triadic_ops ) ) ) ) ) ) )
      ( define
        lambda
        ( arity )
        ( lambda
          ( name )
          ( let
            ( list
              name
              ( quote lambda )
              arguments
              ( cons name arguments ) )
            ( arguments
              first
```

```
arity
( quote ( arg1 arg2 arg3 ) ) ) ) ) ) ) ) )
```

```
( letrec
  ( letrec
    ( lambda ( input ) ( librarian ( head input ) ( tail input ) predefined )

    ( comment
      quote
      ( ( S-expression librarian )
        ( Geraint Jones, PRG, Oxford )
        ( last changed 29 March 1983 ) ) )
    ( predefined quote NIL )
    ( librarian
      lambda
      ( expression input associst )
      ( letrec
        ( if
          ( eq includes ( quote NIL ) )
          ( append finalprompt ( output expression input ) )
          ( append
            prompts
            ( librarian
              ( instance expression bindings )
              restofinput
              bindings ) ) )
        ( includes requirements expression )
        ( do_bindings bind_names includes input associst )
        ( prompts 1 do_bindings )
        ( restofinput 2 do_bindings )
        ( bindings 3 do_bindings )
        ( finalprompt
          quote
          ( Type 'end' to finish, anything else to print result ) ) ) )
    ( requirements
      lambda
      ( expression )
      ( if
        ( atom expression )
        ( quote NIL )
      ( if
        ( isinclude expression )
        ( cons ( filename expression ) ( quote NIL ) )
        ( append
          ( requirements ( head expression ) )
          ( requirements ( tail expression ) ) ) ) ) )
    ( instance
      lambda
      ( expression associst )
      ( if
        ( atom expression )
        expression
      ( if
        ( isinclude expression )
        ( associate ( filename expression ) associst )
        ( cons
```

```
            ( instance ( head expression ) associist )
            ( instance ( tail expression ) associist ) ) ) ) )
    ( output
      lambda
      ( result input )
      ( if
        ( eq ( head input ) ( quote end ) )
        ( quote ( Exit librarian ) )
        ( cons result ( output result ( tail input ) ) ) ) ) )
    ( bind_names
      lambda
      ( namelist input associist )
      ( letrec
        ( if
          ( eq namelist ( quote NIL ) )
          ( list ( quote NIL ) input associist )
        ( if
          ( defined ( head namelist ) associist )
          ( bind_names ( tail namelist ) input associist )
          ( list
            ( append ( 1 bind_head ) ( 1 bind_tail ) )
            ( 2 bind_tail )
            ( 3 bind_tail ) ) ) )
        ( bind_head bind_new ( head namelist ) ( head input ) )
        ( bind_new
          lambda
          ( name definition )
          ( list
            ( cons
              name
            ( cons ( quote = ) ( sequence definition ( quote NIL ) ) ) )
            name
            definition ) )
        ( bind_tail
          bind_names
          ( tail namelist )
          ( tail input )
          ( bind ( 2 bind_head ) ( 3 bind_head ) associist ) ) ) ) )
( isinclude
  lambda
  ( e )
  ( unless
    ( atom e )
    ( and
      ( eq ( head e ) ( quote include ) )
      ( unless
        ( atom ( tail e ) )
        ( and
          ( atom ( head ( tail e ) ) )
          ( eq ( tail ( tail e ) ) ( quote NIL ) ) ) ) ) ) ) )
( filename lambda ( e ) ( head ( tail e ) ) ) )
```

```
( letrec
   ( run_and_halt load_go_loop )
   ( load_go_loop
      letrec
      ( application load_go ( quote NIL ) )
      ( load_go lambda NIL ( consume ( output ( execute ( input ) ) ) ) )
      ( consume
         letrec
         ( lambda ( s ) ( application step ( list s ) ) )
         ( step
            lambda
            ( s )
            ( if
               ( atom s )
               ( sequence ( print_item newline ) load_go_loop )
               ( sequence
                  ( print_item ( head s ) )
                  ( consume ( tail s ) ) ) ) ) )
      ( output
         lambda
         ( s )
         ( if
            ( atom s )
            ( if
               ( eq s ( quote NIL ) )
               ( quote NIL )
               ( cons s ( quote NIL ) ) )
            ( flatten ( head s ) ( output ( tail s ) ) ) ) )
      ( execute
         lambda
         ( in_stream )
         ( if
            ( letrec
               ( or
                  ( atom closure )
               ( or ( atom code ) ( not ( number first_op ) ) ) )
               ( closure head in_stream )
               ( code head closure )
               ( first_op head code ) )
            ( quote ( input is not a closure ) )
            ( ( load_code ( head in_stream ) ) ( tail in_stream ) ) ) )
      ( input
         lambda
         NIL
         ( letrec
            ( stream item ( input ) )
            ( stream lambda ( a b ) ( sequence a ( cons a b ) ) )
            ( item read_item ) ) ) )
   ( application lambda ( f e ) ( strict_cons f e ) ) )
```

```
( lambda ( s ) ( load_code ( head s ) ) )
```

```
( letrec
   ( lambda
      ( keyboard )
      ( stream
         ( apply
            ( load_code ( head keyboard ) )
            ( args ( head ( tail keyboard ) ) ( tail ( tail keyboard ) ) ) ) ) ) )
   ( stream lambda ( s ) ( cons s ( quote NIL ) ) )
   ( args
      lambda
      ( n l )
      ( if
         ( eq n ( quote 0 ) )
         ( quote NIL )
         ( strict_cons
            ( head l )
            ( args ( sub n ( quote 1 ) ) ( tail l ) ) ) ) ) ) )
```

```
( letrec
   ( letrec
      ( lambda
         ( kb )
         ( append
            ( quote ( Logic LispKit Interpreter: type end to finish ) )
            ( interact
               kb
               ( quote ( patience ) )
               ( quote ( ( quote 50 ) ) )
               ( quote NotDefined )
               ( quote NIL ) ) ) )
      ( comment
         quote
         ( ( Logic LispKit interpreter. Geraint Jones. PRG Oxford )
            ( Last changed 11 November from the text of )
            ( LispKit Interpreter. Geraint Jones. PRG Oxford )
            ( Last changed 8 November 1982 ) ) )
      ( interact
         lambda
         ( kb globaln globalv it database )
         ( letrec
            ( cons ( chr ( quote 13 ) ) ( cons ( quote > ) line ) )
            ( line
               if
               ( eq tag ( quote exit ) )
               ( quote ( Exit logic interpreter ) )
               ( if
                  ( eq tag ( quote message ) )
                  ( append
                     message
                     ( interact
                        ( tail kb )
                        newglobaln
                        newglobalv
                        it
                        newdatabase ) )
                  ( if
                     ( eq tag ( quote evaluation ) )
                     ( append
                        ( print expression )
                        ( interact
                           ( tail kb )
                           globaln
                           globalv
                           expression
                           database ) )
                     ( if
                        ( eq tag ( quote restore ) )
                        ( interact
                           ( append
                              file
```

```
            ( let
               ( if
                  ( eq ( head t ) ( quote > ) )
                  ( tail t )
                  t )
               ( t tail kb ) ) )
         globaln
         globalv
         it
         database )
      ( cons
         ( quote Error )
         ( interact ( tail kb ) globaln globalv it database ) ) ) ) ) )
( tag head result )
( message head ( tail result ) )
( expression head ( tail result ) )
( file tail result )
( newglobaln head ( tail ( tail result ) ) )
( newglobalv head ( tail ( tail ( tail result ) ) ) )
( newdatabase head ( tail ( tail ( tail ( tail result ) ) ) ) )
( result loop ( head kb ) globaln globalv it database )
( print
   letrec
   ( lambda ( s ) ( first patience ( flatten s ) ) )
   ( first
      lambda
      ( n l )
      ( if
         ( eq l ( quote NIL ) )
         ( quote NIL )
      ( if
         ( eq n ( quote 0 ) )
         ( let
            ( cons
               stop
            ( cons stop ( cons stop ( quote NIL ) ) ) )
            ( stop chr ( quote 46 ) ) )
         ( cons
            ( head l )
            ( first ( sub n ( quote 1 ) ) ( tail l ) ) ) ) ) ) )
   ( patience
      evaluate
      ( quote patience )
      globaln
      globalv
      it
      database )
   ( flatten
      lambda
      ( s )
      ( letrec
         ( if
            ( isfunction s )
            ( quote ( **function** ) )
         ( if
```

```
                ( isvariable s )
                ( subscript ( tail s ) ( quote NIL ) ) )
             ( if
                ( atom s )
                ( cons s ( quote NIL ) )
                ( cons open ( append ( flatten ( head s ) ) t ) ) ) ) )
             ( t
                if
                ( eq ( head r ) ( quote NIL ) )
                ( cons close ( quote NIL ) )
                ( if
                   ( eq ( head r ) open )
                   ( tail r )
                   ( cons
                      point
                      ( append r ( cons close ( quote NIL ) ) ) ) ) ) )
             ( r flatten ( tail s ) ) )
             ( subscript
                lambda
                ( v s )
                ( if
                   ( atom v )
                   ( cons v s )
                   ( subscript
                      ( tail v )
                      ( cons colon ( cons ( head v ) s ) ) ) ) ) )
             ( open chr ( quote 40 ) )
             ( close chr ( quote 41 ) )
             ( point chr ( quote 46 ) )
             ( colon chr ( quote 58 ) ) ) ) ) ) )
      ( loop
         lambda
         ( command globaln globalv it database )
         ( letrec
            ( if
               ( atom command )
               ( if
                  ( eq command ( quote end ) )
                  exit
               ( if
                  ( eq command ( quote save ) )
                  ( save
                     ( cons ( quote database ) globaln )
                     globaln
                     globalv
                     database )
               ( if
                  ( eq command ( quote vars ) )
                  ( message globaln globaln globalv database )
               ( if
                  ( eq command ( quote new ) )
                  ( message
                     ( quote ( new database ) )
                     globaln
                     globalv
```

```
        ( quote NIL ) )
      ( evaluation expression ) ) ) ) )
( if
  ( eq keyword ( quote def ) )
  ( if
    ( eq name ( quote Error ) )
    error
  ( if
    ( atom name )
    ( message
      ( cons name ( quote ( defined ) ) )
      ( cons name globaln )
      ( cons value globalv )
      database )
    error ) )
( if
  ( eq keyword ( quote cancel ) )
  ( if
    ( member name globaln )
    ( message
      ( cons name ( quote ( cancelled ) ) )
      ( remove name globaln globaln )
      ( remove name globaln globalv )
      database )
    error )
( if
  ( eq keyword ( quote save ) )
  ( save ( tail command ) globaln globalv database )
( if
  ( eq keyword ( quote restore ) )
  ( restore ( tail command ) )
( if
  ( if
    ( eq keyword ( quote fact ) )
    ( quote T )
    ( eq keyword ( quote forall ) ) )
  ( message
    ( quote ( asserted ) )
    globaln
    globalv
    ( cons command database ) )
  ( evaluation expression ) ) ) ) ) )
( keyword head command )
( name head' ( tail' command ) )
( value head' ( tail' ( tail' command ) ) )
( expression evaluate command globaln globalv it database )
( remove
  lambda
  ( a l )
  ( if
    ( eq a ( head l ) )
    ( tail l )
    ( cons
      ( head l )
      ( remove a ( tail l ) ( tail l ) ) ) ) ) )
```

```
( exit cons ( quote exit ) ( quote NIL ) )
( message
   lambda
   ( m n v d )
   ( cons
      ( quote message )
   ( cons
      m
   ( cons n ( cons v ( cons d ( quote NIL ) ) ) ) ) ) ) )
( evaluation
   lambda
   ( e )
   ( cons ( quote evaluation ) ( cons e ( quote NIL ) ) ) ) )
( error message ( quote ( Error ) ) globaln globalv )
( save
   lambda
   ( l globaln globalv database )
   ( letrec
      ( message
         ( cons
            ( cons
               ( quote restore )
               ( deflist
                  l
                  globaln
                  globalv
                  ( if
                     ( member ( quote database ) l )
                     database
                     ( quote NIL ) ) ) )
            ( quote NIL ) )
         globaln
         globalv
         database )
      ( deflist
         lambda
         ( l n v d )
         ( if
            ( eq n ( quote NIL ) )
            d
            ( deflist
               l
               ( tail n )
               ( tail v )
               ( if
                  ( member ( head n ) l )
                  ( cons
                     ( cons
                        ( quote def )
                     ( cons
                        ( head n )
                     ( cons ( head v ) ( quote NIL ) ) ) ) )
                     d )
                  d ) ) ) ) ) )
( restore lambda ( f ) ( cons ( quote restore ) f ) ) ) )
```

```
( evaluate
  letrec
  ( lambda
    ( e globaln globalv it database )
    ( letrec
      ( eval e n v )
      ( n
        cons
        ( cons ( quote it ) ( cons ( quote database ) globaln ) )
        ( quote NIL ) )
      ( v
        cons
        ( cons
          it
        ( cons ( assemble database ) ( listeval globalv n v ) ) )
        ( quote NIL ) ) ) )
  ( eval
    lambda
    ( e n v )
    ( letrec
      ( if
        ( atom e )
        ( associate e n v )
      ( if
        ( eq keyword ( quote quote ) )
        text1
      ( if
        ( eq keyword ( quote atom ) )
        ( if
          ( atom argument1 )
          ( quote T )
          ( isvariable argument1 ) )
      ( if
        ( eq keyword ( quote head ) )
        ( if
          ( indivisible argument1 )
          ( quote Error )
          ( head argument1 ) )
      ( if
        ( eq keyword ( quote tail ) )
        ( if
          ( indivisible argument1 )
          ( quote Error )
          ( tail argument1 ) )
      ( if
        ( eq keyword ( quote cons ) )
        ( cons
          ( if ( reserved argument1 ) ( quote Error ) argument1 )
          argument2 )
      ( if
        ( eq keyword ( quote eq ) )
        ( equal argument1 argument2 )
      ( if
        ( eq keyword ( quote leq ) )
        ( leq argument1 argument2 )
```

```
( if
   ( eq keyword ( quote add ) )
   ( add argument1 argument2 )
( if
   ( eq keyword ( quote sub ) )
   ( sub argument1 argument2 )
( if
   ( eq keyword ( quote mul ) )
   ( mul argument1 argument2 )
( if
   ( eq keyword ( quote div ) )
   ( if
      ( eq argument2 ( quote 0 ) )
      ( quote Error )
      ( div argument1 argument2 ) )
( if
   ( eq keyword ( quote rem ) )
   ( if
      ( eq argument2 ( quote 0 ) )
      ( quote Error )
      ( rem argument1 argument2 ) )
( if
   ( eq keyword ( quote if ) )
   ( if argument1 argument2 argument3 )
( if
   ( eq keyword ( quote lambda ) )
   ( makefunction text1 text2 n v )
( if
   ( eq keyword ( quote let ) )
   ( let
      ( eval text1 newn newv )
      ( newn cons ( names definitions ) n )
      ( newv cons ( listeval ( values definitions ) n v ) v ) )
( if
   ( eq keyword ( quote letrec ) )
   ( letrec
      ( eval text1 newn newv )
      ( newn cons ( names definitions ) n )
      ( newv
         cons
         ( listeval ( values definitions ) newn newv )
         v ) )
( if
   ( eq keyword ( quote chr ) )
   ( chr argument1 )
( if
   ( eq keyword ( quote all ) )
   ( solve
      text1
      ( tail' texts )
      ( associate ( quote database ) n v ) )
( if
   ( eq keyword ( quote logic ) )
   ( letrec
      ( eval text1 newn newv )
```

```
            ( newn cons ( quote ( database ) ) n )
            ( newv
              cons
              ( cons
                ( dbappend
                  ( assemble ( tail' texts ) )
                  ( associate ( quote database ) n v ) )
                ( quote NIL ) )
              v )
            ( dbappend
              lambda
              ( n o )
              ( if
                ( eq n ( quote NIL ) )
                o
              ( if
                ( atom n )
                ( quote NIL )
                ( cons
                  ( head n )
                  ( dbappend ( tail n ) o ) ) ) ) ) )
          ( letrec
            ( if
              ( isfunction applicand )
              ( eval body newn newv )
              ( quote Error ) )
            ( body tail ( head function ) )
            ( newn
              cons
              ( head ( head function ) )
              ( head ( tail function ) ) )
            ( newv cons arguments ( tail ( tail function ) ) )
            ( function tail applicand )
            ( applicand eval ( head e ) n v ))))) ))))) ))))) ))))) ))
        ( keyword head e )
        ( arguments listeval texts n v )
        ( argument1 head' arguments )
        ( argument2 head' ( tail' arguments ) )
        ( argument3 head' ( tail' ( tail' arguments ) ) )
        ( definitions assoclist' ( tail' texts ) )
        ( texts tail e )
        ( text1 head' texts )
        ( text2 head' ( tail' texts ) ) ) )
  ( listeval
    let
    ( lambda ( l n v ) ( map ( e n v ) l ) )
    ( e
      lambda
      ( n v )
      ( lambda ( x ) ( eval x n v ) ) ) )
  ( associate
    letrec
    ( lambda
      ( a n v )
      ( if
```

```
            ( eq n ( quote NIL ) )
            ( quote NotDefined )
         ( if
            ( member a ( head n ) )
            ( locate a ( head n ) ( head v ) )
            ( associate a ( tail n ) ( tail v ) ) ) ) )
      ( locate
         lambda
         ( a n v )
         ( if
            ( atom v )
            ( quote Error )
         ( if
            ( eq a ( head n ) )
            ( head v )
            ( locate a ( tail n ) ( tail v ) ) ) ) ) )
   ( names
      let
      ( lambda ( d ) ( map v d ) )
      ( v lambda ( b ) ( head b ) ) )
   ( values
      let
      ( lambda ( d ) ( map v d ) )
      ( v lambda ( b ) ( tail b ) ) )
   ( indivisible
      lambda
      ( c )
      ( if
         ( atom c )
         ( quote T )
      ( if ( isfunction c ) ( quote T ) ( isvariable c ) ) ) )
   ( solve
      letrec
      ( lambda
         ( variables constraints database )
         ( realise
            variables
            ( loop
               ( quote 1 )
               ( cons
                  ( cons
                     ( setvars
                        ( quote 0 )
                        ( markvars variables constraints ) )
                     ( quote NIL ) )
                  ( quote NIL ) )
               database ) ) )
      ( realise
         letrec
         ( lambda
            ( variables environments )
            ( map ( instantiate ( map sub0 variables ) ) environments ) )
         ( sub0
            lambda
            ( v )
```

```
      ( subscript ( quote 0 ) ( makevariable v ) ) )
   ( instantiate
      lambda
      ( v )
      ( lambda ( e ) ( instance v e ) ) )
   ( instance
      lambda
      ( v e )
      ( if
         ( isvariable v )
         ( if
            ( defined v e )
            ( instance ( associate v e ) e )
            v )
         ( if
            ( atom v )
            v
            ( cons
               ( instance ( head v ) e )
               ( instance ( tail v ) e ) ) ) ) ) ) )
( loop
   lambda
   ( level waiting database )
   ( letrec
      ( if
         ( eq waiting ( quote NIL ) )
         ( quote NIL )
         ( append
            solved
            ( loop
               ( add ( quote 1 ) level )
               resubmitted
               database ) ) )
      ( solved head deduction )
      ( resubmitted tail deduction )
      ( deduction deduce waiting ( setvars level database ) ) ) )
( deduce
   lambda
   ( waiting database )
   ( letrec
      ( if
         ( eq waiting ( quote NIL ) )
         ( quote ( NIL ) )
         ( cons solved resubmitted ) )
      ( solved
         if
         ( eq constraints ( quote NIL ) )
         ( cons environment solved' )
         solved' )
      ( resubmitted
         if
         ( eq constraints ( quote NIL ) )
         resubmitted'
         ( append
            ( resolve constraints environment database )
```

```
            resubmitted' ) )
        ( constraints head ( head waiting ) )
        ( environment tail ( head waiting ) )
        ( solved' head rest )
        ( resubmitted' tail rest )
        ( rest deduce ( tail waiting ) database ) ) )
  ( resolve
    lambda
    ( constraints environment database )
    ( letrec
      ( if
        ( eq database ( quote NIL ) )
        ( quote NIL )
      ( if
        ( eq unification ( quote impossible ) )
        rest
        ( cons ( cons relaxation unification ) rest ) ) )
      ( unification
        unify
        ( head constraints )
        ( head clause )
        environment )
      ( relaxation append ( tail clause ) ( tail constraints ) )
      ( clause head database )
      ( rest resolve constraints environment ( tail database ) ) ) ) )
  ( unify
    lambda
    ( a b substitution )
    ( letrec
      ( if
        ( equal a' b' )
        substitution
      ( if
        ( isvariable a' )
        ( bind a' b' substitution )
      ( if
        ( isvariable b' )
        ( bind b' a' substitution )
      ( if
        ( if ( atom a' ) ( quote T ) ( atom b' ) )
        ( quote impossible )
      ( if
        ( eq unifyheads ( quote impossible ) )
        ( quote impossible )
        unifytails ) ) ) ) )
      ( a' associate a substitution )
      ( b' associate b substitution )
      ( unifyheads unify ( head a' ) ( head b' ) substitution )
      ( unifytails unify ( tail a' ) ( tail b' ) unifyheads ) ) )
  ( defined
    lambda
    ( v e )
    ( if
      ( eq e ( quote NIL ) )
      ( quote F )
```

```
        ( if
          ( equal v ( head ( head e ) ) )
          ( quote T )
          ( defined v ( tail e ) ) ) ) )
    ( associate
      letrec
      ( lambda
        ( v e )
        ( if
          ( defined v e )
          ( associate ( immediate v e ) e )
          v ) )
      ( immediate
        lambda
        ( v e )
        ( if
          ( equal v ( head ( head e ) ) )
          ( tail ( head e ) )
          ( immediate v ( tail e ) ) ) ) )
    ( bind
      lambda
      ( n v e )
      ( cons ( cons n v ) e ) )
    ( setvars
      lambda
      ( i e )
      ( if
        ( isvariable e )
        ( subscript i e )
        ( if
          ( atom e )
          e
          ( cons
            ( setvars i ( head e ) )
            ( setvars i ( tail e ) ) ) ) ) )
    ( subscript
      lambda
      ( i v )
      ( makevariable ( cons i ( tail v ) ) ) ) )
( assemble
  lambda
  ( d )
  ( letrec
    ( if
      ( atom d )
      ( quote NIL )
      ( if
        ( atom clause )
        rest
        ( if
          ( eq keyword ( quote fact ) )
          ( cons ( tail clause ) rest )
          ( if
            ( eq keyword ( quote forall ) )
            ( cons
```

```
                ( markvars
                  ( head ( tail clause ) )
                  ( tail ( tail clause ) ) )
                rest )
              rest ) ) ) )
          ( clause head d )
          ( keyword head clause )
          ( rest assemble ( tail d ) ) ) )
      ( markvars
        lambda
        ( v e )
        ( if
          ( atom e )
          ( if ( member e v ) ( makevariable e ) e )
          ( cons
            ( markvars v ( head e ) )
            ( markvars v ( tail e ) ) ) ) ) ) ) )
( append
  lambda
  ( a b )
  ( if
    ( atom a )
    b
    ( cons ( head a ) ( append ( tail a ) b ) ) ) )
( member
  lambda
  ( a l )
  ( if
    ( atom l )
    ( quote F )
  ( if
    ( eq a ( head l ) )
    ( quote T )
    ( member a ( tail l ) ) ) ) )
( equal
  lambda
  ( a b )
  ( if
    ( eq a b )
    ( quote T )
  ( if
    ( if ( atom a ) ( quote T ) ( atom b ) )
    ( quote F )
  ( if
    ( equal ( head a ) ( head b ) )
    ( equal ( tail a ) ( tail b ) )
    ( quote F ) ) ) ) )
( map
  lambda
  ( f l )
  ( if
    ( atom l )
    l
    ( cons ( f ( head l ) ) ( map f ( tail l ) ) ) ) )
( head'
```

```
   lambda
   ( c )
   ( if ( atom c ) ( quote Error ) ( head c ) ) )
( tail'
   lambda
   ( c )
   ( if ( atom c ) ( quote Error ) ( tail c ) ) )
( assoclist'
   lambda
   ( l )
   ( if
      ( atom l )
      ( quote NIL )
   ( if
      ( atom ( head l ) )
      ( assoclist' ( tail l ) )
   ( if
      ( atom ( head ( head l ) ) )
      ( cons ( head l ) ( assoclist' ( tail l ) ) )
      ( assoclist' ( tail l ) ) ) ) ) )
( isvariable
   lambda
   ( v )
   ( if
      ( atom v )
      ( quote F )
      ( eq ( head v ) ( quote VariableTag ) ) ) )
( makevariable lambda ( v ) ( cons ( quote VariableTag ) v ) )
( isfunction
   lambda
   ( f )
   ( if
      ( atom f )
      ( quote F )
      ( eq ( head f ) ( quote FunctionTag ) ) ) )
( makefunction
   lambda
   ( formals body n v )
   ( cons
      ( quote FunctionTag )
   ( cons ( cons formals body ) ( cons n v ) ) ) )
( reserved
   lambda
   ( a )
   ( if
      ( eq a ( quote FunctionTag ) )
      ( quote T )
      ( eq a ( quote VariableTag ) ) ) ) )
```

```
( restore
  ( def
    list
    ( lambda
      ( l )
      ( if
        ( eq l ( quote NIL ) )
        ( quote NIL )
        ( cons
          newline
        ( cons ( head ( head l ) ) ( list ( tail l ) ) ) ) ) ) ) )
  ( def newline ( chr ( quote 13 ) ) )
  ( def
    palindromes
    ( logic
      ( list ( all ( x ) ( x <> x ) ) )
      ( forall ( x ) ( x = NIL + x ) )
      ( forall
        ( a x y z )
        ( ( a . x ) = ( a . y ) + z )
        ( x = y + z ) )
      ( fact ( NIL <> NIL ) )
      ( forall
        ( a x y z )
        ( ( a . x ) <> y )
        ( x <> z )
        ( y = z + ( a ) ) ) ) ) )
```

```
( letrec
  ( letrec
    ( interpreter ( quote muFP ) eval mufp_lib )
    ( interpreter
      lambda
      ( name eval predefined )
      ( letrec
        ( lambda
          ( kb )
          ( append
            ( list name ( quote interpreter ) )
            ( interpret kb ( quote NIL ) ) ) )
        ( interpret
          lambda
          ( kb env )
          ( letrec
            ( append
              ( list newline ( quote > ) )
              ( if
                ( eq first ( quote end ) )
                ( list ( quote Exit ) name ( quote interpreter ) )
              ( if
                ( eq first ( quote vars ) )
                ( append ( map head env ) irest )
              ( if
                ( eq first ( quote dump ) )
                ( append
                  ( reduce
                    append
                    ( cons
                      ( quote NIL )
                      ( map
                        ( lambda
                          ( x )
                          ( list ( showdef ( head x ) ) newline ) )
                        env ) ) )
                  irest )
              ( if
                ( atom first )
                irest
              ( if
                ( eq ( head first ) ( quote def ) )
                ( interpret
                  ( tail kb )
                  ( update
                    env
                    ( el ( quote 2 ) first )
                    ( el ( quote 3 ) first ) ) )
              ( if
                ( eq ( head first ) ( quote show ) )
                ( cons
                  ( pretty ( showdef ( el ( quote 2 ) first ) ) )
```

```
                irest )
            ( if
              ( eq ( head first ) ( quote cancel ) )
              ( interpret
                ( tail kb )
                ( unbind ( el ( quote 2 ) first ) env ) )
              ( if
                ( eq ( head first ) ( quote run ) )
                ( append
                  ( run
                    ( el ( quote 2 ) first )
                    ( untilend ( tail kb ) ) )
                  ( interpret ( afterend kb ) env ) )
                ( if
                  ( eq ( head first ) ( quote edit ) )
                  ( let
                    ( append
                      ( edit_output e )
                      ( interpret
                        ( afterend kb )
                        ( update
                          env
                          ( el ( quote 2 ) first )
                          ( edit_file e ) ) ) )
                    ( e
                      edit
                      ( associate ( el ( quote 2 ) first ) env )
                      ( untilend ( tail kb ) ) ) )
                  irest ) ) ) ) ) ) ) ) )
          ( showdef
            lambda
            ( n )
            ( list ( quote def ) n ( associate n env ) ) )
          ( irest interpret ( tail kb ) env )
          ( first head kb )
          ( run
            lambda
            ( exp input )
            ( letrec
              ( ( eval exp realenv ) input )
              ( realenv
                append
                ( map
                  ( lambda
                    ( x )
                    ( cons
                      ( head x )
                      ( eval ( tail x ) realenv ) ) )
                  env )
                predefined ) ) ) ) ) ) )
( eval
  lambda
  ( exp env )
  ( fp_eval
    ( ( mufp_eval ( lambda ( a ) ( defined a mufp_lib ) ) ) exp )
```

```
    env ) )
( mufp_eval
  lambda
  ( mfp )
  ( letrec
    mufp_eval
    ( mufp_eval
      lambda
      ( e )
      ( let
        ( if
          ( atom e )
          ( if ( mfp e ) stateless e )
        ( if
          ( eq ( head e ) ( quote select ) )
          stateless
        ( if
          ( eq ( head e ) ( quote constant ) )
          stateless
        ( if
          ( eq ( head e ) ( quote alpha ) )
          ( list
            ( quote compose )
            ( quote zip )
            ( list
              ( quote alpha )
              ( mufp_eval ( el ( quote 2 ) e ) ) ) )
            ( quote zip ) ) )
        ( if
          ( eq ( head e ) ( quote slash ) )
          ( list
            ( quote compose )
            ( list
              ( quote slash ) )
            ( list
              ( quote compose )
              ( mufp_eval ( el ( quote 2 ) e ) )
              ( quote zip ) ) )
            ( quote zip ) ) )
        ( if
          ( eq ( head e ) ( quote construct ) )
          ( list
            ( quote compose )
            ( quote zip )
            ( cons
              ( quote construct )
              ( map mufp_eval ( tail e ) ) ) ) )
        ( if
          ( eq ( head e ) ( quote if ) )
          ( list
            ( quote compose )
            ( quote
              ( alpha
                ( if
                  ( select 1 )
```

```
                    ( select 2 )
                    ( select 3 ) ) ) )
               ( quote zip )
               ( cons
                  ( quote construct )
                  ( map mufp_eval ( tail e ) ) ) )
            ( if
               ( eq ( head e ) ( quote mu ) )
               ( list
                  ( quote loop )
                  ( list
                     ( quote compose )
                     ( quote zip )
                     ( mufp_eval ( el ( quote 2 ) e ) )
                     ( quote zip ) )
                  ( el ( quote 3 ) e ) )
               ( if
                  ( eq ( head e ) ( quote compose ) )
                  ( cons ( quote compose ) ( map mufp_eval ( tail e ) ) )
                  ( quote **synerr** ) ) ) ) ) ) ) ) )
            ( stateless list ( quote alpha ) e ) ) ) ) )
( fp_eval
   lambda
   ( exp env )
   ( letrec
      ( fp_eval exp )
      ( fp_eval
         lambda
         ( f )
         ( if
            ( atom f )
            ( if
               ( defined f env )
               ( associate f env )
               ( lambda ( y ) ( quote **liberr** ) ) )
            ( if
               ( eq ( head f ) ( quote select ) )
               ( lambda
                  ( x )
                  ( el ( el ( quote 2 ) f ) x ) )
               ( if
                  ( eq ( head f ) ( quote compose ) )
                  ( reduce
                     ( lambda
                        ( x y )
                        ( lambda ( z ) ( x ( y z ) ) ) )
                     ( map fp_eval ( tail f ) ) )
                  ( if
                     ( eq ( head f ) ( quote construct ) )
                     ( construct ( map fp_eval ( tail f ) ) )
                     ( if
                        ( eq ( head f ) ( quote alpha ) )
                        ( let
                           ( lambda ( x ) ( map g x ) )
                           ( g fp_eval ( el ( quote 2 ) f ) ) )
```

```
        ( if
          ( eq ( head f ) ( quote slash ) )
          ( let
            ( lambda
              ( x )
              ( reduce
                ( lambda
                  ( x y )
                  ( g ( list x y ) ) )
                x ) )
            ( g fp_eval ( el ( quote 2 ) f ) ) )
        ( if
          ( eq ( head f ) ( quote constant ) )
          ( lambda ( x ) ( el ( quote 2 ) f ) )
        ( if
          ( eq ( head f ) ( quote loop ) )
          ( let
            ( lambda
              ( i )
              ( letrec
                ( el ( quote 1 ) p )
                ( p
                  g
                  ( list
                    )
                    ( cons
                      ( el ( quote 3 ) f )
                      ( el ( quote 2 ) p ) ) ) ) ) )
            ( g fp_eval ( el ( quote 2 ) f ) ) )
        ( if
          ( eq ( head f ) ( quote if ) )
          ( let
            ( lambda
              ( x )
              ( let
                ( if
                  ( eq y ( quote 1 ) )
                  ( b x )
                ( if
                  ( eq y ( quote 0 ) )
                  ( c x )
                  ( quote **iferr** ) ) )
                ( y a x ) ) )
            ( a fp_eval ( el ( quote 2 ) f ) )
            ( b fp_eval ( el ( quote 3 ) f ) )
            ( c fp_eval ( el ( quote 4 ) f ) ) )
          ( lambda
            ( x )
            ( quote **synerror** ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
( mufp_lib
  list
  ( cons
    ( quote appendl )
    ( lambda
      ( x )
```

```
      ( cons ( head x ) ( el ( quote 2 ) x ) ) ) )
  ( cons
    ( quote appendr )
    ( lambda
      ( x )
      ( append
        ( head x )
        ( list ( el ( quote 2 ) x ) ) ) ) )
  ( cons ( quote hd ) head )
  ( cons ( quote tl ) tail )
  ( cons
    ( quote distl )
    ( lambda
      ( x )
      ( map
        ( lambda ( y ) ( list ( head x ) y ) )
        ( el ( quote 2 ) x ) ) ) )
  ( cons
    ( quote distr )
    ( lambda
      ( x )
      ( map
        ( lambda
          ( y )
          ( list y ( el ( quote 2 ) x ) ) )
        ( head x ) ) ) )
  ( cons ( quote id ) ( lambda ( x ) x ) )
  ( cons
    ( quote zip )
    ( letrec
      zip
      ( zip
        lambda
        ( a )
        ( if
          ( atom a )
          a
          ( if
            ( atom ( head a ) )
            ( head a )
            ( let
              ( cons
                ( map head a' )
                ( zip ( map tail a' ) ) )
              ( a'
                map
                ( lambda
                  ( x )
                  ( if
                    ( atom x )
                    ( quote ( **ziperr** . **ziperr** ) )
                    x ) )
                a ) ) ) ) ) ) )
  ( cons
    ( quote eq )
```

```
            ( lambda
               ( x )
               ( if
                  ( eq
                     ( el ( quote 1 ) x )
                     ( el ( quote 2 ) x ) )
                  ( quote 1 )
                  ( quote 0 ) ) ) )
         ( cons
            ( quote null )
            ( lambda
               ( x )
               ( if
                  ( eq x ( quote NIL ) )
                  ( quote 1 )
                  ( quote 0 ) ) ) )
         ( cons
            ( quote add )
            ( lambda
               ( x )
               ( add
                  ( el ( quote 1 ) x )
                  ( el ( quote 2 ) x ) ) ) )
         ( cons
            ( quote sub )
            ( lambda
               ( x )
               ( sub
                  ( el ( quote 1 ) x )
                  ( el ( quote 2 ) x ) ) ) )
         ( cons
            ( quote mul )
            ( lambda
               ( x )
               ( mul
                  ( el ( quote 1 ) x )
                  ( el ( quote 2 ) x ) ) ) )
         ( cons
            ( quote div )
            ( lambda
               ( x )
               ( div
                  ( el ( quote 1 ) x )
                  ( el ( quote 2 ) x ) ) ) )
         ( cons
            ( quote rem )
            ( lambda
               ( x )
               ( rem
                  ( el ( quote 1 ) x )
                  ( el ( quote 2 ) x ) ) ) ) ) )
   ( reduce
     lambda
     ( f l )
     ( lf
```

```
      ( atom ( tail l ) )
      ( head l )
      ( f ( head l ) ( reduce f ( tail l ) ) ) ) ) )
( el
   lambda
   ( n l )
   ( if
      ( atom l )
      ( quote **elerr** )
   ( if
      ( eq n ( quote 1 ) )
      ( head l )
      ( el ( sub n ( quote 1 ) ) ( tail l ) ) ) ) ) )
( construct
   lambda
   ( l )
   ( if
      ( atom l )
      ( lambda ( x ) l )
      ( let
         ( lambda ( x ) ( cons ( f x ) ( g x ) ) )
         ( f head l )
         ( g construct ( tail l ) ) ) ) ) ) )
```

```
( def and ( if ( select 1 ) ( select 2 ) ( constant 0 ) ) )
( def or ( if ( select 1 ) ( constant 1 ) ( select 2 ) ) )
( def not ( if id ( constant 0 ) ( constant 1 ) ) )
( def xor ( compose and ( construct or ( compose not and ) ) ) )
( def half-adder ( construct xor and ) )
( def
  full-adder
  ( compose
    ( construct
      ( compose ( select 1 ) ( select 1 ) )
      ( compose
        or
        ( construct
          ( compose ( select 2 ) ( select 1 ) )
          ( select 2 ) ) ) )
    ( construct
      ( compose
        half-adder
        ( construct ( compose ( select 1 ) ( select 1 ) ) ( select 2 ) ) )
      ( compose ( select 2 ) ( select 1 ) ) )
    ( construct half-adder ( select 3 ) ) ) )
```

```
( letrec
   nFib
   ( nFib
      lambda
      ( n )
      ( if
         ( leq n ( quote 1 ) )
         ( quote 1 )
         ( add
            ( add
               ( nFib ( sub n ( quote 1 ) ) )
               ( nFib ( sub n ( quote 2 ) ) ) )
            ( quote 1 ) ) ) ) )
```

```
( letrec
   ( lambda ( kb ) ( map_until ( load_code ( head kb ) ) ( tail kb ) ) )
   ( map_until
     lambda
     ( f l )
     ( if
        ( eq ( head l ) ( quote end ) )
        ( quote NIL )
        ( cons
           ( f ( head l ) )
        ( cons newline ( map_until f ( tail l ) ) ) ) ) ) ) )
```

```
( letrec
   ( p ( n ( quote 2 ) ) )
   ( p
      lambda
      ( l )
      ( cons
         ( head l )
         ( p ( ( s ( head l ) ) ( tail l ) ) ) ) )
   ( s
      lambda
      ( p )
      ( letrec
         sp
         ( sp
            lambda
            ( l )
            ( if
               ( eq ( rem ( head l ) p ) ( quote 0 ) )
               ( sp ( tail l ) )
               ( cons ( head l ) ( sp ( tail l ) ) ) ) ) ) )
   ( n
      lambda
      ( x )
      ( cons x ( n ( add x ( quote 1 ) ) ) ) ) )
```

```
( letrec
  ( letrec
    X
    ( X cons ( quote 1 ) ( merge X2 ( merge X3 X5 ) ) )
    ( X2 times ( quote 2 ) X )
    ( X3 times ( quote 3 ) X )
    ( X5 times ( quote 5 ) X ) )
  ( merge
    lambda
    ( a b )
    ( if
      ( eq ( head a ) ( head b ) )
      ( merge a ( tail b ) )
    ( if
      ( leq ( head a ) ( head b ) )
      ( cons ( head a ) ( merge ( tail a ) b ) )
      ( cons ( head b ) ( merge a ( tail b ) ) ) ) ) )
  ( times
    lambda
    ( c l )
    ( cons ( mul c ( head l ) ) ( times c ( tail l ) ) ) ) )
```

```
( letrec
  ( lambda ( kb ) ( execute ( untilend ( tail kb ) ) ( head kb ) ) )
  ( execute
    lambda
    ( c db )
    ( if
      ( eq c ( quote NIL ) )
      ( quote NIL )
    ( if
      ( isadd ( head c ) )
      ( execute
        ( tail c )
        ( addset
          db
          ( eval ( arg1 ( head c ) ) db )
          ( mkatom ( arg2 ( head c ) ) )
          ( eval ( arg3 ( head c ) ) db ) ) )
    ( if
      ( isdb ( head c ) )
      ( cons db ( execute ( tail c ) db ) )
      ( cons
        ( eval ( head c ) db )
        ( execute ( tail c ) db ) ) ) ) ) )
  ( eval
    lambda
    ( e db )
    ( if
      ( isunion e )
      ( union
        ( eval ( arg1 e ) db )
        ( eval ( arg2 e ) db ) )
    ( if
      ( isinter e )
      ( intersection
        ( eval ( arg1 e ) db )
        ( eval ( arg2 e ) db ) )
    ( if
      ( isdiff e )
      ( difference
        ( eval ( arg1 e ) db )
        ( eval ( arg2 e ) db ) )
    ( if
      ( isim e )
      ( imageset
        ( eval ( arg1 e ) db )
        ( mkatom ( arg2 e ) )
        db )
    ( if
      ( isinvim e )
      ( invimage
        ( eval ( arg2 e ) db )
        ( mkatom ( arg1 e ) )
```

```
        db )
    ( if ( isatomlist e ) e emptyset ) ) ) ) ) ) )
( imageset
  lambda
  ( nodeset attr db )
  ( reduce
    union
    ( map ( lambda ( n ) ( image n attr db ) ) nodeset )
    emptyset ) )
( image
  lambda
  ( node attr db )
  ( if
    ( defined node db )
    ( let
      ( if ( defined attr a ) ( associate attr a ) emptyset )
      ( a associate node db ) )
    emptyset ) )
( invimage
  lambda
  ( nodeset attr db )
  ( filter
    ( lambda
      ( n )
      ( not
        ( eq
          ( intersection ( image n attr db ) nodeset )
          emptyset ) ) )
    ( domain db ) ) )
( addset
  lambda
  ( db ns1 attr ns2 )
  ( reduce
    ( lambda ( n db ) ( addel n attr ns2 db ) )
    ns1
    db ) )
( addel
  lambda
  ( node attr nodeset db )
  ( let
    ( let
      ( update db node ( update a attr ( union nodeset s ) ) )
      ( s if ( defined attr a ) ( associate attr a ) emptyset ) )
    ( a
      if
      ( defined node db )
      ( associate node db )
      ( quote NIL ) ) ) )
( isunion
  lambda
  ( e )
  ( and
    ( not ( atom e ) )
    ( eq ( head e ) ( quote union ) ) ) )
( isinter
```

```
      lambda
      ( e )
      ( and
         ( not ( atom e ) )
         ( eq ( head e ) ( quote inter ) ) ) )
   ( isdiff
      lambda
      ( e )
      ( and
         ( not ( atom e ) )
         ( eq ( head e ) ( quote diff ) ) ) )
   ( isim
      lambda
      ( e )
      ( and
         ( not ( atom e ) )
         ( eq ( head e ) ( quote im ) ) ) )
   ( isinvim
      lambda
      ( e )
      ( and
         ( not ( atom e ) )
         ( eq ( head e ) ( quote inv ) ) ) )
   ( isadd
      lambda
      ( e )
      ( and
         ( not ( atom e ) )
         ( eq ( head e ) ( quote add ) ) ) )
   ( isdb lambda ( e ) ( eq e ( quote db ) ) )
   ( isatomlist
      lambda
      ( e )
      ( or
         ( eq e ( quote NIL ) )
         ( and
            ( not ( atom e ) )
         ( and ( atom ( head e ) ) ( isatomlist ( tail e ) ) ) ) ) )
   ( arg1
      lambda
      ( e )
      ( if
         ( leq ( quote 2 ) ( length e ) )
         ( head ( tail e ) )
         ( quote NIL ) ) )
   ( arg2
      lambda
      ( e )
      ( if
         ( leq ( quote 3 ) ( length e ) )
         ( head ( tail ( tail e ) ) )
         ( quote NIL ) ) )
   ( arg3
      lambda
      ( e )
```

```
  ( if
    ( leq ( quote 4 ) ( length e ) )
    ( head ( tail ( tail ( tail e ) ) ) )
    ( quote NIL ) ) )
( mkatom lambda ( x ) ( if ( atom x ) x ( quote NIL ) ) ) )
```

```
( ( John ( loves Mary Logic ) ( isa Male Programmer ) )
  ( Mary ( loves John Logic Whiskey ) ( isa Female Programmer ) )
  ( Giving1 ( isa Giving ) ( giver John ) ( givee Mary ) ( given Book ) )
  ( Giving2 ( isa Giving ) ( giver John ) ( givee Mary ) ( given Flowers ) )
  ( Giving3 ( isa Giving ) ( giver Mary ) ( givee John ) ( given Kiss ) ) )
```

```
( lambda
   ( library )
   ( cons ( quote letrec ) ( cons ( quote *** ) library ) ) )
```

```
( lambda ( kb ) ( printerrors ( syntax ( head kb ) ) ) )
```

ASSOCIATION LIB

```
( ( domain
    lambda
    ( a )
    ( if
      ( atom a )
      ( quote NIL )
      ( cons ( head ( head a ) ) ( domain ( tail a ) ) ) ) )
  ( defined
    lambda
    ( e a )
    ( unless
      ( atom a )
      ( or
        ( eq ( head ( head a ) ) e )
        ( defined e ( tail a ) ) ) ) )
  ( associate
    lambda
    ( e a )
    ( if
      ( eq ( head ( head a ) ) e )
      ( tail ( head a ) )
      ( associate e ( tail a ) ) ) )
  ( bind lambda ( e d a ) ( cons ( cons e d ) a ) )
  ( unbind
    letrec
    ( lambda
      ( e a )
      ( if ( defined e a ) ( u e a ) a ) )
    ( u
      lambda
      ( e a )
      ( if
        ( eq ( head ( head a ) ) e )
        ( tail a )
        ( cons ( head a ) ( u e ( tail a ) ) ) ) ) )
  ( update
    letrec
    ( lambda
      ( a e d )
      ( if
        ( defined e a )
        ( u e d a )
        ( bind e d a ) ) )
    ( u
      lambda
      ( e d a )
      ( if
        ( eq ( head ( head a ) ) e )
        ( cons ( cons e d ) ( tail a ) )
```

(cons (head a) (u e d (tail a)))))))

```
( ( edit
    lambda
    ( f l )
    ( editloop l ( state f ( quote NIL ) f ) ) )
  ( editloop
    lambda
    ( i t )
    ( letrec
      ( if
        ( atom i )
        ( return_file ( head ( top f s d ) ) )
      ( if
        ( eq key ( quote print ) )
        ( return_message ( list newline file ) ( editloop ( tail i ) t' ) )
      ( if
        ( eq key ( quote continue ) )
        ( editloop ( tail i ) t' )
        ( return_message
          ( list newline ( quote Error ) )
          ( editloop ( tail i ) t ) ) ) ) )
      ( key if ( atom step ) ( quote error ) ( head step ) )
      ( file head ( tail step ) )
      ( t' head ( tail ( tail step ) ) )
      ( step editstep ( head i ) t )
      ( f head t )
      ( s head ( tail t ) )
      ( d head ( head ( tail t ) ) ) ) ) )
  ( editstep
    lambda
    ( c t )
    ( let
      ( if
        ( atom c )
        ( if
          ( eq c ( quote d ) )
          ( ( strict step ) ( delete1 f s d ) )
        ( if
          ( eq c ( quote file ) )
          ( print f t )
        ( if
          ( eq c ( quote p ) )
          ( print ( pretty ( dump ( quote 3 ) f ) ) t )
        ( if
          ( eq c ( quote top ) )
          ( step ( top f s d ) )
        ( if
          ( eq c ( quote u ) )
          ( ( strict step ) ( up f s d ) )
        ( if
          ( eq c ( quote undelete ) )
          ( step ( undelete f s d ) )
        ( if
```

```
      ( number c )
      ( ( strict step ) ( move c f s d ) )
      ( quote error ) ) ) ) ) ) ) )
( if
   ( and ( eq key ( quote a ) ) twoargs )
   ( ( strict step ) ( after pattern template f s d ) )
( if
   ( and ( eq key ( quote b ) ) twoargs )
   ( ( strict step ) ( before pattern template f s d ) )
( if
   ( and ( eq key ( quote c ) ) twoargs )
   ( ( strict step ) ( change pattern template f s d ) )
( if
   ( and ( eq key ( quote d ) ) onearg )
   ( ( strict step ) ( delete pattern f s d ) )
( if
   ( and ( eq key ( quote e ) ) twoargs )
   ( ( strict step ) ( exchange pattern template f s d ) )
( if
   ( and ( eq key ( quote f ) ) onearg )
   ( ( strict step ) ( find pattern f s d ) )
( if
   ( and
      ( eq key ( quote g ) )
   ( and twoargs ( not ( number ( head ( tail c ) ) ) ) ) )
   ( step ( global pattern template f s d ) )
( if
   ( and ( eq key ( quote p ) ) onearg )
   ( if
      ( number argument )
      ( print ( pretty ( dump argument f ) ) t )
   ( if
      ( eq argument ( quote all ) )
      ( print ( pretty f ) t )
   ( if
      ( eq argument ( quote file ) )
      ( print f t )
      ( quote error ) ) ) )
( if
   ( and ( eq key ( quote r ) ) onearg )
   ( step ( replace argument f s d ) )
   ( quote error ) ) ) ) ) ) ) ) ) )
( key head c )
( onearg
   unless
   ( atom ( tail c ) )
   ( eq ( tail ( tail c ) ) ( quote NIL ) ) )
( twoargs
   unless
   ( atom ( tail c ) )
   ( unless
      ( atom ( tail ( tail c ) ) )
      ( eq ( tail ( tail ( tail c ) ) ) ( quote NIL ) ) ) )
( argument head ( tail c ) )
( pattern compilep ( head ( tail c ) ) )
```

```
( template compile1 ( head ( tail ( tail c ) ) ) )
( f head t )
( s head ( tail t ) )
( d head ( tail ( tail t ) ) ) ) ) )
```

```
( ( up
    lambda
    ( f s d )
    ( if
      ( eq s ( quote NIL ) )
      ( quote error )
      ( state ( ( head s ) f ) ( tail s ) d ) ) )
  ( top
    lambda
    ( f s d )
    ( if
      ( eq s ( quote NIL ) )
      ( state f s d )
      ( top ( ( head s ) f ) ( tail s ) d ) ) )
  ( undelete lambda ( f s d ) ( state d s f ) )
  ( move
    lambda
    ( n f s d )
    ( letrec
      ( m n f newcursor )
      ( m
        lambda
        ( n f' k )
        ( if
          ( atom f' )
          ( if
            ( eq n ( quote 1 ) )
            ( state f' ( cons k s ) d )
            ( quote error ) )
          ( if
            ( eq n ( quote 1 ) )
            ( state
              ( head f' )
              ( cons ( keeptail f' k ) s )
              d )
            ( m
              ( sub n ( quote 1 ) )
              ( tail f' )
              ( keephead f' k ) ) ) ) ) ) )
  ( after
    lambda
    ( pattern template f s d )
    ( letrec
      ( a f newcursor )
      ( a
        lambda
        ( f' k )
        ( let
          ( if
            ( atom f' )
            ( quote error )
          ( if
```

```
                ( eq  env_head ( quote error ) )
                ( a  ( tail f' ) ( keephead f' k ) )
                ( state
                   ( k
                      ( cons
                         ( head f' )
                         ( cons ( template env_head ) ( tail f' ) ) ) )
                      s
                      f ) ) )
             ( env_head pattern ( head f' ) ) ) ) ) )
   ( before
      lambda
      ( pattern template f s d )
      ( letrec
         ( b f newcursor )
         ( b
            lambda
            ( f' k )
            ( let
               ( if
                  ( atom f' )
                  ( if
                     ( eq env_all ( quote error ) )
                     ( quote error )
                     ( state
                        ( k ( cons ( template env_all ) f' ) )
                        s
                        f ) )
               ( if
                  ( eq env_head ( quote error ) )
                  ( b ( tail f' ) ( keephead f' k ) )
                  ( state ( k ( cons ( template env_head ) f' ) ) s f ) ) )
               ( env_all pattern f' )
               ( env_head pattern ( head f' ) ) ) ) ) )
   ( change
      lambda
      ( pattern template f s d )
      ( let
         ( if
            ( eq env_all ( quote error ) )
            ( quote error )
            ( state ( template env_all ) s f ) )
         ( env_all pattern f ) ) )
   ( delete
      lambda
      ( pattern f s d )
      ( letrec
         ( e f newcursor )
         ( e
            lambda
            ( f' k )
            ( if
               ( atom f' )
               ( if
                  ( eq ( pattern f' ) ( quote error ) )
```

```
            ( quote error )
            ( state ( k ( quote NIL ) ) s f ) )
         ( if
            ( eq ( pattern ( head f' ) ) ( quote error ) )
            ( e ( tail f' ) ( keephead f' k ) )
            ( state ( k ( tail f' ) ) s f ) ) ) ) ) )
( delete1
   lambda
   ( f s d )
   ( if
      ( atom f )
      ( quote error )
      ( state ( tail f ) s f ) ) )
( exchange
   lambda
   ( pattern template f s d )
   ( letrec
      ( e f newcursor )
      ( e
         lambda
         ( f' k )
         ( let
            ( if
               ( atom f' )
               ( if
                  ( eq env_all ( quote error ) )
                  ( quote error )
                  ( state ( k ( template env_all ) ) s f ) )
               ( if
                  ( eq env_head ( quote error ) )
                  ( e ( tail f' ) ( keephead f' k ) )
                  ( state
                     ( k ( cons ( template env_head ) ( tail f' ) ) )
                     s
                     f ) ) )
            ( env_all pattern f' )
            ( env_head pattern ( head f' ) ) ) ) ) ) )
( find
   lambda
   ( pattern f s d )
   ( letrec
      ( g f newcursor s )
      ( g
         lambda
         ( f' k s' )
         ( let
            ( if ( eq across ( quote error ) ) down across )
            ( across g' f' k s' )
            ( down g'' f' k s' ) ) )
      ( g'
         lambda
         ( f' k s' )
         ( if
            ( atom f' )
            ( if
```

```
                ( eq ( pattern f' ) ( quote error ) )
                ( quote error )
                ( state f' ( cons k s' ) d ) )
            ( if
              ( eq ( pattern ( head f' ) ) ( quote error ) )
              ( g' ( tail f' ) ( keephead f' k ) s' )
              ( state
                ( head f' )
                ( cons ( keeptail f' k ) s' )
                d ) ) ) )
        ( g''
          lambda
          ( f' k s' )
          ( let
            ( if
              ( atom f' )
              ( quote error )
              ( if
                ( eq component ( quote error ) )
                ( g'' ( tail f' ) ( keephead f' k ) s' )
                component ) )
              ( component
                g
                ( head f' )
                newcursor
                ( cons ( keeptail f' k ) s' ) ) ) ) ) ) )
    ( global
      lambda
      ( pattern template f s d )
      ( letrec
        ( state ( g f ) s f )
        ( g
          lambda
          ( f' )
          ( let
            ( if
              ( eq env_all ( quote error ) )
              ( if
                ( atom f' )
                f'
                ( cons
                  ( if
                    ( eq env_head ( quote error ) )
                    ( g ( head f' ) )
                    ( template ( g' env_head ) ) )
                  ( g ( tail f' ) ) ) )
              ( template ( g' env_all ) ) )
            ( env_all pattern f' )
            ( env_head pattern ( head f' ) ) ) )
        ( g'
          lambda
          ( env )
          ( lambda ( a ) ( g ( env a ) ) ) ) ) )
    ( replace lambda ( template f s d ) ( state template s f ) ) )
```

```
( ( compilep
    letrec
    ( lambda
      ( pat )
      ( let
        ( lambda ( x ) ( c x ( lambda ( x ) x ) ) )
        ( c head ( cp pat ( quote NIL ) ) ) ) )
    ( cp
      lambda
      ( p v )
      ( letrec
        ( if
          ( number p )
          ( if
            ( member p v )
            ( cons ( oldvar p ) v )
            ( cons ( newvar p ) ( cons p v ) ) )
          ( if
            ( atom p )
            ( cons ( atomic p ) v )
            ( cons
              ( composite p ( head h ) ( head t ) )
              ( tail t ) ) ) )
        ( h cp ( head p ) v )
        ( t cp ( tail p ) ( tail h ) ) ) )
    ( oldvar
      lambda
      ( p )
      ( lambda
        ( x e )
        ( if
          ( unless
            ( eq e ( quote error ) )
            ( equal x ( e p ) ) )
          e
          ( quote error ) ) ) )
    ( newvar
      lambda
      ( p )
      ( lambda
        ( x e )
        ( if
          ( eq e ( quote error ) )
          ( quote error )
          ( lambda
            ( a )
            ( if ( eq a p ) x ( e a ) ) ) ) ) )
    ( atomic
      lambda
      ( p )
      ( lambda
        ( x e )
```

```
        ( if ( eq x p ) e ( quote error ) ) ) )
    ( composite
      lambda
      ( p h t )
      ( lambda
        ( x e )
        ( if
          ( or ( atom x ) ( eq e ( quote error ) ) )
          ( quote error )
          ( t ( tail x ) ( h ( head x ) e ) ) ) ) ) ) )
( compilet
  lambda
  ( tem )
  ( lambda
    ( e )
    ( letrec
      ( ct tem )
      ( ct
        lambda
        ( t )
        ( if
          ( number t )
          ( e t )
          ( if
            ( atom t )
            t
            ( cons ( ct ( head t ) ) ( ct ( tail t ) ) ) ) ) ) ) ) ) ) )
```

```
( ( newcursor lambda ( x ) x )
  ( keephead
    lambda
    ( f k )
    ( lambda ( x ) ( k ( cons ( head f ) x ) ) ) )
  ( keeptail
    lambda
    ( f k )
    ( lambda ( x ) ( k ( cons x ( tail f ) ) ) ) )
  ( edit_file lambda ( r ) ( head r ) )
  ( edit_output lambda ( r ) ( tail r ) )
  ( return_file lambda ( f ) ( cons f ( quote NIL ) ) )
  ( return_message
    lambda
    ( m r )
    ( cons ( edit_file r ) ( append m ( edit_output r ) ) ) )
  ( state
    lambda
    ( f s d )
    ( cons f ( cons s ( cons d ( quote NIL ) ) ) ) )
  ( print
    lambda
    ( f t )
    ( cons ( quote print ) ( cons f ( cons t ( quote NIL ) ) ) ) )
  ( step
    lambda
    ( t )
    ( cons
      ( quote continue )
    ( cons ( quote NIL ) ( cons t ( quote NIL ) ) ) ) )
  ( strict
    lambda
    ( f )
    ( lambda
      ( t )
      ( if
        ( eq t ( quote error ) )
        ( quote error )
        ( f t ) ) ) ) )
```

```
( ( head'
    lambda
    ( c )
    ( if ( atom  c ) ( quote Error ) ( head c ) ) )
  ( tail'
    lambda
    ( c )
    ( if ( atom  c ) ( quote Error ) ( tail c ) ) )
  ( associst'
    lambda
    ( l )
    ( if
      ( atom l )
      ( quote NIL )
    ( if
      ( atom ( head l ) )
      ( associst' ( tail l ) )
    ( if
      ( atom ( head ( head l ) ) )
      ( cons ( head l ) ( associst' ( tail l ) ) )
      ( associst' ( tail l ) ) ) ) ) )
  ( isfunction
    lambda
    ( f )
    ( unless ( atom f ) ( eq ( head f ) ( quote __function__ ) ) ) )
  ( makefunction
    lambda
    ( formals body n v )
    ( cons
      ( quote __function__ )
    ( cons ( cons formals body ) ( cons n v ) ) ) )
  ( showfunction
    lambda
    ( f )
    ( let
      ( cons ( head' l ) ( cons ( tail' l ) ( quote NIL ) ) )
      ( l head' ( tail f ) ) ) )
  ( iscode
    lambda
    ( k )
    ( unless ( atom k ) ( eq ( head k ) ( quote ____code____ ) ) ) )
  ( makecode lambda ( k ) ( cons ( quote ____code____ ) k ) )
  ( showcode
    lambda
    ( k )
    ( let
      ( cons
        ( head' l )
      ( cons ( quote in ) ( cons ( tail' l ) ( quote NIL ) ) ) )
      ( l tail k ) ) )
  ( reserved
    lambda
```

```
( a )
( member a ( quote ( __function__ _____code_____ ) ) ) ) )
```

```
( ( comp
    lambda
    ( e n c )
    ( if
      ( atom e )
      ( cons LD_code ( cons ( location e n ) ( cons AP0_code c ) ) )
    ( if
      ( eq ( head e ) ( quote quote ) )
      ( cons LDC_code ( cons ( head ( tail e ) ) c ) )
    ( if
      ( eq ( head e ) ( quote add ) )
      ( comp
        ( head ( tail e ) )
        n
      ( comp
        ( head ( tail ( tail e ) ) )
        n
        ( cons ADD_code c ) ) )
    ( if
      ( eq ( head e ) ( quote sub ) )
      ( comp
        ( head ( tail e ) )
        n
      ( comp
        ( head ( tail ( tail e ) ) )
        n
        ( cons SUB_code c ) ) )
    ( if
      ( eq ( head e ) ( quote mul ) )
      ( comp
        ( head ( tail e ) )
        n
      ( comp
        ( head ( tail ( tail e ) ) )
        n
        ( cons MUL_code c ) ) )
    ( if
      ( eq ( head e ) ( quote div ) )
      ( comp
        ( head ( tail e ) )
        n
      ( comp
        ( head ( tail ( tail e ) ) )
        n
        ( cons DIV_code c ) ) )
    ( if
      ( eq ( head e ) ( quote rem ) )
      ( comp
        ( head ( tail e ) )
        n
      ( comp
        ( head ( tail ( tail e ) ) )
```

```
      n
      ( cons REM_code c ) ) )
  ( if
    ( eq ( head e ) ( quote leq ) )
    ( comp
      ( head ( tail e ) )
      n
    ( comp
      ( head ( tail ( tail e ) ) )
      n
      ( cons LEQ_code c ) ) )
  ( if
    ( eq ( head e ) ( quote eq ) )
    ( comp
      ( head ( tail e ) )
      n
    ( comp
      ( head ( tail ( tail e ) ) )
      n
      ( cons EQ_code c ) ) )
  ( if
    ( eq ( head e ) ( quote head ) )
    ( comp
      ( head ( tail e ) )
      n
      ( cons HEAD_code ( cons AP0_code c ) ) )
  ( if
    ( eq ( head e ) ( quote tail ) )
    ( comp
      ( head ( tail e ) )
      n
      ( cons TAIL_code ( cons AP0_code c ) ) )
  ( if
    ( eq ( head e ) ( quote atom ) )
    ( comp ( head ( tail e ) ) n ( cons ATOM_code c ) )
  ( if
    ( eq ( head e ) ( quote cons ) )
    ( complazy
      ( head ( tail ( tail e ) ) )
      n
    ( complazy ( head ( tail e ) ) n ( cons CONS_code c ) ) )
  ( if
    ( eq ( head e ) ( quote if ) )
    ( let
      ( comp
        ( head ( tail e ) )
        n
        ( cons SEL_code ( cons thenpart ( cons elsepart c ) ) ) )
      ( thenpart comp ( head ( tail ( tail e ) ) ) n JOIN_seq )
      ( elsepart
        comp
        ( head ( tail ( tail ( tail e ) ) ) )
        n
        JOIN_seq ) )
  ( if
```

```
        ( eq ( head e ) ( quote lambda ) )
        ( let
           ( cons LDF_code ( cons body c ) )
           ( body
              comp
              ( head ( tail ( tail e ) ) )
              ( cons ( head ( tail e ) ) n )
              RTN_seq ) )
     ( if
        ( eq ( head e ) ( quote let ) )
        ( let
           ( let
              ( complist
                 args
                 n
                 ( cons LDF_code ( cons body ( cons AP_code c ) ) ) ) )
              ( body comp ( head ( tail e ) ) m RTN_seq ) )
           ( m cons ( domain ( tail ( tail e ) ) ) n )
           ( args exprs ( tail ( tail e ) ) ) )
     ( if
        ( eq ( head e ) ( quote letrec ) )
        ( let
           ( let
              ( cons
                 DUM_code
                 ( complist
                    args
                    m
                    ( cons
                       LDF_code
                       ( cons body ( cons RAP_code c ) ) ) ) ) )
              ( body comp ( head ( tail e ) ) m RTN_seq ) )
           ( m cons ( domain ( tail ( tail e ) ) ) n )
           ( args exprs ( tail ( tail e ) ) ) )
        ( complist
           ( tail e )
           n
           ( comp ( head e ) n ( cons AP_code c ))))) ))))) ))))) ))))) ) )
  ( complist
     lambda
     ( e n c )
     ( if
        ( eq e ( quote NIL ) )
        ( cons LDC_code ( cons ( quote NIL ) c ) )
        ( complist
           ( tail e )
           n
           ( complazy ( head e ) n ( cons CONS_code c ) ) ) ) ) )
  ( complazy
     lambda
     ( e n c )
     ( cons LDE_code ( cons ( comp e n UPD_seq ) c ) ) )
  ( location
     lambda
     ( e n )
```

```
    ( letrec
       ( if
          ( member e ( head n ) )
          ( cons ( quote 0 ) ( posn e ( head n ) ) )
          ( inchead ( location e ( tail n ) ) ) )
       ( posn
          lambda
          ( e n )
          ( if
             ( eq e ( head n ) )
             ( quote 0 )
             ( add ( quote 1 ) ( posn e ( tail n ) ) ) ) )
       ( inchead
          lambda
          ( i )
          ( cons ( add ( quote 1 ) ( head i ) ) ( tail i ) ) ) ) )
( exprs
   lambda
   ( d )
   ( if
      ( eq d ( quote NIL ) )
      ( quote NIL )
      ( cons ( tail ( head d ) ) ( exprs ( tail d ) ) ) ) )
( freevars
   lambda
   ( e )
   ( if
      ( atom e )
      ( singleton e )
   ( if
      ( eq ( head e ) ( quote quote ) )
      emptyset
   ( if
      ( eq ( head e ) ( quote lambda ) )
      ( let
         ( difference ( freevars body ) arguments )
         ( body head ( tail ( tail e ) ) )
         ( arguments head ( tail e ) ) )
   ( if
      ( eq ( head e ) ( quote let ) )
      ( let
         ( reduce
            union
            ( map ( lambda ( d ) ( freevars ( tail d ) ) ) definitions )
            ( difference ( freevars body ) ( domain definitions ) ) )
         ( body head ( tail e ) )
         ( definitions tail ( tail e ) ) )
   ( if
      ( eq ( head e ) ( quote letrec ) )
      ( let
         ( difference
            ( reduce
               union
               ( map ( lambda ( d ) ( freevars ( tail d ) ) ) definitions )
               ( freevars body ) )
```

```
            ( domain definitions ) )
          ( body  head ( tail e ) )
          ( definitions tail ( tail e ) ) )
        ( reduce
          union
          ( map
            freevars
            ( if ( member ( head e ) operators ) ( tail e ) e ) )
          emptyset ) ) ) ) ) ) )
  ( structure
    let
    ( lambda
      ( e )
      ( if
        ( or ( atom e ) ( eq ( head e ) ( quote quote ) ) )
        token
        ( if
          ( or
            ( eq  ( head e ) ( quote letrec ) )
            ( eq  ( head e ) ( quote let ) ) )
          ( cons
            ( head e )
          ( cons
            ( structure ( head ( tail e ) ) )
            ( map
              ( lambda
                ( d )
                ( cons ( head d ) ( structure ( tail d ) ) ) )
              ( tail ( tail e ) ) ) ) )
        ( if
          ( eq ( head e ) ( quote lambda ) )
          ( list
            ( head e )
            ( head ( tail e ) )
            ( structure ( head ( tail ( tail e ) ) ) ) )
          ( reduce
            ( lambda
              ( h t )
              ( if
                ( and ( atom h ) ( atom t ) )
                token
                ( cons h t ) ) )
            ( map structure e )
            ( quote NIL ) ) ) ) ) ) )
    ( token quote * ) )
  ( operators append monadic_ops ( append diadic_ops triadic_ops ) )
  ( monadic_ops quote ( head tail atom ) )
  ( diadic_ops quote ( add sub mul div rem leq eq cons ) )
  ( triadic_ops quote ( if ) ) )
```

```
( ( LD_code quote 1 )
  ( LOC_code quote 2 )
  ( LOF_code quote 3 )
  ( AP_code quote 4 )
  ( RTN_code quote 5 )
  ( RTN_seq quote ( 5 ) )
  ( DUM_code quote 6 )
  ( RAP_code quote 7 )
  ( SEL_code quote 8 )
  ( JOIN_code quote 9 )
  ( JOIN_seq quote ( 9 ) )
  ( HEAD_code quote 10 )
  ( TAIL_code quote 11 )
  ( ATOM_code quote 12 )
  ( CONS_code quote 13 )
  ( EQ_code quote 14 )
  ( ADD_code quote 15 )
  ( SUB_code quote 16 )
  ( MUL_code quote 17 )
  ( DIV_code quote 18 )
  ( REM_code quote 19 )
  ( LEQ_code quote 20 )
  ( STOP_code quote 21 )
  ( STOP_seq quote ( 21 ) )
  ( LDE_code quote 22 )
  ( UPD_code quote 23 )
  ( UPD_seq quote ( 23 ) )
  ( AP0_code quote 24 )
  ( READ_code quote 25 )
  ( PRINT_code quote 26 ) )
```

```
( ( run_and_halt quote ( ( 1 ( 0  0 ) 24 21 ) ) )
  ( print_item
    quote
    ( ( 1 ( 0  0 ) 24 26 1 ( 0  0 ) 5 ) ) )
  ( read_item quote ( ( 25 5 ) ) )
  ( chr quote ( ( 1 ( 0  0 ) 24 27 5 ) ) )
  ( apply_code
    quote
    ( ( 1 ( 0  1 ) 24 1 ( 0 . 0 ) 24 4 5 ) ) )
  ( flexible
    lambda
    ( f )
    ( let
      ( strict_cons code environment )
      ( code
        quote
        ( 2 NIL 3 NIL 11 10 13 1 ( 1 . 0 ) 24 4 5 ) )
      ( environment
        strict_cons
        ( strict_cons f ( quote NIL ) )
        ( quote NIL ) ) ) )
  ( strict_cons
    quote
    ( ( 1 ( 0 . 1 ) 24 1 ( 0 . 0 ) 24 13 5 ) ) )
  ( sequence
    quote
    ( ( 1 ( 0  0 ) 24 1 ( 0  1 ) 13 10 24 5 ) ) )
  ( make_closure
    lambda
    ( f )
    ( sequence
      ( inspect_code ( head f ) )
      ( sequence ( inspect_env ( tail f ) ) f ) ) )
  ( make_arglist lambda ( l ) ( sequence ( inspect_arglist l ) l ) )
  ( inspect_code lambda ( f ) ( finite f ) )
  ( inspect_env
    lambda
    ( e )
    ( if
      ( atom e )
      ( quote T )
      ( sequence
        ( inspect_arglist ( head e ) )
        ( inspect_env ( tail e ) ) ) ) )
  ( inspect_arglist
    lambda
    ( l )
    ( if ( atom l ) ( quote T ) ( inspect_arglist ( tail l ) ) ) )
  ( finite
    lambda
    ( e )
    ( if
```

```
    ( atom  e )
    ( quote  T )
  ( if
    ( finite  ( head  e ) )
    ( finite  ( tail  e ) )
    ( quote  F ) ) ) ) )
```

```
( ( emptyset quote NIL )
  ( singleton lambda ( e ) ( cons e ( quote NIL ) ) )
  ( addelement
    lambda
    ( e l )
    ( if ( member e l ) l ( cons e l ) ) )
  ( remelement
    lambda
    ( e l )
    ( if
      ( atom l )
      ( quote NIL )
      ( if
        ( eq e ( head l ) )
        ( tail l )
        ( cons ( head l ) ( remelement e ( tail l ) ) ) ) ) )
  ( union
    lambda
    ( a b )
    ( if
      ( atom a )
      b
      ( if
        ( member ( head a ) b )
        ( union ( tail a ) b )
        ( union ( tail a ) ( cons ( head a ) b ) ) ) ) )
  ( intersection
    lambda
    ( a b )
    ( if
      ( atom a )
      ( quote NIL )
      ( if
        ( member ( head a ) b )
        ( cons ( head a ) ( intersection ( tail a ) b ) )
        ( intersection ( tail a ) b ) ) ) )
  ( difference
    lambda
    ( a b )
    ( if
      ( atom a )
      ( quote NIL )
      ( if
        ( member ( head a ) b )
        ( difference ( tail a ) b )
        ( cons ( head a ) ( difference ( tail a ) b ) ) ) ) ) )
```

```
( ( dump
    let
    ( lambda
      ( n f )
      ( if
        ( atom f )
        f
      ( if
        ( leq n ( quote 0 ) )
        token
        ( let
          ( if
            ( and
              ( eq h token )
              ( if
                ( atom t )
                ( eq t token )
                ( and
                  ( eq ( head t ) token )
                  ( eq ( tail t ) ( quote NIL ) ) ) ) ) )
            token
            ( cons h t ) )
          ( h dump ( sub n ( quote 1 ) ) ( head f ) )
          ( t dump n ( tail f ) ) ) ) ) )
    ( token quote * ) )
  ( flatten
    letrec
    ( lambda
      ( s c )
      ( if
        ( atom s )
        ( cons s c )
        ( cons open ( flattentail s c ) ) ) )
    ( flattentail
      lambda
      ( s c )
      ( if
        ( eq s ( quote NIL ) )
        ( cons close c )
      ( if
        ( atom s )
        ( cons point ( cons s ( cons close c ) ) )
        ( flatten ( head s ) ( flattentail ( tail s ) c ) ) ) ) ) )
    ( open chr ( quote 40 ) )
    ( point chr ( quote 46 ) )
    ( close chr ( quote 41 ) ) )
  ( pretty
    let
    ( letrec
      ( lambda ( s ) ( p s ( quote 0 ) ) )
      ( p
        lambda
```

```
      ( s x )
      ( if
        ( or
          ( atom s )
          ( leq ( quote 0 ) ( g s ( sub linelength x ) ) ) )
        s
        ( ( if ( n s ) q' p' )
          s
          ( add x ( quote 2 ) ) ) ) )
  ( g
    lambda
    ( s x )
    ( if
      ( leq ( quote 0 ) x )
      ( if
        ( atom s )
        ( sub x ( atomsize s ) )
        ( g' s ( sub x listsize ) ) )
      noroom ) )
  ( g'
    lambda
    ( s x )
    ( if
      ( leq ( quote 0 ) x )
      ( if
        ( eq s ( quote NIL ) )
        x
        ( if
          ( atom s )
          ( sub x ( add dotsize ( atomsize s ) ) )
          ( g' ( tail s ) ( g ( head s ) x ) ) ) )
      noroom ) )
  ( p'
    lambda
    ( s x )
    ( cons
      ( p ( head s ) x )
      ( if
        ( atom ( tail s ) )
        ( tail s )
        ( p'' ( tail s ) x ) ) ) )
  ( p'' lambda ( s x ) ( i x ( p' s x ) ) )
  ( q'
    lambda
    ( s x )
    ( cons
      ( p ( head s ) x )
      ( if
        ( eq ( tail ( tail s ) ) ( quote NIL ) )
        ( p'' ( tail s ) ( sub x ( quote 2 ) ) )
        ( q'' ( tail s ) x ) ) ) )
  ( q'' lambda ( s x ) ( i x ( q' s x ) ) )
  ( i lambda ( x s ) ( cons newline ( i' x s ) ) )
  ( i'
    lambda
```

```
      ( x s )
      ( if
        ( eq x ( quote 0 ) )
        s
        ( cons space ( i' ( sub x ( quote 1 ) ) s ) ) ) )
   ( n
      lambda
      ( s )
      ( and
        ( atom ( head s ) )
        ( unless
          ( atom ( tail s ) )
          ( n' ( head s ) ( tail s ) ) ) ) )
   ( n'
      lambda
      ( k s )
      ( if
        ( eq ( tail s ) ( quote NIL ) )
        ( unless
          ( atom ( head s ) )
          ( eq ( head ( head s ) ) k ) )
        ( unless
          ( atom ( tail s ) )
          ( n' k ( tail s ) ) ) ) ) )
( linelength quote 60 )
( atomsize lambda ( s ) ( quote 4 ) )
( listsize quote 4 )
( dotsize quote 2 )
( noroom quote -1 ) ) )
```

```
( ( quicksort
    lambda
    ( less )
    ( letrec
      sort
      ( sort
        lambda
        ( l )
        ( if
          ( atom l )
          l
          ( let
            ( append
              ( sort beginning )
              ( append middle ( sort ending ) ) )
            ( beginning
              filter
              ( lambda ( x ) ( less x ( head l ) ) )
              ( tail l ) )
            ( middle
              filter
              ( lambda ( x ) ( incomparable x ( head l ) ) )
              l )
            ( ending
              filter
              ( lambda ( x ) ( less ( head l ) x ) )
              ( tail l ) ) ) ) )
      ( incomparable
        lambda
        ( a b )
        ( if
          ( less a b )
          ( quote F )
        ( if ( less b a ) ( quote F ) ( quote T ) ) ) ) ) ) )
```

```
( ( append
    lambda
    ( e1 e2 )
    ( if
      ( atom e1 )
      e2
      ( cons ( head e1 ) ( append ( tail e1 ) e2 ) ) ) )
  ( member
    lambda
    ( e l )
    ( unless
      ( atom l )
      ( or
        ( eq e ( head l ) )
        ( member e ( tail l ) ) ) ) )
  ( equal
    lambda
    ( e1 e2 )
    ( or
      ( eq e1 e2 )
      ( unless
        ( or ( atom e1 ) ( atom e2 ) )
        ( and
          ( equal ( head e1 ) ( head e2 ) )
          ( equal ( tail e1 ) ( tail e2 ) ) ) ) ) )
  ( null lambda ( e ) ( eq e ( quote NIL ) ) )
  ( length
    lambda
    ( l )
    ( if
      ( atom l )
      ( quote 0 )
      ( add ( quote 1 ) ( length ( tail l ) ) ) ) )
  ( first
    lambda
    ( n l )
    ( if
      ( or ( eq n ( quote 0 ) ) ( atom l ) )
      ( quote NIL )
      ( cons
        ( head l )
        ( first ( sub n ( quote 1 ) ) ( tail l ) ) ) ) )
  ( list flexible ( lambda ( l ) l ) )
  ( not lambda ( c ) ( if c ( quote F ) ( quote T ) ) )
  ( or lambda ( c1 c2 ) ( if c1 ( quote T ) c2 ) )
  ( and lambda ( c1 c2 ) ( if c1 c2 ( quote F ) ) )
  ( unless lambda ( c1 c2 ) ( if c1 ( quote F ) c2 ) )
  ( until
    lambda
    ( e l )
    ( if
      ( eq ( head l ) e )
```

```
      ( quote NIL )
      ( cons ( head l ) ( until e ( tail l ) ) ) ) )
( after
   lambda
   ( e l )
   ( if
      ( eq ( head l ) e )
      ( tail l )
      ( after e ( tail l ) ) ) )
( untilend lambda ( l ) ( until ( quote end ) l ) )
( afterend lambda ( l ) ( after ( quote end ) l ) )
( map
   lambda
   ( f l )
   ( if
      ( atom l )
      ( quote NIL )
      ( cons ( f ( head l ) ) ( map f ( tail l ) ) ) ) )
( reduce
   lambda
   ( f l z )
   ( if
      ( atom l )
      z
      ( f ( head l ) ( reduce f ( tail l ) z ) ) ) )
( transpose
   lambda
   ( m )
   ( if
      ( atom m )
      m
      ( letrec
         ( t m )
         ( t
            lambda
            ( m )
            ( let
               ( if
                  ( reduce or ( map atom m ) ( quote F ) )
                  ( quote NIL )
                  ( cons heads ( t tails ) ) )
               ( heads map head m )
               ( tails map tail m ) ) ) ) ) )
( filter
   lambda
   ( p l )
   ( if
      ( atom l )
      ( quote NIL )
   ( if
      ( p ( head l ) )
      ( cons ( head l ) ( filter p ( tail l ) ) )
      ( filter p ( tail l ) ) ) ) )
( close
   lambda
```

```
    ( r l )
    ( let
      ( if ( equal l l' ) l ( close r l' ) )
      ( l' r l ) ) )
( number lambda ( x ) ( eq x ( add x ( quote 0 ) ) ) )
( load_code lambda ( c ) ( c ) )
( apply
  lambda
  ( f l )
  ( apply_code ( make_closure f ) ( make_arglist l ) ) )
( newline chr ( quote 13 ) )
( space chr ( quote 32 ) ) )
```

```
( ( err_undef quote 1 )
  ( err_fewargs quote 2 )
  ( err_manyargs quote 3 )
  ( err_arglist quote 4 )
  ( err_invform quote 5 )
  ( err_invdef quote 6 )
  ( err_deftwice quote 7 )
  ( err_formlist quote 8 )
  ( err_formarg quote 9 )
  ( err_actllist quote 10 ) )
```

```
( ( syntax
    letrec
    ( lambda ( e ) ( check e ( quote NIL ) ( quote NIL ) ) )
    ( check
      lambda
      ( e n p )
      ( letrec
        ( let
          ( if
            ( atom e )
            ( defined e n p )
          ( if
            ( eq keyword ( quote quote ) )
            ( list ( quote 2 ) keyword e p )
          ( if
            ( member keyword monadic_ops )
            ( operation ( quote 2 ) keyword e n p )
          ( if
            ( member keyword diadic_ops )
            ( operation ( quote 3 ) keyword e n p )
          ( if
            ( member keyword triadic_ops )
            ( operation ( quote 4 ) keyword e n p )
          ( if
            ( eq keyword ( quote lambda ) )
            ( checkfun keyword e n p )
          ( if
            ( member keyword ( quote ( let letrec ) ) )
            ( checkdef keyword e n p )
            ( checklist e e n p ) ) ) ) ) ) ) ) )
          ( keyword head e ) )
        ( defined
          lambda
          ( e n p )
          ( if
            ( eq n ( quote NIL ) )
            ( error err_undef e e p )
          ( if
            ( member e ( head n ) )
            ( quote NIL )
            ( defined e ( tail n ) p ) ) ) )
        ( list
          lambda
          ( length keyword e p )
          ( letrec
            ( count length e )
            ( count
              lambda
              ( n i )
              ( if
                ( atom i )
                ( if
```

```
                ( eq l ( quote NIL ) )
                ( if
                  ( eq n ( quote 0 ) )
                  ( quote NIL )
                  ( has err_fewargs ) )
                ( has err_arglist ) )
            ( if
              ( eq n ( quote 0 ) )
              ( has err_manyargs )
              ( count ( sub n ( quote 1 ) ) ( tail l ) ) ) ) ) )
        ( has lambda ( n ) ( error n keyword e p ) ) ) )
    ( operation
      lambda
      ( length keyword e n p )
      ( provided
        ( list length keyword e p )
        ( checklist ( tail e ) e n p ) ) )
    ( checkdef
      lambda
      ( keyword e n p )
      ( letrec
        ( if
          ( atom ( tail e ) )
          invalid
          ( both
            ( check body n' p )
            ( checkdefs
              definitions
              ( quote NIL )
              e
              ( if ( eq keyword ( quote letrec ) ) n' n )
              p ) ) )
        ( invalid error err_invform keyword e p )
        ( body head ( tail e ) )
        ( definitions tail ( tail e ) )
        ( n'
          letrec
          ( cons ( definiends definitions ) n )
          ( definiends
            lambda
            ( d )
            ( if
              ( atom d )
              ( quote NIL )
              ( if
                ( unless
                  ( atom ( head d ) )
                  ( atom ( head ( head d ) ) ) )
                ( cons
                  ( head ( head d ) )
                  ( definiends ( tail d ) ) )
                ( definiends ( tail d ) ) ) ) ) )
        ( checkdefs
          lambda
          ( d l e n p )
```

```
            ( if
              ( atom d )
              ( if
                ( eq d ( quote NIL ) )
                ( quote NIL )
                invalid )
            ( if
              ( unless
                ( atom ( head d ) )
                ( atom ( head ( head d ) ) ) )
              ( both
                ( both
                  ( if
                    ( member ( head ( head d ) ) l )
                    ( error
                      err_deftwice
                      ( head ( head d ) )
                      e
                      p )
                    ( quote NIL ) )
                  ( check
                    ( tail ( head d ) )
                    n
                    ( cons ( head ( head d ) ) p ) ) )
                ( checkdefs
                  ( tail d )
                  ( cons ( head ( head d ) ) l )
                  e
                  n
                  p ) )
              ( both
                ( error err_invdef ( head d ) e p )
                ( checkdefs ( tail d ) l e n p ) ) ) ) ) ) )
( checkfun
  lambda
  ( keyword e n p )
  ( letrec
    ( provided
      ( list ( quote 3 ) keyword e p )
      ( both
        ( formallist formals ( quote NIL ) )
        ( check body ( cons ( clean formals ) n ) p ) ) )
    ( body head ( tail ( tail e ) ) )
    ( formals head ( tail e ) )
    ( formallist
      lambda
      ( f l )
      ( if
        ( atom f )
        ( if
          ( eq f ( quote NIL ) )
          ( quote NIL )
          ( error err_formlist e e p ) )
        ( if
          ( atom ( head f ) )
```

```
                    ( if
                      ( member ( head f ) l )
                      ( both
                        ( error err_deftwice ( head f ) e p )
                        ( formallist ( tail f ) l ) )
                      ( formallist
                        ( tail f )
                        ( cons ( head f ) l ) ) )
                    ( both
                      ( error err_formarg ( head f ) e p )
                      ( formallist ( tail f ) l ) ) ) ) ) )
              ( clean
                lambda
                ( l )
                ( if
                  ( atom l )
                  ( quote NIL )
                  ( if
                    ( atom ( head l ) )
                    ( cons ( head l ) ( clean ( tail l ) ) )
                    ( clean ( tail l ) ) ) ) ) ) )
        ( checklist
          lambda
          ( l e n p )
          ( if
            ( atom l )
            ( if
              ( eq l ( quote NIL ) )
              ( quote NIL )
              ( error err_actlist l e p ) )
            ( both
              ( check ( head l ) n p )
              ( checklist ( tail l ) e n p ) ) ) ) ) )
    ( error
      lambda
      ( n a e p )
      ( cons
        ( cons
          n
        ( cons a ( cons e ( cons p ( quote NIL ) ) ) ) )
        ( quote NIL ) ) )
    ( provided
      lambda
      ( a b )
      ( if ( eq a ( quote NIL ) ) b a ) )
    ( both append ) )
  ( printerrors
    letrec
    ( lambda
      ( e )
      ( append
        ( quote ( Syntax check ) )
        ( if
          ( eq e ( quote NIL ) )
          ( quote ( revealed no errors ) )
```

```
        ( printlist e ) ) ) )
( printlist
  lambda
  ( l )
  ( if
     ( eq l ( quote NIL ) )
     ( quote NIL )
     ( append ( print ( head l ) ) ( printlist ( tail l ) ) ) ) )
( print
  lambda
  ( x )
  ( let
     ( if
        ( eq n err_undef )
        ( send
           ( cons a ( quote ( used but not defined ) ) )
           p )
     ( if
        ( eq n err_fewargs )
        ( send
           ( cons a ( quote ( has too few arguments ) ) )
           ( cons e p ) )
     ( if
        ( eq n err_manyargs )
        ( send
           ( cons a ( quote ( has too many arguments ) ) )
           ( cons e p ) )
     ( if
        ( eq n err_arglist )
        ( send
           ( cons a ( quote ( has an incorrect argument list ) ) )
           ( cons e p ) )
     ( if
        ( eq n err_invform )
        ( send
           ( cons ( quote incorrect ) ( cons a ( quote ( form ) ) ) )
           ( cons e p ) )
     ( if
        ( eq n err_invdef )
        ( send
           ( quote ( incorrect form of definitions ) )
           ( cons ( dump ( quote 2 ) a ) p ) )
     ( if
        ( eq n err_deftwice )
        ( send
           ( cons a ( quote ( defined more than once ) ) )
           ( cons e p ) )
     ( if
        ( eq n err_formlist )
        ( send
           ( quote ( incorrect formal argument list ) )
           ( cons e p ) )
     ( if
        ( eq n err_formarg )
        ( send ( quote ( incorrect formal argument ) ) ( cons e p ) )
```

```
      ( if
        ( eq  n err_actlist )
        ( send
          ( quote ( incorrect actual argument list ) )
          ( cons e p ) )
        ( send
          ( append
            ( quote ( unexpected error number ) )
            ( cons n ( quote NIL ) ) )
          p ) ) ) ) ) ) ) ) ) )
      ( n head x )
      ( a head ( tail x ) )
      ( e dump ( quote 2 ) ( head ( tail ( tail x ) ) ) ) )
      ( p head ( tail ( tail ( tail x ) ) ) ) ) ) )
( send
  lambda
  ( m p )
  ( cons
    newline
    ( append
      m
    ( append
      ( list newline space space space space )
      ( if
        ( eq p ( quote NIL ) )
        ( quote ( in the body of the program ) )
        ( position p ) ) ) ) ) )
( position
  lambda
  ( p )
  ( if
    ( eq  p  ( quote NIL ) )
    ( quote NIL )
    ( cons
      ( quote in )
    ( cons  ( head p ) ( position ( tail p ) ) ) ) ) ) ) )
```

```
( ( 1 lambda ( t ) ( head t ) )
  ( 2 lambda ( t ) ( head ( tail t ) ) )
  ( 3 lambda ( t ) ( head ( tail ( tail t ) ) ) )
  ( 4 lambda ( t ) ( head ( tail ( tail ( tail t ) ) ) ) )
  ( 5
    lambda
    ( t )
    ( head ( tail ( tail ( tail ( tail t ) ) ) ) )
  ( 6
    lambda
    ( t )
    ( head ( tail ( tail ( tail ( tail ( tail t ) ) ) ) ) )
  ( el
    lambda
    ( n t )
    ( if
      ( eq n ( quote 1 ) )
      ( head t )
      ( el ( sub n ( quote 1 ) ) ( tail t ) ) ) ) )
```

Pascal sources for the virtual machine

The reference virtual machine

```
program LispKit(Input, Output, InFile, OutFile);

(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Reference model lazy interactive SECD machine, 3                *)
(*      -- version 3a                                  April 83      *)
(*      -- IMPLODE and EXPLODE instructions, version 3b   May 83     *)
(*                                                                   *)
(*      Machine specific code has been omitted from this text        *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*    (c)  Copyright  P Henderson, G A Jones, S B Jones              *)
(*                    Oxford University Computing Laboratory         *)
(*                    Programming Research Group                     *)
(*                    8-11 Keble Road                                *)
(*                    OXFORD  OX1 3QD                                *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Documentation:                                                  *)
(*                                                                   *)
(*     P Henderson, G A Jones, S B Jones                             *)
(*         The LispKit Manual                                        *)
(*         Oxford University Computing Laboratory                    *)
(*         Programming Research Group technical monograph PRG-32     *)
(*         Oxford, August 1983                                       *)
(*                                                                   *)
(*     P Henderson                                                   *)
(*         Functional Programming: Application and Implementation.   *)
(*         Prentice-Hall International, London, 1980                 *)
(*                                                                   *)
(*-------------------------------------------------------------------*)

(*-------------------- Machine dependent constants ------------------*)

label 99;

const TopCell = [ omitted ] ;          (*    size of heap storage     *)

(*---------------- Machine dependent file management ----------------*)

[ omitted ]

(*--------------------- Character input and output --------------------*)

procedure GetChar(var ch : char); [ omitted ]

procedure PutChar(ch : char); [ omitted ]
```

```
(*------- Machine dependent initialisation and finalisation -------*)

procedure Initialise(Version, SubVersion : char); { omitted }

procedure Terminate; { omitted }

(*--------- The code which follows is in Standard Pascal ----------*)
(*   As far as is possible, it is also machine independent.   The  *)
(*  most obvious machine dependency is that the character code of  *)
(*  the host machine has been assumed to be ISO-7 or similar.      *)
(*----------------------------------------------------------------*)

procedure Machine;

const Version = '3';
      SubVersion = 'b';

type TokenType = (Numeric, Alpha, Delimiter);

var Marked, IsAtom, IsNumb : packed array [1..TopCell] of 0..1;
        {-----------------------------------------------}
        {       Cell type coding:   IsAtom   IsNumb     }
        {       Cons                  0         0       }
        {       Recipe                0         1       }
        {       Number                1         1       }
        {       Symbol                1         0       }
        {-----------------------------------------------}
    Head, Tail : array [1..TopCell] of integer;
        {-----------------------------------------------}
        { Head is also used for value of integer, IVAL  }
        {                       pointer to symbol, SVAL }
        {                       BODY of recipe          }
        { Tail is also used for ENVironment of recipe   }
        {-----------------------------------------------}

    S, E, C, D, W, SymbolTable : integer;
    NILL, T, F, OpenParen, Point, CloseParen : integer;
    FreeCell : integer;
    Halted : boolean;

    InCh : char;
    InTokType : TokenType;

(*------------------ Garbage collection routines ------------------*)

procedure CollectGarbage;

    procedure Mark(n : integer);
    begin if Marked[n] = 0
          then begin Marked[n] := 1;
                     if (IsAtom[n] = 0) or (IsNumb[n] = 0) then
                           begin Mark(Head[n]); Mark(Tail[n]) end
          end
    end {Mark};
```

```
    procedure MarkAccessibleStore;
    begin Mark(NILL); Mark(T); Mark(F);
          Mark(OpenParen); Mark(Point); Mark(CloseParen);
          Mark(S); Mark(E); Mark(C); Mark(D); Mark(W)
    end;

    procedure ScanSymbols(var i : integer);
    begin if i <> NILL then
                if Marked[Head[Head[i]]] = 1 then
                      begin Marked[i] := 1;
                            Marked[Head[i]] := 1;
                            ScanSymbols(Tail[i])
                      end
                else begin i := Tail[i]; ScanSymbols(i) end
    end;

    procedure ConstructFreeList;
    var i : integer;
    begin for i := 1 to TopCell do
            if Marked[i] = 0 then
                begin Tail[i] := FreeCell; FreeCell := i end
            else Marked[i] := 0
    end;

begin MarkAccessibleStore;
      ScanSymbols(SymbolTable);
      FreeCell := 0;
      ConstructFreeList;
      if FreeCell = 0 then
            begin writeln(Output, 'Cell store overflow'); Terminate end
end {CollectGarbage};

(*------------------- Storage allocation routines ------------------*)

function Cell : integer;
begin if FreeCell = 0 then CollectGarbage;
      Cell := FreeCell;
      FreeCell := Tail[FreeCell]
end {Cell};

function Cons : integer;
var i : integer;
begin i := Cell;
      IsAtom[i] := 0; IsNumb[i] := 0; Head[i] := NILL; Tail[i] := NILL;
      Cons := i
end {Cons};

function Recipe : integer;
var i : integer;
begin i := Cell;
      IsAtom[i] := 0; IsNumb[i] := 1; Head[i] := NILL; Tail[i] := NILL;
      Recipe := i
end {Recipe};

function Symb : integer;
```

```
var i : integer;
begin i := Cell;
      IsAtom[i] := 1; IsNumb[i] := 0; Head[i] := NILL; Tail[i] := NILL;
      Symb := i
end {Symb};

function Numb : integer;
var i : integer;
begin i := Cell;
      IsAtom[i] := 1; IsNumb[i] := 1;
      Numb := i
end {Numb};

function IsCons(i : integer) : boolean;
begin IsCons := (IsAtom[i] = 0) and (IsNumb[i] = 0) end;

function IsRecipe(i : integer) : boolean;
begin IsRecipe := (IsAtom[i] = 0) and (IsNumb[i] = 1) end;

function IsNumber(i : integer) : boolean;
begin IsNumber := (IsAtom[i] = 1) and (IsNumb[i] = 1) end;

function IsSymbol(i : integer) : boolean;
begin IsSymbol := (IsAtom[i] = 1) and (IsNumb[i] = 0) end;

function IsNill(i : integer) : boolean;
begin IsNill := IsSymbol(i) and (Head[i] = Head[NILL]) end;

procedure Store(var T : integer);
var Si, Sij, Tj : integer;
    found : boolean;
begin Tj := T;
      if IsAtom[Tj] = 1 then Tj := NILL
      else
        begin while IsAtom[Tail[Tj]] = 0 do Tj := Tail[Tj];
              Tail[Tj] := NILL
        end;
      Si := SymbolTable; found := false;
      while (not found) and (Si <> NILL) do
          begin Sij := Head[Head[Si]]; Tj := T; found := true;
                while found and (Tj <> NILL) and (Sij <> NILL) do
                   begin if Head[Tj] <> Head[Sij] then
                              if Head[Head[Tj]] = Head[Head[Sij]]
                              then Head[Tj] := Head[Sij]
                              else found := false;
                         Tj := Tail[Tj]; Sij := Tail[Sij]
                   end;
                if found then found := Tj = Sij;
                if found then T := Head[Si] else Si := Tail[Si]
          end;
      if not found then
          begin Tj := T;          (* NB: T may be an alias for W *)
                W := Cons;
                Tail[W] := Tj;
                Head[W] := Symb;
```

```
                Head[Head[W]] := Tail[W];
                Tail[W] := SymbolTable;
                SymbolTable := W;
                T := Head[W]
          end
end {Store};

procedure InitListStorage;

        var i : integer;

        function List(ch : char) : integer;
        begin W := Cons;
              Head[W] := Numb; Head[Head[W]] := ord(ch);
              List := W
        end {List};

    procedure OneChar(var reg : integer; ch : char);
    begin reg := List(ch); Store(reg) end {OneChar};

begin FreeCell := 1;
      for i := 1 to TopCell - 1 do
               begin Marked[i] := 0; Tail[i] := i + 1 end;
      Marked[TopCell] := 0;
      Tail[TopCell] := 0;
      NILL := Symb; Head[NILL] := NILL; Tail[NILL] := NILL;
      S := NILL; E := NILL; C := NILL; D := NILL; W := NILL;
      T := NILL; F := NILL;
      OpenParen := NILL; Point := NILL; CloseParen := NILL;
      Head[NILL] := List('N');
      Tail[Head[NILL]] := List('I');
      Tail[Tail[Head[NILL]]] := List('L');
      SymbolTable := Cons;
    { Head[SymbolTable] := NILL; the symbol ...     }
    { Tail[SymbolTable] := NILL; the empty list ... }
      OneChar(T, 'T');
      OneChar(F, 'F');
      OneChar(OpenParen, '(');
      OneChar(Point, '.');
      OneChar(CloseParen, ')')
end {InitListStorage};

procedure Update(x, y : integer);
begin IsAtom[x] := IsAtom[y];
      IsNumb[x] := IsNumb[y];
      Head[x] := Head[y];
      Tail[x] := Tail[y]
end {Update};

(*---------------------- Token input and output ---------------------*)

procedure GetToken(var Token : integer);
var x : char;
    p : integer;
begin while InCh = ' ' do GetChar(InCh);
```

```
        x := InCh;
        GetChar(InCh);
        if (('0' <= x) and (x <= '9'))
          or (    ((x = '-') or (x = '+'))
              and ('0' <= InCh) and (InCh <= '9')) then
            begin InTokType := Numeric;
                  Token := Numb;
                  if (x = '+') or (x = '-')
                      then Head[Token] := 0
                      else Head[Token] := ord(x) - ord('0');
                  while ('0' <= InCh) and (InCh <= '9') do
                      begin Head[Token] := (10 * Head[Token])
                                           + (ord(InCh) - ord('0'));
                            GetChar(InCh)
                      end;
                  if x = '-' then Head[Token] := - Head[Token]
            end
    else
        if (x = '(') or (x = ')') or (x = '.') then
            begin InTokType := Delimiter;
                  if x = '(' then Token := OpenParen
                  else if x = '.' then Token := Point
                  else Token := CloseParen
            end
    else
        begin InTokType := Alpha;
            Token := Cons; p := Token;
            Head[p] := Numb; Head[Head[p]] := ord(x);
            while not ( (InCh = '(') or (InCh = ')')
                        or (InCh = '.') or (InCh = ' ') ) do
                begin Tail[p] := Cons; p := Tail[p];
                      Head[p] := Numb; Head[Head[p]] := ord(InCh);
                      GetChar(InCh)
                end;
            Store(Token)
        end
end {GetToken};

procedure PutSymbol(Symbol : integer);
var p : integer;
begin p := Head[Symbol];
      while p <> NILL do
            begin PutChar(chr(Head[Head[p]])); p := Tail[p] end;
      PutChar(' ')
end {PutSymbol};

procedure PutNumber(Number : integer);

  procedure PutN(n : integer);
  begin if n > 9 then PutN(n div 10);
      PutChar(chr(ord('0') + (n mod 10)))
  end;

begin if Head[Number] < 0 then
          begin PutChar('-'); PutN(-Head[Number]) end
```

```
            else PutN(Head[Number]);
      PutChar(' ')
end {PutNumber};

procedure PutRecipe(E : integer);
begin PutChar('*');
      PutChar('*');
      PutChar('R');
      PutChar('E');
      PutChar('C');
      PutChar('I');
      PutChar('P');
      PutChar('E');
      PutChar('*');
      PutChar('*');
      PutChar(' ')
end {PutRecipe};

(*----------------- S-expression input and output -----------------*)

procedure GetExp(var E : integer);

    procedure GetList(var E : integer);
    begin if E = CloseParen then E := NILL
          else begin W := Cons; Head[W] := E; E := W;
                     if Head[E] = OpenParen then
                         begin GetToken(Head[E]); GetList(Head[E]) end;
                     GetToken(Tail[E]);
                     if Tail[E] = Point then
                          begin GetExp(Tail[E]); GetToken(W) end
                     else GetList(Tail[E])
               end
    end {GetList};

begin GetToken(E);
      if E = OpenParen then begin GetToken(E); GetList(E) end
end {GetExp};

procedure PutExp(E  : integer);
var p : integer;
begin if IsRecipe(E) then PutRecipe(E)
 else if IsSymbol(E) then PutSymbol(E)
 else if IsNumber(E) then PutNumber(E)
 else begin PutSymbol(OpenParen);
            p := E;
            while IsCons(p) do begin PutExp(Head[p]); p := Tail[p] end;
            if not IsNill(p) then begin PutSymbol(Point); PutExp(p) end;
            PutSymbol(CloseParen)
      end
end {PutExp};

procedure LoadBootstrapProgram;
begin InCh := ' ';
      GetExp(S);                    (* NB GetExp corrupts W *)
      E := Tail[S]; C := Head[S];
```

```
      S := NILL; D := NILL; W := NILL
end {LoadBootstrapProgram};

{*------------ Microcode for SECD machine operations -------------*}

procedure LDX;
var Wx, i : integer;
begin Wx := E;
      for i := 1 to Head[Head[Head[Tail[C]]]] do Wx := Tail[Wx];
      Wx := Head[Wx];
      for i := 1 to Head[Tail[Head[Tail[C]]]] do Wx := Tail[Wx];
      Wx := Head[Wx];
      W := Cons; Head[W] := Wx; Tail[W] := S; S := W;
      C := Tail[Tail[C]]
end {LDX};

procedure LDCX;
begin W := Cons; Head[W] := Head[Tail[C]]; Tail[W] := S; S := W;
      C := Tail[Tail[C]]
end {LDCX};

procedure LDFX;
begin W := Cons; Head[W] := Cons;
      Head[Head[W]] := Head[Tail[C]]; Tail[Head[W]] := E;
      Tail[W] := S; S := W;
      C := Tail[Tail[C]]
end {LDFX};

procedure APX;
begin W := Cons; Head[W] := Tail[Tail[S]];
      Tail[W] := Cons; Head[Tail[W]] := E;
      Tail[Tail[W]] := Cons; Head[Tail[Tail[W]]] := Tail[C];
      Tail[Tail[Tail[W]]] := D; D := W;
      W := Cons; Head[W] := Head[Tail[S]]; Tail[W] := Tail[Head[S]];
      E := W;
      C := Head[Head[S]];
      S := NILL
end {APX};

procedure RTNX;
begin W := Cons; Head[W] := Head[S]; Tail[W] := Head[D]; S := W;
      E := Head[Tail[D]];
      C := Head[Tail[Tail[D]]];
      D := Tail[Tail[Tail[D]]]
end {RTNX};

procedure DUMX;
begin W := Cons; Head[W] := NILL; Tail[W] := E; E := W; C := Tail[C]
end {DUMX};

procedure RAPX;
begin W := Cons; Head[W] := Tail[Tail[S]];
      Tail[W] := Cons; Head[Tail[W]] := Tail[E];
      Tail[Tail[W]] := Cons; Head[Tail[Tail[W]]] := Tail[C];
      Tail[Tail[Tail[W]]] := D; D := W;
```

```
      E := Tail[Head[S]]; Head[E] := Head[Tail[S]];
      C := Head[Head[S]];
      S := NILL
end {RAPX};

procedure SELX;
begin W := Cons; Head[W] := Tail[Tail[Tail[C]]]; Tail[W] := D; D := W;
      if Head[Head[S]] = Head[T] then C := Head[Tail[C]]
                                 else C := Head[Tail[Tail[C]]];
      S := Tail[S]
end {SELX};

procedure JOINX; begin C := Head[D]; D := Tail[D] end {JOINX};

procedure CARX;
begin W := Cons; Head[W] := Head[Head[S]]; Tail[W] := Tail[S]; S := W;
      C := Tail[C]
end {CARX};

procedure CDRX;
begin W := Cons; Head[W] := Tail[Head[S]]; Tail[W] := Tail[S]; S := W;
      C := Tail[C]
end {CDRX};

procedure ATOMX;
begin W := Cons;
      if IsAtom(Head[S]) = 1 then Head[W] := T else Head[W] := F;
      Tail[W] := Tail[S]; S := W;
      C := Tail[C]
end {ATOMX};

procedure CONSX;
begin W := Cons; Head[W] := Cons;
      Head[Head[W]] := Head[S]; Tail[Head[W]] := Head[Tail[S]];
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
end {CONSX};

procedure EQX;
begin W := Cons;
      if ( ( IsSymbol(Head[S]) and IsSymbol(Head[Tail[S]]) )
        or ( IsNumber(Head[S]) and IsNumber(Head[Tail[S]]) ) )
       and (Head[Head[S]] = Head[Head[Tail[S]]])
      then Head[W] := T
      else Head[W] := F;
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
end {EQX};

procedure ADDX;
begin W := Cons;
      Head[W] := Numb;
      Head[Head[W]] := Head[Head[Tail[S]]] + Head[Head[S]];
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
```

```
end {ADDX};

procedure SUBX;
begin W := Cons;
      Head[W] := Numb;
      Head[Head[W]] := Head[Head[Tail[S]]] - Head[Head[S]];
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
end {SUBX};

procedure MULX;
begin W := Cons;
      Head[W] := Numb;
      Head[Head[W]] := Head[Head[Tail[S]]] * Head[Head[S]];
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
end {MULX};

procedure DIVX;
begin W := Cons;
      Head[W] := Numb;
      Head[Head[W]] := Head[Head[Tail[S]]] div Head[Head[S]];
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
end {DIVX};

procedure REMX;
begin W := Cons;
      Head[W] := Numb;
      Head[Head[W]] := Head[Head[Tail[S]]] mod Head[Head[S]];
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
end {REMX};

procedure LEQX;
begin W := Cons;
      if Head[Head[Tail[S]]] <= Head[Head[S]] then Head[W] := T
                                              else Head[W] := F;
      Tail[W] := Tail[Tail[S]]; S := W;
      C := Tail[C]
end {LEQX};

procedure STOPX;
begin if IsAtom[Head[S]] = 1 then Halted := true
      else begin W := Cons; Head[W] := Tail[S];
                 Tail[W] := Cons; Head[Tail[W]] := E;
                 Tail[Tail[W]] := Cons; Head[Tail[Tail[W]]] := C;
                 Tail[Tail[Tail[W]]] := D; D := W;
                 C := Head[Head[Head[S]]];
                 W := Cons;
                 Head[W] := Tail[Head[S]];
                 Tail[W] := Tail[Head[Head[S]]];
                 E := W;
                 S := NILL
           end
```

```
end {STOPX};

procedure LDEX;
begin W := Cons; Tail[W] := S; S := W;
      Head[W] := Recipe;
      Head[Head[W]] := Head[Tail[C]]; Tail[Head[W]] := E;
      C := Tail[Tail[C]]
end {LDEX};

procedure UPDX;
begin Update(Head[Head[D]],Head[S]);
      S := Head[D];
      E := Head[Tail[D]];
      C := Head[Tail[Tail[D]]];
      D := Tail[Tail[Tail[D]]]
end {UPDX};

procedure APOX;
begin if IsRecipe(Head[S]) then
          begin W := Cons; Head[W] := S;
                Tail[W] := Cons; Head[Tail[W]] := E;
                Tail[Tail[W]] := Cons; Head[Tail[Tail[W]]] := Tail[C];
                Tail[Tail[Tail[W]]] := D; D := W;
                C := Head[Head[S]];
                E := Tail[Head[S]];
                S := NILL
          end
      else C := Tail[C]
end {APOX};

procedure READX;
begin W := Cons; Tail[W] := S; S := W; GetExp(Head[S]); C := Tail[C]
end {READX};

procedure PRINTX;
begin PutExp(Head[S]); S := Tail[S]; C := Tail[C] end {PRINTX};

procedure IMPLODEX;
begin W := Cons; Head[W] := Head[S]; Tail[W] := Tail[S]; S := W;
      if IsNumber(Head[S]) then
            if Head[Head[S]] = ord(' ') then Head[S] := NILL
                  else begin W := Cons;
                             Head[W] := Head[S];
                             Head[S] := W
                       end;
      Store(Head[S]);
      C := Tail[C]
end {IMPLODEX};

procedure EXPLODEX;
begin W := Cons; Head[W] := Head[Head[S]]; Tail[W] := Tail[S]; S := W;
      C := Tail[C]
end {EXPLODEX};

(*-------------------- Instruction decode loop ---------------------*)
```

```
procedure FetchExecuteLoop;
label 1;
begin Halted := false;
   1: case Head[Head[C]] of
                 1:   LDX;              11:  CDRX;
                 2:  LDCX;              12: ATOMX;
                 3:  LDFX;              13: CONSX;
                 4:   APX;              14:   EQX;
                 5:  RTNX;              15:  ADDX;
                 6:  DUMX;              16:  SUBX;
                 7:  RAPX;              17:  MULX;
                 8:  SELX;              18:  DIVX;
                 9: JOINX;              19:  REMX;
                10:  CARX;              20:  LEQX;

                21: begin STOPX; if Halted then Terminate end;

                22: LDEX;              25: READX;
                23: UPDX;              26: PRINTX;
                24: APOX;              27: IMPLODEX;
                                       28: EXPLODEX
      end;
      goto 1
end {FetchExecuteLoop};

(*------------------ body of procedure Machine ---------------------*)

begin Initialise(Version, SubVersion);
      InitListStorage;
      LoadBootstrapProgram;
      FetchExecuteLoop
end {Machine};

(*------------------ body of program LispKit ----------------------*)

begin Machine; 99: end {LispKit}.
```

```
program LispKit(Input, Output, InFile, OutFile);

(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Reference model lazy interactive SECD machine, 3                *)
(*      -- version 3a                                  April 83      *)
(*      -- IMPLODE and EXPLODE instructions, version 3b   May 83     *)
(*                                                                   *)
(*   Modifications specific to UCSD pascal  gaj        April 83      *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*    (c)  Copyright  P Henderson, G A Jones, S B Jones              *)
(*                    Oxford University Computing Laboratory         *)
(*                    Programming Research Group                     *)
(*                    8-11 Keble Road                                *)
(*                    OXFORD  OX1 3QD                                *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Documentation:                                                  *)
(*                                                                   *)
(*     P Henderson, G A Jones, S B Jones                             *)
(*         The LispKit Manual                                        *)
(*         Oxford University Computing Laboratory                    *)
(*         Programming Research Group technical monograph PRG-32     *)
(*         Oxford, August 1983                                       *)
(*                                                                   *)
(*     P Henderson                                                   *)
(*         Functional Programming: Application and Implementation.   *)
(*         Prentice-Hall International, London, 1980                 *)
(*                                                                   *)
(*-------------------------------------------------------------------*)

(*------------------ Machine dependent constants -------------------*)

label 99;

const TopCell = 8000;                (*    size of heap storage     *)

(*--------------- Machine dependent file management ----------------*)

var InOpen : boolean;
    InFile : interactive;
    OutFile : text;

procedure OpenInFile;
var s : string;
begin writeln(Output);
      write(Output, 'Take input from where? ');
      readln(Input, s);
      if s = '' then s := 'CONSOLE:';
```

```
      {$I-} reset(InFile, s) {$I+};
      InOpen := IOResult = 0;
      if not InOpen then write(Output, 'Cannot find ', s)
end {OpenInFile};

procedure CloseInFile; begin close(InFile, NORMAL); InOpen := false end;

procedure ChangeOutFile;
var s : string;
    ok : boolean;
begin close(OutFile, LOCK);
      repeat writeln(Output);
             write(Output, 'Send output to where? ');
             readln(Input, s);
             if s = '' then s := 'CONSOLE:';
             {$I-} rewrite(OutFile, s) {$I+};
             ok := IOResult = 0;
             if not ok then write(Output, 'Cannot write to ', s)
      until ok
end {ChangeOutFile};

(*------------------- Character input and output --------------------*)

procedure GetChar(var ch : char);
const EM = 25;
begin while not InOpen do OpenInFile;
      if eof(InFile) then begin CloseInFile; ch := ' ' end
      else
        if eoln(InFile) then begin readln(InFile); ch := ' ' end
      else
        begin read(InFile, ch);
              if ch = chr(EM) then
                   begin readln(InFile); ChangeOutFile; ch := ' ' end
        end
end {GetChar};

procedure PutChar(ch : char);
const CR = 13;
begin if ch = chr(CR) then writeln(OutFile) else write(OutFile, ch)
end {PutChar};

(*------- Machine dependent initialisation and finalisation --------*)

procedure Initialise(Version, SubVersion : char);
begin writeln(Output, 'Sage Pascal SECD machine ', Version, SubVersion);
      {$I-} reset(InFile, '*SECD.BOOT') {$I+};
      InOpen := IOResult = 0;
      if not InOpen then writeln(Output, 'No file *SECD.BOOT');
      rewrite(OutFile, 'CONSOLE:')
end {Initialise};

procedure Terminate; begin close(OutFile); exit(PROGRAM) end {Terminate};

(*------------------ Machine independent code ----------------------*)
```

```
procedure Machine; { omitted }

begin Machine; 99: end {LispKit}.
```

```
[INHERIT('SYSSLIBRARY:STARLET')] { FAB-related definitions }

program LispKit(Input, Output, InFile, OutFile);

(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Reference model lazy interactive SECD machine, 3                *)
(*      -- version 3a                                  April 83      *)
(*      -- IMPLODE and EXPLODE instructions, version 3b   May 83     *)
(*                                                                   *)
(*   Modifications specific to VAX VMS Pascal gaj      April 83      *)
(*   Break long lines in file output gaj              August 83      *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*    (c)  Copyright  P Henderson, G A Jones, S B Jones              *)
(*                    Oxford University Computing Laboratory         *)
(*                    Programming Research Group                     *)
(*                    8-11 Keble Road                                *)
(*                    OXFORD  OX1 3QD                                *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Documentation:                                                  *)
(*                                                                   *)
(*    P Henderson, G A Jones, S B Jones                              *)
(*        The LispKit Manual                                         *)
(*        Oxford University Computing Laboratory                     *)
(*        Programming Research Group technical monograph PRG-32      *)
(*        Oxford, August 1983                                        *)
(*                                                                   *)
(*    P Henderson                                                    *)
(*        Functional Programming: Application and Implementation.    *)
(*        Prentice-Hall International, London, 1980                  *)
(*                                                                   *)
(*-------------------------------------------------------------------*)

(*------------------ Machine dependent constants -------------------*)

label 99;

const TopCell = 40000;      (*    size of heap storage     *)
      FileRecordLimit = 255;
      OutFileWidth = 200;
      OutTermWidth = 80;

(*--------------- Machine dependent file management ----------------*)

var InOpen : boolean;
    InFile : text;
    NewInput, InFromTerminal, OutToTerminal : boolean;
    OutFile : text;
```

```
    OutFileColumn : integer;
    OutTermColumn : integer;
    NullName : packed array[1..255] of char;

function IsTerminal(VAR f : text) : boolean;
  type phyle = [UNSAFE] text;
       pointer = ↑FAB$TYPE;
  var p : pointer;
  function PAS$FAB(VAR f : phyle) : pointer; EXTERN;
begin p := PAS$FAB(f);
      IsTerminal := p↑.FAB$L_DEV = 201588743
end {IsTerminal};

procedure OpenInFile;
var s : packed array[1..255] of char;
begin writeln(Output);
      write(Output, 'Take input from where? ');
      readln(Input, s);
      writeln(Output);
      if s = NullName then
        open(File_Variable := InFile,
             File_Name := 'SYS$INPUT',
             History := Old)
      else
        open(File_Variable := InFile,
             File_Name := s,
             History := Old,
             Error := CONTINUE);
      InOpen := Status(InFile) <= 0;
      if InOpen then
         begin reset(InFile); InFromTerminal := IsTerminal(InFile) end
      else write(Output, 'Cannot read from that file')
end {OpenInFile};

procedure CloseInFile; begin close(InFile); InOpen := false end;

procedure ChangeOutFile;
var s : packed array[1..255] of char;
    ok : boolean;
begin close(OutFile);
      repeat writeln(Output);
             write(Output, 'Send output to where? ');
             readln(Input, s);
             if s = NullName then
               open(File_Variable := OutFile,
                    File_Name := 'SYS$OUTPUT',
                    History := New,
                    Record_Length := FileRecordLimit)
             else
               open(File_Variable := OutFile,
                    File_Name := s,
                    History := New,
                    Record_Length := FileRecordLimit,
                    Error := CONTINUE);
             ok := Status(OutFile) <= 0;
```

```
            if ok then rewrite(OutFile)
            else write(Output, 'Cannot write to that file')
      until ok;
      OutToTerminal := IsTerminal(OutFile);
      OutTermColumn := 0;
      OutFileColumn := 0
end {ChangeOutFile};

(*------------------- Character input and output -------------------*)

procedure GetChar(VAR ch : char);
const EM = 8;
begin while not InOpen do begin OpenInFile; NewInput := true end;
      if eof(InFile) then begin CloseInFile; ch := ' ' end
      else
        if eoln(InFile) then
            begin readln(InFile); NewInput := true; ch := ' ' end
      else
        begin if NewInput then
            begin if InFromTerminal then OutTermColumn := 0;
                  NewInput := false
            end;
            read(InFile, ch);
            if ch = chr(EM) then
                begin readln(InFile); ChangeOutFile; ch := ' ' end
        end;
end {GetChar};

procedure PutChar(ch : char);
const CR = 13;
begin if ch = ' ' then
        if OutToTerminal then
          begin if OutTermColumn >= OutTermWidth then ch := chr(CR) end
        else
          begin if OutFileColumn >= OutFileWidth then ch := chr(CR) end;
      if ch = chr(CR) then
        begin writeln(OutFile);
            if OutToTerminal then
                OutTermColumn := 0
            else OutFileColumn := 0
        end
      else
        begin write(OutFile, ch);
            if OutToTerminal then
                OutTermColumn := OutTermColumn + 1
            else OutFileColumn := OutFileColumn + 1
        end
end {PutChar};

(*------- Machine dependent initialisation and finalisation -------*)

procedure Initialise(Version, SubVersion : char);
var i : 1..255;
begin writeln(Output, 'VAX Pascal SECD machine ', Version, SubVersion);
      for i := 1 to 255 do NullName[i] := ' ';
```

```
      open(File_Variable := InFile,
           File_Name := 'LISPKIT$SECDBOOT',
           History := Old,
           Error := CONTINUE);
      InOpen := status(InFile) <= 0;
      if InOpen then
         begin reset(InFile); InFromTerminal := IsTerminal(InFile) end
      else writeln(Output, 'No file LispKit$SECDboot');
      NewInput := true;
      open(File_Variable := OutFile,
           File_Name := 'SYS$OUTPUT',
           History := New,
           Record_Length := FileRecordLimit);
      rewrite(OutFile);
      OutToTerminal := IsTerminal(OutFile);
      OutTermColumn := 0;
      OutFileColumn := 0
end {Initialise};

procedure Terminate;
begin writeln(OutFile); close(OutFile); goto 99 end {Terminate};

(*------------------ Machine independent code ----------------------*)

procedure Machine; { omitted }

begin Machine; 99: end {LispKit}.
```

```
program LispKit(Input, Output, InFile, OutFile);

(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Reference model lazy interactive SECD machine, 3                *)
(*      -- version 3a                                  April 83      *)
(*      -- IMPLODE and EXPLODE instructions, version 3b   May 83     *)
(*                                                                   *)
(*   Modifications specific to ICL Perq POS Pascal gaj  April 83     *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*    (c)  Copyright  P Henderson, G A Jones, S B Jones              *)
(*                    Oxford University Computing Laboratory         *)
(*                    Programming Research Group                     *)
(*                    8-11 Keble Road                                *)
(*                    OXFORD  OX1 3QD                                *)
(*                                                                   *)
(*-------------------------------------------------------------------*)
(*                                                                   *)
(*   Documentation:                                                  *)
(*                                                                   *)
(*     P Henderson, G A Jones, S B Jones                             *)
(*         The LispKit Manual                                        *)
(*         Oxford University Computing Laboratory                    *)
(*         Programming Research Group technical monograph PRG-32     *)
(*         Oxford, August 1983                                       *)
(*                                                                   *)
(*     P Henderson                                                   *)
(*         Functional Programming: Application and Implementation.   *)
(*         Prentice-Hall International, London, 1980                 *)
(*                                                                   *)
(*-------------------------------------------------------------------*)

(*------------------ Machine dependent constants -------------------*)

imports Perq_string from PERQ_STRING; (*  for the type string   *)
imports stream from STREAM;           (* for I/O error trapping *)

label 99;

const TopCell = 10000;      (*     size of heap storage     *)

(*---------------- Machine dependent file management ----------------*)

var InOpen : boolean;
    InFile : text;
    OutFile : text;

procedure TryReset(s : string);

  handler ResetError(f : pathname);
```

```
  begin InOpen := false; exit(TryReset) end {ResetError};

begin reset(InFile, s);
      InOpen := true
end {TryReset};

function TryRewrite(s : string) : boolean;

  handler RewriteError(f : pathname);
  begin TryRewrite := false; exit(TryRewrite) end {RewriteError};

begin rewrite(OutFile, s);
      TryRewrite := true
end {TryRewrite};

procedure OpenInFile;
var s : string;
begin writeln(Output);
      write(Output, 'Take input from where? ');
      readln(Input, s);
      if s = '' then s := 'CONSOLE:';
      TryReset(s);
      if not InOpen then write(Output, 'Cannot find ', s)
end {OpenInFile};

procedure CloseInFile; begin close(InFile); InOpen := false end;

procedure ChangeOutFile;
var s : string;
    ok : boolean;
begin close(OutFile);
      repeat writeln(Output);
             write(Output, 'Send output to where? ');
             readln(Input, s);
             if s = '' then s := 'CONSOLE:';
             ok := TryRewrite(s);
             if not ok then write(Output, 'Cannot write to ', s)
      until ok
end {ChangeOutFile};

(*------------------- Character input and output -----------------*)

procedure GetChar(var ch : char);
const EM = 25;
begin while not InOpen do OpenInFile;
      if eof(InFile) then begin CloseInFile; ch := ' ' end
      else
        if eoln(InFile) then begin readln(InFile); ch := ' ' end
      else
        begin read(InFile, ch);
              if ch = chr(EM) then
                    begin readln(InFile); ChangeOutFile; ch := ' ' end
        end
end {GetChar};
```

```
procedure PutChar(ch : char);
const CR = 13;
begin if ch = chr(CR) then writeln(OutFile) else write(OutFile, ch)
end {PutChar};

(*------- Machine dependent initialisation and finalisation -------*)

procedure Initialise(Version, SubVersion : char);
begin writeln(Output, 'Perq Pascal SECD machine ', Version, SubVersion);
      TryReset('SECD.BOOT');
      if not InOpen then writeln(Output, 'No file SECD.BOOT');
      rewrite(OutFile, 'CONSOLE:')
end {Initialise};

procedure Terminate; begin close(OutFile); exit(LispKit) end {Terminate};

(*------------------ Machine independent code ---------------------*)

procedure Machine; { omitted }

begin Machine; 99: end {LispKit}.
```

Oxford University Computing Laboratory
Programming Research Group Technical Monographs
Autumn 1983

This manual is one of a series of technical monographs on topics in the field of computation. Copies may be obtained from:

Programming Research Group, *(Technical Monographs)*
8-11, Keble Road, Oxford. OX1 3QD. England.

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-5 Dana Scott
Data Types as Lattices

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages

PRG-7 Dana Scott
Continuous Lattices

PRG-8 Joseph Stoy and Christopher Strachey
OS6 - an Experimental Operating System for a Small Computer

PRG-9 Christopher Strachey and Joseph Stoy
The Text of OSPub

PRG-10 Christopher Strachey
The Varieties of Programming Language

PRG-11 Christopher Strachey and Christopher P. Wadsworth
Continuations: A Mathematical Semantics for Handling Full Jumps

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

PRG-13 Robert Milne
The Formal Semantics of Computer Languages and their Implementation

PRG-14 Shan S. Kuo, Michael Linck and Sohrab Saadat
A Guide to Communicating Sequential Processes

PRG-15 Joseph Stoy
The Congruence of Two Programming Language Definitions

PRG-16 C. A. R. Hoare, S. D. Brookes and A. W. Roscoe
A Theory of Communicating Sequential Processes

PRG-17 Andrew P. Black
Report on the Programming Notation 3R

PRG-18 Elizabeth Fielding
The Specification of Abstract Mappings and their Implementation as B^+-trees

PRG-19 Dana Scott
Lectures on a Mathematical Theory of Computation

PRG-20 Zhou Chao Chen and C. A. R. Hoare
Partial Correctness of Communicating Processes and Protocols

PRG-21 Bernard Sufrin
Formal Specification of a Display Editor

PRG-22 C. A. R. Hoare
A Model of Communicating Sequential Processes

PRG-23 C. A. R. Hoare
A Calculus of Total Correctness for Communicating Sequential Processes

PRG-24 Bernard Sufrin
Reading Formal Specifications

PRG-25 C. B. Jones
Development Methods for Computer Programs including a Notion of Interference

PRG-26 Zhou Chao Chen
The Consistency of the Calculus of Total Correctness for Communicating Processes

PRG-27 C. A. R. Hoare
Programming is an Engineering Profession

PRG-28 John Hughes
Graph Reduction with Super-Combinators

PRG-29 C. A. R. Hoare
Specifications, Programs and Implementations

PRG-30 Alejandro Teruel
Case Studies in Specification: Four Games

PRG-31 Ian D. Cottam
The Rigorous Development of a System Version Control Database

PRG-32 Peter Henderson, Geraint A. Jones and Simon B. Jones
The LispKit Manual

PRG-33 C. A. R. Hoare
Notes on Communicating Sequential Processes

PRG-34 Simon B. Jones
Abstract Machine Support for Purely Functional Operating Systems

PRG-35 S. D. Brookes
A Non-deterministic Model for Communicating Sequential Processes

PRG-36 T. Clement
The Formal Specification of a Conference Organizing System