

File created: 15-Apr-2024 21:22:06 {DSK}<home>larry>il>medley>sources>CMLEVAL.;7

edit by: lmm

changes to: (VARS CMLEVALCOMS)

previous date: 15-Apr-2024 20:14:11 {DSK}<home>larry>il>medley>sources>CMLEVAL.;5

Read Table: XCL

Package: INTERLISP

Format: XCCS

(RPAQQ CMLEVALCOMS

(

;;; Common Lisp interpreter

(COMS ;; These really don't belong here

(FUNCTIONS CL:EQUAL CL:EQUALP)

;; For the byte compiler: Optimize by constant fold and coerce to EQ where possible

(PROP BYTEMACRO CL:EQUAL CL:EQUALP)

(PROP DOPVAL CL:EQUAL))

(COMS (FUNCTIONS \\REMOVE-DECLS)

(FUNCTIONS CL:SPECIAL-FORM-P))

(COMS (SPECIAL-FORMS INTERLISP)

(PROP DMACRO INTERLISP COMMON-LISP)

(FNS COMMON-LISP))

(COMS (ADDVARS (LAMBDA SPLST CL:LAMBDA))

(FNS \\TRANSLATE-CL\\:LAMBDA)

(VARIABLES *CHECK-ARGUMENT-COUNTS* *SPECIAL-BINDING-MARK*))

(VARIABLES CL:LAMBDA-LIST-KEYWORDS CL:CALL-ARGUMENTS-LIMIT CL:LAMBDA-PARAMETERS-LIMIT)

(STRUCTURES CLOSURE ENVIRONMENT)

(FUNCTIONS \\MAKE-CHILD-ENVIRONMENT)

(COMS (FNS CL:EVAL \\EVAL-INVOKED-LAMBDA \\INTERPRET-ARGUMENTS \\INTERPRETER-LAMBDA CHECK-BINDABLE
CHECK-KEYWORDS)

(FUNCTIONS ARG-REF)

(PROP DMACRO .COMPILER-SPREAD-ARGUMENTS.))

(FNS DECLARED-SPECIAL)

(COMS

(FNS CL:FUNCALL CL:APPLY)

; FUNCALL and APPLY, not quite same as Interlisp

(PROP DMACRO CL:APPLY CL:FUNCALL))

(COMS

; COMPILER-LET needs to work differently compiled and
; interpreted

(FNS CL:COMPILER-LET COMP.COMPILER-LET)

(PROP DMACRO CL:COMPILER-LET)

(SPECIAL-FORMS CL:COMPILER-LET))

(COMS

; Lexical function- and macro-binding forms: FLET, LABELS, and
; MACROLET.

(SPECIAL-FORMS CL:MACROLET CL:FLET CL:LABELS))

(SPECIAL-FORMS QUOTE)

(COMS (SPECIAL-FORMS THE)

(PROP DMACRO THE))

(COMS (PROP DMACRO CL:EVAL-WHEN)

(FNS CL:EVAL-WHEN)

(SPECIAL-FORMS CL:EVAL-WHEN))

(COMS (SPECIAL-FORMS DECLARE)

(FUNCTIONS CL:LOCALLY))

(COMS

; Interlisp version on LLINTERP

(SPECIAL-FORMS PROGN)

(FNS \\EVAL-PROGN))

(COMS

; Confused because currently Interlisp special form, fixing
; MACRO-FUNCTION is complex
; The Interlisp function is on LLINTERP

(SPECIAL-FORMS PROG1)

(FUNCTIONS PROG1))

(COMS (SPECIAL-FORMS LET* LET)

(PROP MACRO LET LET*))

(FNS \\LET*-RECURSION |\\LETtran|))

(COMS (SPECIAL-FORMS COND)

(FUNCTIONS COND))

(COMS (FNS CL:IF)

(SPECIAL-FORMS CL:IF)

(PROP DMACRO CL:IF))

(COMS

; Interlisp NLAMBDA definitions on LLINTERP
; both special form and macro

(FUNCTIONS AND OR)

(SPECIAL-FORMS AND OR))

(COMS

; BLOCK and RETURN go together

(FNS CL:BLOCK)

(PROP DMACRO CL:BLOCK)

(SPECIAL-FORMS CL:BLOCK)

(FUNCTIONS RETURN)

(FNS CL:RETURN-FROM)

(SPECIAL-FORMS CL:RETURN-FROM))

```

(COMS                                     ; IL and CL versions of FUNCTION.
  (FNS CL:FUNCTION)
  (PROP DMACRO CL:FUNCTION)
  (SPECIAL-FORMS CL:FUNCTION FUNCTION)
  (FUNCTIONS CL:FUNCTIONP CL:COMPILED-FUNCTION-P))
(SPECIAL-FORMS CL:MULTIPLE-VALUE-CALL CL:MULTIPLE-VALUE-PROG1)
(FNS COMP.CL-EVAL)
(FUNCTIONS CL:EVALHOOK CL:APPLYHOOK)
(VARIABLES *EVALHOOK* *APPLYHOOK* CL:*SKIP-EVALHOOK* CL:*SKIP-APPLYHOOK*)
(COMS                                     ; CONSTANTS mechanism
  (FNS CL:CONSTANTP)
  (SETFS CL:CONSTANTP)
  (FUNCTIONS XCL::SET-CONSTANTP))

(COMS                                     ; Interlisp SETQ for Common Lisp and vice versa
  (SPECIAL-FORMS CL:SETQ SETQ)
  (PROP DMACRO CL:SETQ)
  ;; An nlambda definition for cl:setq so cmlddefer may use cl:setq will run in the init
  (FNS CL:SETQ)
  (FUNCTIONS SETQ)
  (FNS SET-SYMBOL)
  (FUNCTIONS CL:PSETQ)
  (FUNCTIONS SETQQ))
(COMS (SPECIAL-FORMS CL:CATCH CL:THROW CL:UNWIND-PROTECT)
  (FNS CL:THROW CL:CATCH CL:UNWIND-PROTECT))
(COMS (FUNCTIONS PROG PROG*)
  (SPECIAL-FORMS GO CL:TAGBODY)
  (FNS CL:TAGBODY))

(COMS                                     ; for macro caching
  (FNS CACHEMACRO)
  (VARIABLES *MACROEXPAND-HOOK*)
  (VARS (*IN-COMPILER-LET* NIL)))

(COMS ;; PROCLAIM and friends.
  ;; Needs to come first because DEFVARs put it out. With package code in the init, also need this here rather than CMLEVAL
  (FUNCTIONS CL:PROCLAIM)
  ;; used by the codewalker, too
  (MACROS VARIABLE-GLOBALLY-SPECIAL-P VARIABLE-GLOBAL-P)
  (FUNCTIONS XCL::DECL-SPECIFIER-P XCL::SET-DECL-SPECIFIER-P)
  (FUNCTIONS XCL::GLOBALLY-NOTINLINE-P XCL::SET-GLOBALLY-NOTINLINE-P)
  (SETFS XCL::DECL-SPECIFIER-P XCL::GLOBALLY-NOTINLINE-P)
  (PROP PROPTYPE GLOBALLY-SPECIAL GLOBALVAR SI::DECLARATION-SPECIFIER SI::GLOBALLY-NOTINLINE
    SPECIAL-FORM))
(PROP (FILETYPE MAKEFILE-ENVIRONMENT)
  CMLEVAL)
(DECLARE\ : EVAL@COMPILE DONTCOPY (OPTIMIZERS CL-EVAL-FN3-CALL))
(DECLARE\ : DONTVAL@LOAD DOEVAL@COMPILE DONTCOPY (LOCALVARS . T))
(DECLARE\ : DONTVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILEVARs
  (ADDVARS (NLAMA CL:TAGBODY CL:UNWIND-PROTECT CL:CATCH CL:SETQ CL:BLOCK CL:EVAL-WHEN
    CL:COMPILER-LET COMMON-LISP)
    (NLAML CL:THROW CL:FUNCTION CL:RETURN-FROM CL:IF)
    (LAMA CL:APPLY CL:FUNCALL))))

```

```
;; Common Lisp interpreter
```

```
;; These really don't belong here
```

```

(CL:DEFUN CL:EQUAL (CL::X CL::Y)
  (CL:TYPECASE CL::X
    (CL:SYMBOL (EQ CL::X CL::Y))
    (CL:NUMBER (EQL CL::X CL::Y))
    (CONS (AND (CL:CONSP CL::Y)
      (CL:EQUAL (CAR CL::X)
        (CAR CL::Y))
      (CL:EQUAL (CDR CL::X)
        (CDR CL::Y))))
    (STRING (AND (CL:STRINGP CL::Y)
      (CL:STRING= CL::X CL::Y)))
    (CL:BIT-VECTOR (AND (CL:BIT-VECTOR-P CL::Y)
      (LET ((CL::SX (CL:LENGTH CL::X))
        (AND (EQL CL::SX (CL:LENGTH CL::Y))
          (CL:DOTIMES (CL::I CL::SX T)
            (CL:IF (NOT (EQ (BIT CL::X CL::I)
              (BIT CL::Y CL::I)))
              (RETURN NIL)))))))
    (PATHNAME (AND (CL:PATHNAMEP CL::Y)
      (%PATHNAME-EQUAL CL::X CL::Y)))
    (T (EQ CL::X CL::Y))))

```

```

(CL:DEFUN CL:EQUALP (CL::X CL::Y)
  (CL:TYPECASE CL::X
    (CL:SYMBOL (EQ CL::X CL::Y))
    (CL:NUMBER (AND (CL:NUMBERP CL::Y)
      (= CL::X CL::Y)))

```

```

(CONS (AND (CL:CONSP CL::Y)
  (CL:EQUALP (CAR CL::X)
    (CAR CL::Y))
  (CL:EQUALP (CDR CL::X)
    (CDR CL::Y))))
(CL:CHARACTER (AND (CL:CHARACTERP CL::Y)
  (CL:CHAR-EQUAL CL::X CL::Y)))
(STRING (AND (CL:STRINGP CL::Y)
  (STRING-EQUAL CL::X CL::Y)))
(PATHNAME (AND (CL:PATHNAMEP CL::Y)
  (%PATHNAME-EQUAL CL::X CL::Y)))
(CL:VECTOR (AND (CL:VECTORP CL::Y)
  (LET ((CL::SX (CL:LENGTH CL::X)))
    (AND (EQL CL::SX (CL:LENGTH CL::Y))
      (CL:DOTIMES (CL::I CL::SX T)
        (CL:IF (NOT (CL:EQUALP (CL:AREF CL::X CL::I)
          (CL:AREF CL::Y CL::I)))
          (RETURN NIL)))))))
(CL:ARRAY (AND (CL:ARRAYP CL::Y)
  (CL:EQUAL (CL:ARRAY-DIMENSIONS CL::X)
    (CL:ARRAY-DIMENSIONS CL::Y))
  (LET ((CL::FX (%FLATTEN-ARRAY CL::X))
    (CL::FY (%FLATTEN-ARRAY CL::Y)))
    (CL:DOTIMES (CL::I (CL:ARRAY-TOTAL-SIZE CL::X)
      T)
      (CL:IF (NOT (CL:EQUALP (CL:AREF CL::FX CL::I)
        (CL:AREF CL::FY CL::I)))
        (RETURN NIL))))))
(T ;; so that datatypes will be properly compared
  (OR (EQ CL::X CL::Y)
    (LET ((CL::TYPENAME (TYPENAME CL::X))
      (AND (EQ CL::TYPENAME (TYPENAME CL::Y))
        (LET ((CL::DESCRIPTORS (GETDESCRIPTORS CL::TYPENAME)))
          (CL:IF CL::DESCRIPTORS
            (FOR CL::FIELD IN CL::DESCRIPTORS ALWAYS (CL:EQUALP (FETCHFIELD CL::FIELD
              CL::X)
                (FETCHFIELD CL::FIELD CL::Y)
              ))))))))

```

;; For the byte compiler: Optimize by constant fold and coerce to EQ where possible

```
(PUTPROPS CL:EQUAL BYTEMACRO COMP.EQ)
```

```
(PUTPROPS CL:EQUALP BYTEMACRO COMP.EQ)
```

```
(PUTPROPS CL:EQUAL DOPVAL (2 CMLEQUAL))
```

```
(CL:DEFUN \REMOVE-DECLS (CL::BODY CL::ENVIRONMENT)
```

;;; This is like parse-body, except that it returns the body and a list of specials declared in this frame. It side-effects the environment to mark the
 ;;; specials.

```

(PROG ((CL::SPECIALS NIL)
  CL::FORM)
  CL::NEXT-FORM
  (CL:IF (NULL CL::BODY)
    (GO CL::DONE))
  (CL:SETQ CL::FORM (CAR CL::BODY))
  CL::RETRY-FORM
  (COND
    ((OR (CL:ATOM CL::FORM)
      (NOT (CL:SYMBOLP (CAR CL::FORM))))
      (GO CL::DONE))
    ((EQ (CAR CL::FORM)
      'DECLARE)
      (CL:MAPC #'(CL:LAMBDA (CL:DECLARATION)
        (CL:WHEN (CL:CONSP CL:DECLARATION)
          (CL:WHEN (OR (EQ (CAR CL:DECLARATION)
            'CL:SPECIAL)
              (EQ (CAR CL:DECLARATION)
                'SPECVARS))
            (CL:IF (EQ (CDR CL:DECLARATION)
              T)
                ;; (specvars . t) refers to all variables inside this scope, not just those bound in this frame. So
                ;; handling (specvars . t) by declaring the variables in this frame special would not be correct. Hence
                ;; print a warning and continue.
                (CL:WARN "(IL:SPECVARS . T) has no effect in the CL evaluator.")
                (CL:MAPC #'(CL:LAMBDA (CL::NAME)
                  (CL:PUSH CL::NAME CL::SPECIALS))
                  (CDR CL:DECLARATION))))))
        (CDR CL::FORM))
      (CL:POP CL::BODY)
      (GO CL::NEXT-FORM))
    ((CL:SPECIAL-FORM-P (CAR CL::FORM))

```

```

      (GO CL::DONE))
      (T (LET ((CL::NEW-FORM (CL:MACROEXPAND-1 CL::FORM CL::ENVIRONMENT)))
          (COND
            ((AND (NOT (EQ CL::NEW-FORM CL::FORM))
                  (CL:CONSP CL::NEW-FORM))
              (CL:SETQ CL::FORM CL::NEW-FORM)
              (GO CL::RETRY-FORM))
            (T (GO CL::DONE))))))
CL::DONE
  (RETURN (CL:IF CL::SPECIALS
    (PROGN (FOR CL::VAR IN CL::SPECIALS DO (CL:SETF (ENVIRONMENT-VARS CL::ENVIRONMENT)
      (LIST* CL::VAR *SPECIAL-BINDING-MARK*
        (ENVIRONMENT-VARS CL::ENVIRONMENT))
      (CL:VALUES CL::BODY CL::SPECIALS))
    CL::BODY))))

(CL:DEFUN CL:SPECIAL-FORM-P (CL:X)
  (GET CL:X 'SPECIAL-FORM))

(DEFINE-SPECIAL-FORM INTERLISP PROGN)

(PUTPROPS INTERLISP DMACRO ((X . Y)
  (PROGN X . Y)))

(PUTPROPS COMMON-LISP DMACRO ((X)
  X))

(DEFINEQ
  (COMMON-LISP
    (NLAMBDA COMMON-LISP-FORMS
      (\\EVAL-PROGN COMMON-LISP-FORMS NIL)))
    ; Edited 12-Feb-87 20:24 by Pavel
  )

(ADDTOVAR LAMBDA SPLST CL:LAMBDA)

(DEFINEQ
  (\\TRANSLATE-CL\\:LAMBDA
    (LAMBDA (EXPR)
      ; Edited 13-Feb-87 23:20 by Pavel
      (LET
        (VRBLS KEYVARS OPTVARS AUXLIST RESTFORM VARTYP BODY KEYWORDS (CNT 1)
          (MIN 0)
          (MAX 0)
          DECLS
          (SIMPLEP T))
        (|for| BINDING VAR |in| (CAR (CDR EXPR))
          |do|
            (SELECTQ BINDING
              ((&REST &BODY)
                (SETQ VARTYP ' &REST))
              (&OPTIONAL (SETQ VARTYP BINDING))
              (&AUX (SETQ VARTYP BINDING))
              (&ALLOW-OTHER-KEYS
                (OR (EQ VARTYP ' &KEY)
                  (ERROR "&ALLOW-OTHER-KEYS not in &KEY"))))
              (&KEY (SETQ VARTYP ' &KEY))
              (SELECTQ VARTYP
                (NIL "required" (|push| VRBLS BINDING)
                  (|add| CNT 1)
                  (|add| MIN 1)
                  (|add| MAX 1)
                  (AND *CHECK-ARGUMENT-COUNTS* (SETQ SIMPLEP NIL)))
                (&REST (SETQ RESTFORM `((, BINDING (|for| I |from| , CNT |to| |-args-| |collect| (ARG |-args-| I))))
                  (SETQ MAX NIL)
                  (SETQ SIMPLEP NIL))
                (&AUX (|push| AUXLIST BINDING))
                (&KEY (LET*
                  (SVAR (INIT (COND
                    ((LISTP BINDING)
                     (PROG1 (CADR BINDING)
                       (SETQ SVAR (CADDR BINDING))
                       (SETQ BINDING (CAR BINDING))))))
                    (KEY (COND
                      ((LISTP BINDING)
                       (PROG1 (CAR BINDING)
                         (SETQ BINDING (CADR BINDING))))
                      (T (MAKE-KEYWORD BINDING))))))
                  (COND
                    (SVAR (|push| KEYVARS (LIST SVAR T))))
                  (|push| KEYVARS
                    (LIST BINDING
                      `(|for| \\INDEX |from| , CNT |to| |-args-| |by| 2

```

```

      (when| (EQ (ARG |-args-| \\INDEX)
                ,KEY)
      |do| (RETURN (ARG |-args-| (ADD1 \\INDEX)))
      |finally| (RETURN , (COND
                          (SVAR `(PROGN (SETQ ,SVAR NIL)
                                          ,INIT))
                          (T INIT))))))
    (SETQ MAX NIL)
    (SETQ SIMPLEP NIL))
  (&OPTIONAL (OR (LISTP BINDING)
                 (SETQ BINDING (LIST BINDING)))
    (LET ((SVAR (CADDR BINDING))
          (CL:WHEN SVAR
            (push| OPTVARS SVAR)
            (SETQ SIMPLEP NIL))
          (CL:WHEN (CADR BINDING)
            (SETQ SIMPLEP NIL))
          (push| OPTVARS `',(CAR BINDING)
            (COND
              ((IGREATERP ,CNT |-args-|)
               , (CADR BINDING))
              (T ,@(COND
                     (SVAR `(SETQ ,SVAR T)))
                     (ARG |-args-| ,CNT))))))
      (AND MAX (|add| MAX 1))
      (|add| CNT 1))
    (SHOULDNT)))
  (CL:MULTIPLE-VALUE-SETQ (BODY DECLS)
    (PARSE-BODY (CDR (CDR EXPR))
      NIL))
  (CL:IF SIMPLEP
    `',(LAMBDA (,@ (REVERSE VRBLS)
                  ,@(MAPCAR (REVERSE OPTVARS)
                           (FUNCTION CAR)))
      (DECLARE (LOCALVARS . T))
      ,@DECLS
      (,'LET* (,@ (REVERSE AUXLIST))
              ,@DECLS
              ,@BODY))
    (LAMBDA |-args-|
      (DECLARE (LOCALVARS . T))
      ,@(COND
        ((AND *CHECK-ARGUMENT-COUNTS* MIN (NEQ MIN 0))
         `((COND
            ((ILESSP ,'| -args-| ,MIN)
             (ERROR "Too few args" ,'| -args-|))))))
        ,@(COND
          ((AND *CHECK-ARGUMENT-COUNTS* MAX)
           `((COND
              ((IGREATERP ,'| -args-| ,MAX)
               (ERROR "Too many args" ,'| -args-|))))))
          (,'LET* (,@ (|for| VAR |in| (REVERSE VRBLS) |as| I |from| 1 |collect| (LIST VAR `(ARG |-args-| ,I)))
                  ,@(REVERSE OPTVARS)
                  ,@(REVERSE KEYVARS)
                  ,@RESTFORM
                  ,@(REVERSE AUXLIST))
                  ,@DECLS
                  ,@BODY))))))
  )

(CL:DEFPARAMETER *CHECK-ARGUMENT-COUNTS* NIL)

(DEFGLOBALVAR *SPECIAL-BINDING-MARK* "Variable specially bound. This string should never be visible")

(CL:DEFCONSTANT CL:LAMBDA-LIST-KEYWORDS '(&OPTIONAL &REST &KEY &AUX &BODY &WHOLE &ALLOW-OTHER-KEYS
                                             &ENVIRONMENT &CONTEXT))

(CL:DEFCONSTANT CL:CALL-ARGUMENTS-LIMIT 512)

(CL:DEFCONSTANT CL:LAMBDA-PARAMETERS-LIMIT 512)

(CL:DEFSTRUCT (CLOSURE (:PRINT-FUNCTION (LAMBDA (CLOSURE STREAM)
                                                  (LET ((*PRINT-RADIX* NIL))
                                                      (CL:FORMAT STREAM "#<Interpreted closure @ ~O,~O>"
                                                                (\\HILOC CLOSURE)
                                                                (\\LOLOC CLOSURE))))))

```

;;; An interpreted lexical closure. Contains the function and an environment object.

FUNCTION

ENVIRONMENT)

```
(CL:DEFSTRUCT (ENVIRONMENT (:CONSTRUCTOR \\MAKE-ENVIRONMENT NIL)
  (:COPIER \\COPY-ENVIRONMENT)
  (:PRINT-FUNCTION (LAMBDA (ENV STREAM DEPTH)
    (DECLARE (IGNORE DEPTH))
    (LET ((*PRINT-RADIX* NIL))
      (CL:FORMAT STREAM "#<Lexical Environment @ ~O,~O>"
        (\\HILOC ENV)
        (\\LOLOC ENV))))))
```

;; An environment used by the Common Lisp interpreter. Every environment contains all of the information of its parents. That is, new child environments are made by copying the parent and then pushing new data onto one of the fields. This makes certain tests very fast.

;; Lexically-bound or -declared variables. A property list mapping names into either *SPECIAL-BINDING-MARK* or their values.

VARS

;; Lexical functions and macros. A property list mapping names into either (:function . fn) or (:macro . expansion-fn).

FUNCTIONS

;; A property list mapping block names into unique blips. RETURN-FROMs can throw to the appropriate blip.

BLOCKS

;; A property list mapping TAGBODY bodies into unique blips. GOs throw the correct tail of the body to the blip.

TAGBODIES)

```
(DEFMACRO \\MAKE-CHILD-ENVIRONMENT (PARENT &KEY ((:BLOCK (BLOCK-NAME BLOCK-BLIP))
  NIL BLOCK-P)
  ((:TAGBODY (TAGBODY-TAIL TAGBODY-BLIP))
  NIL TAGBODY-P))
  `(LET* (($PARENT ,PARENT)
    ($$NEW-ENV (CL:IF $PARENT
      (\\COPY-ENVIRONMENT $PARENT)
      (\\MAKE-ENVIRONMENT))))
    ,@(AND BLOCK-P `((CL:SETF (ENVIRONMENT-BLOCKS $$NEW-ENV)
      (LIST* ,BLOCK-NAME ,BLOCK-BLIP (ENVIRONMENT-BLOCKS $$NEW-ENV))))
    ,@(AND TAGBODY-P `((CL:SETF (ENVIRONMENT-TAGBODIES $$NEW-ENV)
      (LIST* ,TAGBODY-TAIL ,TAGBODY-BLIP (ENVIRONMENT-TAGBODIES $$NEW-ENV))))
    $$NEW-ENV)
```

(DEFINEQ

(CL:EVAL

(LAMBDA (CL::EXPRESSION CL::ENVIRONMENT)

; Edited 15-Apr-2024 20:00 by Imm

; Edited 1-Apr-92 12:39 by jds

;; This is in Interlisp and not a DEFUN to help avoid bootstrap death, although bootstrap death is quite possible anyway if, for example, any of the macros here are in Common Lisp and the macro definitions are interpreted.

(DECLARE (LOCALVARS . T))

(COND

((AND *EVALHOOK* (NOT (PROG1 CL::*SKIP-EVALHOOK* (CL:SETQ CL::*SKIP-EVALHOOK* NIL))))

(LET ((CL::HOOKFN *EVALHOOK*))

(*EVALHOOK* NIL))

(CL:FUNCALL CL::HOOKFN CL::EXPRESSION CL::ENVIRONMENT)))

(T (CL:TYPECASE CL::EXPRESSION

(CL:SYMBOL (COND

(NULL CL::EXPRESSION)

NIL)

(EQ CL::EXPRESSION T)

T)

(T (LET (CL::LOC CL::VAL)

(CL:BLOCK CL::EVAL-VARIABLE

(CL:WHEN CL::ENVIRONMENT

(|for| CL::TAIL |on| (ENVIRONMENT-VARS CL::ENVIRONMENT)

|by| (CDDR CL::TAIL) |when| (EQ CL::EXPRESSION (CAR CL::TAIL))

|do| (CL:SETQ CL::VAL (CADR CL::TAIL))

(COND

(EQ CL::VAL *SPECIAL-BINDING-MARK*))

;; return from FOR loop, skipping to SPECIALS code below.

(RETURN NIL))

(T (CL:RETURN-FROM CL::EVAL-VARIABLE CL::VAL))))))

;; following copied from EVALVAR in the Interlisp interpreter

(SETQ CL::LOC (\\STKSCAN CL::EXPRESSION))

(COND

(EQ (CL:SETQ CL::VAL (\\GETBASEPTR CL::LOC 0))

'NOBIND) ; Value is NOBIND even if it was not found as the top-level value.

(CL:ERROR 'UNBOUND-VARIABLE :NAME CL::EXPRESSION))

(T CL::VAL))))))

(CONS (COND

((CL:CONSP (CAR CL::EXPRESSION))

(LET ((CL::ARGCOUNT 1))

;; This is a very very awful hack for getting into internal lambda expressions

;; .COMPILER-SPREAD-ARGUMENTS. is handled specially by the compiler--it iterates over a list pushing things

;; secondly, the (OPCODES) directly calls EVAL-INVOKE-LAMBDA with more args than are given, blowing away
 ;; the following APPLYFN. Larry thought this level of hackery was important for performance.

```
(.COMPILER-SPREAD-ARGUMENTS. (CDR CL::EXPRESSION)
  CL::ARGCOUNT
  (CL-EVAL-FN3-CALL (CAR CL::EXPRESSION)
    CL::ENVIRONMENT)
  ((CL:EVAL CL::ENVIRONMENT))))
(T (LET ((CL::FN-DEFN (AND CL::ENVIRONMENT (CL:GETF (ENVIRONMENT-FUNCTIONS
  CL::ENVIRONMENT)
  (CAR CL::EXPRESSION)))))
  (COND
    ((NULL CL::FN-DEFN) ; The normal case: the function is not lexically-defined.
      (CASE (ARGTYPE (CAR CL::EXPRESSION))
        ((0 2)
          ;; has a Interlisp/CommonLisp lambda-spread definition
          (CL:IF (AND *APPLYHOOK* (NOT (PROG1 CL::*SKIP-APPLYHOOK*
            (CL:SETQ CL::*SKIP-APPLYHOOK* NIL)
            )))
            (LET* ((CL::ARGS (CL:MAPCAR #'(CL:LAMBDA (CL::ARG)
              (CL:EVAL CL::ARG
                CL::ENVIRONMENT))
                (CDR CL::EXPRESSION)))
              (CL::HOOKFN *APPLYHOOK*)
              (*APPLYHOOK* NIL))
              (CL:FUNCALL CL::HOOKFN (CAR CL::EXPRESSION)
                CL::ARGS CL::ENVIRONMENT))
            (LET ((CL::ARGCOUNT 0))
              (.COMPILER-SPREAD-ARGUMENTS. (CDR CL::EXPRESSION)
                CL::ARGCOUNT
                (CAR CL::EXPRESSION)
                ((CL:EVAL CL::ENVIRONMENT))))))
          (T ;; in Common Lisp, special form overrides nlambda definition
            ;; note that the GET will error if not a symbol.
            (LET ((CL::TEMP (AND (CL:SYMBOLP (CAR CL::EXPRESSION))
              (GET (CAR CL::EXPRESSION)
                'SPECIAL-FORM))))
              (COND
                (CL::TEMP ; CAR is the name of a special form.
                  (CL:FUNCALL CL::TEMP (CDR CL::EXPRESSION)
                    CL::ENVIRONMENT))
                ((CL:SETQ CL::TEMP (CL:MACRO-FUNCTION (CAR CL::EXPRESSION)))
                  ; CAR is the name of a macro
                  (CL:EVAL (CL:FUNCALL CL::TEMP CL::EXPRESSION CL::ENVIRONMENT
                    CL::ENVIRONMENT)
                    (T (ERROR "Undefined car of form" (CAR CL::EXPRESSION))))))
                ((EQ (CAR CL::FN-DEFN)
                  :MACRO) ; A use of a lexical macro.
                  (CL:EVAL (CL:FUNCALL (CDR CL::FN-DEFN)
                    CL::EXPRESSION CL::ENVIRONMENT)
                    CL::ENVIRONMENT))
                (T ; A call to a lexical function
                  (LET ((CL::ARGCOUNT 0))
                    (.COMPILER-SPREAD-ARGUMENTS. (CDR CL::EXPRESSION)
                      CL::ARGCOUNT
                      (CDR CL::FN-DEFN)
                      ((CL:EVAL CL::ENVIRONMENT)))))))))
    ))))
(T ;; 3.1.2.1.3 Self-Evaluating Objects
  ;; A form that is neither a symbol nor a cons is defined to be a self-evaluating object. Evaluating such an object yields the same
  ;; object as a result.
  ;; See https://interlisp.org/clhs/Issues/iss145_w
  CL::EXPRESSION)))))
```

(\EVAL-INVOKE-LAMBDA

```
(LAMBDA (N LAM ENV)
  (DECLARE (LOCALVARS . T)) ; Edited 28-Apr-87 11:55 by Pavel
  (LET ((ARGBLOCK (ADDSTACKBASE (- (FETCH (FX NEXTBLOCK) OF (\MYALINK))
    (+ (CL:DECF N)
      N))))))
```

;; First sub-form is a list of (variable initialization) pairs. Initializes the variables, binding them to new values all at once, then executes the
 ;; remaining forms as in a PROG.

```
(CL:MULTIPLE-VALUE-BIND (BODY SPECIALS)
  (\REMOVE-DECLS (CDDR LAM)
    (CL:SETQ ENV (\MAKE-CHILD-ENVIRONMENT ENV)))
  (\INTERPRET-ARGUMENTS "a LAMBDA as the CAR of a form" (CASE (CAR LAM)
    ((LAMBDA OPENLAMBDA) '&INTERLISP)
    ((CL:LAMBDA) '&REQUIRED)
    (T (CL:ERROR "~S ..." is not legal as
      the CAR of a form."
      (CAR LAM)))))
```

```
(CADR LAM)
SPECIALS ENV BODY ARGBLOCK N 0)))))
```

(\\INTERPRET-ARGUMENTS

```
(LAMBDA (\\FN-NAME \\ARGTYPE \\ARGLIST \\SPECIALS \\ENVIRONMENT \\BODY \\ARGUMENT-BLOCK \\LENGTH \\INDEX)
; Edited 7-Apr-88 16:16 by amd
```

```
:: Written in a somewhat arcane style to avoid recursive calls whenever possible, & keep code inline. RECUR does a recursive call if under a
:: PROGV, but otherwise does a GO.
```

```
(CL:MACROLET
  ((RECUR (TAG)
    (GO ,TAG))
   (WITH-BINDING (VAR VAL &REST FORMS)
    (PROGN (CHECK-BINDABLE ,VAR)
      (CL:IF (OR (FMEMB ,VAR \\SPECIALS)
        (VARIABLE-GLOBALLY-SPECIAL-P ,VAR))
        (CL:MACROLET ((RECUR (TAG)
          (\\INTERPRET-ARGUMENTS \\FN-NAME
            , (CL:IF (EQ TAG 'IN-KEYWORDS)
              '\\ARGTYPE
              '' ,TAG)
            \\ARGLIST \\SPECIALS \\ENVIRONMENT \\BODY \\ARGUMENT-BLOCK
            \\LENGTH \\INDEX)))
          (CL:PROGV (LIST ,VAR)
            (LIST ,VAL)
            ,@FORMS))
        (PROGN (CL:SETF (ENVIRONMENT-VARS \\ENVIRONMENT)
          (LIST* ,VAR ,VAL (ENVIRONMENT-VARS \\ENVIRONMENT)))
          ,@FORMS))))))
  (PROG (\\VAR \\VAL \\SVAR \\SP)
```

```
:: dispatch on input type. The in-keywords case is special, since it needs to pass down where the beginning of the keywords section is
```

```
(CASE \\ARGTYPE
  (&REQUIRED (GO &REQUIRED))
  (&OPTIONAL (GO &OPTIONAL))
  (&INTERLISP (GO &INTERLISP))
  (&REST (GO &REST))
  (&KEY (GO &KEY))
  (&AUX (GO &AUX))
  (&BODY (GO &BODY))
  (T (GO IN-KEYWORDS)))
&REQUIRED
  (RETURN (COND
    ((NULL \\ARGLIST)
      (CL:IF (< \\INDEX \\LENGTH)
        (CL:ERROR 'TOO-MANY-ARGUMENTS :CALLEE \\FN-NAME :ACTUAL \\LENGTH :MAXIMUM \\INDEX))
      (RECUR &BODY))
    (T (CASE (SETQ \\VAR (|pop| \\ARGLIST))
      (&OPTIONAL (RECUR &OPTIONAL))
      (&REST (RECUR &REST))
      (&AUX (RECUR &AUX))
      (&KEY (RECUR &KEY))
      (T (COND
        ((>= \\INDEX \\LENGTH)
          (CL:ERROR 'TOO-FEW-ARGUMENTS :CALLEE \\FN-NAME :ACTUAL \\LENGTH :MINIMUM
            (+ 1 \\INDEX
              (FOR ARG IN \\ARGLIST
                WHILE (NOT (FMEMB ARG '(&OPTIONAL &REST &AUX &KEY)))
                SUM 1))))))
        (SETQ \\VAL (ARG-REF \\ARGUMENT-BLOCK (PROG1 \\INDEX (CL:INCF \\INDEX))))
        (WITH-BINDING \\VAR \\VAL (RECUR &REQUIRED)))))))
&OPTIONAL
  (RETURN (COND
    ((NULL \\ARGLIST)
      (CL:IF (< \\INDEX \\LENGTH)
        (CL:ERROR 'TOO-MANY-ARGUMENTS :CALLEE \\FN-NAME :ACTUAL \\LENGTH :MAXIMUM \\INDEX))
      (RECUR &BODY))
    (T (CASE (SETQ \\VAR (|pop| \\ARGLIST))
      (&REST (RECUR &REST))
      (&AUX (RECUR &AUX))
      (&KEY (RECUR &KEY))
      (T (CL:IF (>= \\INDEX \\LENGTH)
        (CL:IF (CL:CONSP \\VAR)
          (PROGN (SETQ \\VAL (CL:EVAL (CADR \\VAR)
            \\ENVIRONMENT))
            (SETQ \\SVAR (CADDR \\VAR))
            (SETQ \\VAR (CAR \\VAR))
            (SETQ \\SP NIL))
          (SETQ \\VAL NIL))
        (PROGN (COND
          ((CL:CONSP \\VAR)
            (SETQ \\SVAR (CADDR \\VAR))
            (SETQ \\SP T)
            (SETQ \\VAR (CAR \\VAR))))
          (SETQ \\VAL (ARG-REF \\ARGUMENT-BLOCK \\INDEX))
          (CL:INCF \\INDEX))))))
```



```

(WITH-BINDING \\VAR \\VAL (CL:IF \\SVAR
(WITH-BINDING \\SVAR \\SP (RECUR &OPTIONAL))
(RECUR &OPTIONAL))))))

&INTERLISP
(RETURNS (COND
  ((NULL \\ARGLIST)
   (RECUR &BODY))
  (T (SETQ \\VAR (|pop| \\ARGLIST))
   (CL:IF (>= \\INDEX \\LENGTH)
    (SETQ \\VAL NIL)
    (PROGN (SETQ \\VAL (ARG-REF \\ARGUMENT-BLOCK \\INDEX))
     (CL:INCF \\INDEX))))
   (WITH-BINDING \\VAR \\VAL (RECUR &INTERLISP))))))

&REST
(SETQ \\VAR (|pop| \\ARGLIST))
(SETQ \\VAL (|for| I |from| \\INDEX |while| (< I \\LENGTH) |collect| (ARG-REF \\ARGUMENT-BLOCK I)))
(RETURNS (WITH-BINDING \\VAR \\VAL (CL:IF (NULL \\ARGLIST)
  (RECUR &BODY)
  (CASE (|pop| \\ARGLIST)
    (&AUX (RECUR &AUX))
    (&KEY (RECUR &KEY))
    (T (CL:ERROR 'INVALID-ARGUMENT-LIST :CALLEE \\FN-NAME))))))
))

&KEY
(OR (EVENP (- \\LENGTH \\INDEX))
  (CL:ERROR "Not an even number of arguments for &KEY"))
(SETQ \\ARGTYPE \\ARGLIST) ; Type is now the beginning of the keyword arguments
IN-KEYWORDS
(RETURNS (COND
  ((NULL \\ARGLIST)
   (CHECK-KEYWORDS \\ARGTYPE \\ARGUMENT-BLOCK \\LENGTH \\INDEX)
   (RECUR &BODY))
  (T (CASE (SETQ \\VAR (|pop| \\ARGLIST))
    (&AUX
     (CHECK-KEYWORDS \\ARGTYPE \\ARGUMENT-BLOCK \\LENGTH \\INDEX)
     (RECUR &AUX))
    (&ALLOW-OTHER-KEYS (CL:IF (NULL \\ARGLIST)
      (RECUR &BODY)
      (CASE (|pop| \\ARGLIST)
        (&AUX (RECUR &AUX))
        (T (CL:ERROR 'INVALID-ARGUMENT-LIST :CALLEE \\FN-NAME))))))
    ))
  ))

(T (COND
  ((CL:CONSP \\VAR)
   (SETQ \\VAL (CADR \\VAR))
   (SETQ \\SVAR (CADDR \\VAR))
   (SETQ \\VAR (CAR \\VAR)))
  (T (SETQ \\SVAR NIL)
   (SETQ \\VAL NIL)))
  (LET ((KEY (CL:IF (CL:CONSP \\VAR)
    (PROG1 (CAR \\VAR)
      (SETQ \\VAR (CADR \\VAR)))
    (MAKE-KEYWORD \\VAR))))
    (|for| I |from| \\INDEX |while| (< I \\LENGTH) |by| 2
    |do| (CL:IF (EQ (ARG-REF \\ARGUMENT-BLOCK I)
      KEY)
      (RETURNS (PROGN (SETQ \\VAL (ARG-REF \\ARGUMENT-BLOCK
        (+ I 1)))
        (SETQ \\SP T))))
      (|finally| (SETQ \\VAL (CL:EVAL \\VAL \\ENVIRONMENT))
        (SETQ \\SP NIL)))
    (WITH-BINDING \\VAR \\VAL (CL:IF \\SVAR
      (WITH-BINDING \\SVAR \\SP (RECUR IN-KEYWORDS))
      (RECUR IN-KEYWORDS))))))

&AUX
(RETURNS (COND
  ((NULL \\ARGLIST)
   (RECUR &BODY))
  (T (SETQ \\VAR (|pop| \\ARGLIST))
   (CL:IF (CL:CONSP \\VAR)
    (PROGN (SETQ \\VAL (CL:EVAL (CADR \\VAR)
      \\ENVIRONMENT))
      (SETQ \\VAR (CAR \\VAR)))
    (SETQ \\VAL NIL))
   (WITH-BINDING \\VAR \\VAL (RECUR &AUX))))))

&BODY
(RETURNS (CL:IF (NULL (CDR \\BODY))
  (CL:IF (CL:CONSP (SETQ \\BODY (CAR \\BODY)))
    (CASE (CAR \\BODY)
      (CL:BLOCK
       ;; special case to handle BLOCK to avoid consing two environments just to enter a normal LAMBDA
       ;; function
       (LET ((BLIP (CONS NIL NIL)))
        (CL:SETF (ENVIRONMENT-BLOCKS \\ENVIRONMENT)
          (LIST* (CADR \\BODY)
            BLIP

```

```

                                (ENVIRONMENT-BLOCKS \\ENVIRONMENT)))
                                (CL:CATCH BLIP
                                  (\\EVAL-PROGN (CDDR \\BODY)
                                                  \\ENVIRONMENT))))
                                (T (CL:EVAL \\BODY \\ENVIRONMENT)))
                                (CL:EVAL \\BODY \\ENVIRONMENT))
                                (PROGN (CL:EVAL (POP \\BODY)
                                                  \\ENVIRONMENT)
                                      (RECUR &BODY))))))

```

(\\INTERPRETER-LAMBDA

```

(LAMBDA (N DEF ENV FN) ; Edited 13-Feb-87 21:21 by Pavel
  (DECLARE (LOCALVARS . T))
  (LET ((ARGBLOCK (ADDSTACKBASE ((|fetch| (BF IVAR) |of| (|fetch| (FX BLINK) |of| (\\MYALINK))))))
        (SETQ ENV (\\MAKE-CHILD-ENVIRONMENT ENV))
        (CL:MULTIPLE-VALUE-BIND (BODY SPECIALS)
          (\\REMOVE-DECLS (CDR (CDR DEF))
                          ENV)
          (\\INTERPRET-ARGUMENTS FN '&REQUIRED (CAR (CDR DEF))
                                   SPECIALS ENV BODY ARGBLOCK (- N 1)
                                   0))))))

```

(CHECK-BINDABLE

```

(LAMBDA (VAR) ; Edited 13-Feb-87 22:06 by Pavel
  (CL:UNLESS (CL:SYMBOLP VAR)
    (CL:ERROR "Attempt to bind a non-symbol: ~S" VAR))
  (CL:WHEN (OR (CL:CONSTANTP VAR)
    (FMEMB VAR CL:LAMBDA-LIST-KEYWORDS))
    (CL:ERROR (CL:IF (CL:KEYWORDP VAR)
      "Attempt to bind a keyword: ~S"
      "Attempt to bind a constant: ~S")
      VAR))
  (CL:WHEN (VARIABLE-GLOBAL-P VAR)
    (CL:CERROR "Go ahead and bind it anyway" "Attempt to bind a variable proclaimed global: ~S" VAR))
  VAR))

```

(CHECK-KEYWORDS

```

(LAMBDA (KEY-ARGUMENTS ARGBLOCK LENGTH N) ; Edited 1-Dec-87 16:47 by amd
  ;; check to see if any keywords in ARGBLOCK are not in the keys - not called if &ALLOW-OTHER-KEYS was set
  (CL:BLOCK CHECK-KEYS
    (LET (BADKEYWORD)
      (CL:DO ((I N (+ I 2)))
        ((>= I LENGTH)
          (LET ((GIVEN-KEY (ARG-REF ARGBLOCK I)))
            (CL:IF (EQ GIVEN-KEY :ALLOW-OTHER-KEYS)
              (CL:IF (ARG-REF ARGBLOCK (CL:1+ I))
                (CL:RETURN-FROM CHECK-KEYS NIL)
                NIL)
              (CL:DO ((KEYTAIL KEY-ARGUMENTS (CDR KEYTAIL))
                ((OR (NULL KEYTAIL)
                  (EQ (CAR KEYTAIL)
                      '&AUX))) ; got to end of keyword segment
                (SETQ BADKEYWORD GIVEN-KEY))
              (LET ((WANTED-KEY (CAR KEYTAIL)))
                (IF (CL:CONSP WANTED-KEY)
                  THEN (SETQ WANTED-KEY (CAR WANTED-KEY))
                  (CL:IF (CL:CONSP WANTED-KEY)
                    (SETQ WANTED-KEY (CAR WANTED-KEY))
                    (SETQ WANTED-KEY (MAKE-KEYWORD WANTED-KEY)))
                  ELSE (SETQ WANTED-KEY (MAKE-KEYWORD WANTED-KEY))
                  (CL:IF (EQ WANTED-KEY GIVEN-KEY)
                    (RETURN NIL))))))
            (CL:IF BADKEYWORD (CL:ERROR "Keyword argument doesn't match expected list of keywords: ~A"
              BADKEYWORD))))))
  )

```

```

(DEFMACRO ARG-REF (BLOCK N)
  ` (\\GETBASEPTR ,BLOCK (LLSH ,N 1)))

```

```

(PUTPROPS .COMPILER-SPREAD-ARGUMENTS. DMACRO (APPLY COMP.SPREAD))

```

```

(DEFINEQ

```

(DECLARED-SPECIAL

```

(LAMBDA (VAR DECLS) (* |Imm| "24-May-86 22:27")
  (AND DECLS (OR (AND (LISTP (CAR DECLS))
    (EQ (CAAR DECLS)
        'DECLARE)
    (|for| DEC |in| (CDAR DECLS) |when| (AND (EQ (CAR DEC)
        'CL:SPECIAL)
        (FMEMB VAR (CDR DEC))))

```

```

      |do| (RETURN T)))
    (DECLARED-SPECIAL VAR (CDR DECLS))))))

```

```

)

```

```

;; FUNCALL and APPLY, not quite same as Interlisp

```

```

(DEFINEQ

```

```

(CL:FUNCALL

```

```

  (CL:LAMBDA (CL::FN &REST CL::ARGS)
    (CL:APPLY CL::FN CL::ARGS))

```

```

; Edited 14-Feb-87 00:16 by Pavel

```

```

(CL:APPLY

```

```

  (LAMBDA CL::N
    (CL:IF (EQ CL::N 0)
      (ERROR "TOO FEW ARGUMENTS TO APPLY")
      (SPREADAPPLY (ARG CL::N 1)
        (LET ((CL::AV (ARG CL::N CL::N)))
          (FOR CL::I FROM (CL:1- CL::N) TO 2 BY -1 DO (CL:PUSH (ARG CL::N CL::I)
                                                                CL::AV))
          CL::AV))))))

```

```

; Edited 14-Feb-87 00:16 by Pavel

```

```

)

```

```

(PUTPROPS CL:APPLY DMACRO (DEFMACRO (FN &REST ARGS) (CASE COMPILE.CONTEXT
  (EFFECT RETURN)
    `(LET ((FN ,FN)
          (CNT , (LENGTH (CDR ARGS))))
      (.SPREAD. ((OPCODES)
                \,@ ARGS)
                CNT FN)))
  (T ;; otherwise might not return multiple values
    'IGNOREMACRO))))

```

```

(PUTPROPS CL:FUNCALL DMACRO (DEFMACRO (FN &REST ARGS) (COND
  ((AND (NLISTP FN)
        (EVERY ARGS (FUNCTION NLISTP))))
    `((OPCODES APPLYFN)
      ,@ARGS
      , (LENGTH ARGS)
      , FN))
  (T (LET ((TEM (GENSYM)))
      `((LAMBDA (,TEM)
        ((OPCODES APPLYFN)
         ,@ARGS
         , (LENGTH ARGS)
         , TEM))
        , FN))))))

```

```

;; COMPILER-LET needs to work differently compiled and interpreted

```

```

(DEFINEQ

```

```

(CL:COMPILER-LET

```

```

  (NLAMBDA $$COMPILER-LET-TAIL
    (CL:PROGV (|for| X |in| (CAR $$COMPILER-LET-TAIL) |collect| (COND
      ((CL:CONSP X)
       (CAR X))
      (T X)))
      (|for| X |in| (CAR $$COMPILER-LET-TAIL) |collect| (COND
        ((CL:CONSP X)
         (\\EVAL (CADR X))))
        (\\EVPROGN (CDR $$COMPILER-LET-TAIL))))))

```

```

; Edited 7-Apr-88 16:05 by amd

```

```

(COMP.COMPILER-LET

```

```

  (LAMBDA (\\A)
    (DECLARE (LOCALVARS . T))
    (CL:PROGV (|for| X |in| (CAR \\A) |collect| (|if| (CL:CONSP X)
      |then| (CAR X)
      |else| X))
      (|for| X |in| (CAR \\A) |collect| (COND
        ((CL:CONSP X)
         (EVAL (CADR X))))
        (COMP.PROGN (CDR \\A))))))

```

```

; Edited 7-Apr-88 16:38 by amd
(* ENTRY POINT INTO BYTECOMPILER)
(* |Imm| "27-May-86 11:17")

```

```

)

```

```

(PUTPROPS CL:COMPILER-LET DMACRO COMP.COMPILER-LET)

```

```
(DEFINE-SPECIAL-FORM CL:COMPILER-LET (CL::ARGS &REST CL::BODY &ENVIRONMENT CL::ENV)
  (LET ((*IN-COMPILER-LET* T))
    (DECLARE (CL:SPECIAL *IN-COMPILER-LET*))
    (CL:PROGV (FOR CL::X IN CL::ARGS COLLECT (IF (CL:CONSP CL::X)
      THEN (CAR CL::X)
      ELSE CL::X))
      (FOR CL::X IN CL::ARGS COLLECT (IF (CL:CONSP CL::X)
      THEN (CL:EVAL (CADR CL::X)
        CL::ENV)
      ELSE NIL))
      (\\EVAL-PROGN CL::BODY CL::ENV))))
```

;; Lexical function- and macro-binding forms: FLET, LABELS, and MACROLET.

```
(DEFINE-SPECIAL-FORM CL:MACROLET (CL::MACRO-DEFNS &BODY CL::BODY &ENVIRONMENT CL::ENV)
  (LET* ((CL::NEW-ENV (\\MAKE-CHILD-ENVIRONMENT CL::ENV))
    (CL::FUNCTIONS (ENVIRONMENT-FUNCTIONS CL::NEW-ENV)))
    (FOR CL::MACRO-DEFN IN CL::MACRO-DEFNS
      DO (CL:SETQ CL::FUNCTIONS (LIST* (CAR CL::MACRO-DEFN)
        (CONS :MACRO `(CL:LAMBDA (SI::$$MACRO-FORM SI::$$MACRO-ENVIRONMENT
          )
            (CL:BLOCK , (CAR CL::MACRO-DEFN)
              , (PARSE-DEFMACRO (CADR CL::MACRO-DEFN)
                'SI::$$MACRO-FORM
                (CDDR CL::MACRO-DEFN)
                (CAR CL::MACRO-DEFN)
                NIL :ENVIRONMENT
                'SI::$$MACRO-ENVIRONMENT))))
          CL::FUNCTIONS)))
    (CL:SETF (ENVIRONMENT-FUNCTIONS CL::NEW-ENV)
      CL::FUNCTIONS)
    (\\EVAL-PROGN CL::BODY CL::NEW-ENV)))
```

```
(DEFINE-SPECIAL-FORM CL:FLET (CL::FN-DEFNS &BODY CL::BODY &ENVIRONMENT CL::ENV)
  (LET* ((CL::NEW-ENV (\\MAKE-CHILD-ENVIRONMENT CL::ENV))
    (CL::FUNCTIONS (ENVIRONMENT-FUNCTIONS CL::NEW-ENV)))
    (FOR CL::FN-DEFN IN CL::FN-DEFNS
      DO (CL:SETQ CL::FUNCTIONS (LIST* (CL:FIRST CL::FN-DEFN)
        (CONS :FUNCTION
          (MAKE-CLOSURE
            :FUNCTION
            (CL:MULTIPLE-VALUE-BIND (CL::BODY CL::DECLS)
              (PARSE-BODY (CDDR CL::FN-DEFN)
                CL::ENV T)
              `(CL:LAMBDA , (CL:SECOND CL::FN-DEFN)
                ,@CL::DECLS
                (CL:BLOCK , (CL:FIRST CL::FN-DEFN)
                  ,@CL::BODY)))
            :ENVIRONMENT CL::ENV)
          CL::FUNCTIONS)))
    (CL:SETF (ENVIRONMENT-FUNCTIONS CL::NEW-ENV)
      CL::FUNCTIONS)
    (\\EVAL-PROGN CL::BODY CL::NEW-ENV)))
```

```
(DEFINE-SPECIAL-FORM CL:LABELS (CL::FN-DEFNS &BODY CL::BODY &ENVIRONMENT CL::ENV)
  (LET* ((CL::NEW-ENV (\\MAKE-CHILD-ENVIRONMENT CL::ENV))
    (CL::FUNCTIONS (ENVIRONMENT-FUNCTIONS CL::NEW-ENV)))
    (FOR CL::FN-DEFN IN CL::FN-DEFNS
      DO (CL:SETQ CL::FUNCTIONS (LIST* (CL:FIRST CL::FN-DEFN)
        (CONS :FUNCTION
          ;; Must share the environment object so that all of the new lexical function bindings
          ;; appear in each new functions environment.
          (MAKE-CLOSURE
            :FUNCTION
            (CL:MULTIPLE-VALUE-BIND (CL::BODY CL::DECLS)
              (PARSE-BODY (CDDR CL::FN-DEFN)
                CL::NEW-ENV T)
              `(CL:LAMBDA , (CL:SECOND CL::FN-DEFN)
                ,@CL::DECLS
                (CL:BLOCK , (CL:FIRST CL::FN-DEFN)
                  ,@CL::BODY)))
            :ENVIRONMENT CL::NEW-ENV)
          CL::FUNCTIONS)))
    (CL:SETF (ENVIRONMENT-FUNCTIONS CL::NEW-ENV)
      CL::FUNCTIONS)
    (\\EVAL-PROGN CL::BODY CL::NEW-ENV)))
```

```
(DEFINE-SPECIAL-FORM QUOTE CAR)
```

```
(DEFINE-SPECIAL-FORM THE (CL::TYPE-SPEC CL::FORM &ENVIRONMENT CL::ENV)
```

```

(CL:IF (AND (CL:CONSP CL::TYPE-SPEC)
            (EQ (CAR CL::TYPE-SPEC)
                'CL:VALUES))
  (LET ((CL:VALUES (CL:MULTIPLE-VALUE-LIST (CL:EVAL CL::FORM CL::ENV))))
    (CL:IF (CL:NOTEVERY #'(CL:LAMBDA (CL::VALUE CL::SPEC)
                                     (TYPEP CL::VALUE CL::SPEC))
              CL:VALUES
              (CDR CL::TYPE-SPEC))
      (CHECK-TYPE-FAIL T CL::FORM CL:VALUES CL::TYPE-SPEC NIL)
      (CL:VALUES-LIST CL:VALUES)))
  (LET ((CL::VALUE (CL:EVAL CL::FORM CL::ENV)))
    (CL:IF (TYPEP CL::VALUE CL::TYPE-SPEC)
      CL::VALUE
      (CHECK-TYPE-FAIL T CL::FORM CL::VALUE CL::TYPE-SPEC NIL))))))

(PUTPROPS THE DMACRO ((SPEC FORM)
                     FORM))

(PUTPROPS CL:EVAL-WHEN DMACRO (DEFMACRO (OPTIONS &BODY BODY) (AND (OR (FMEMB 'COMPILE OPTIONS)
                                                                    (FMEMB 'CL:COMPILE OPTIONS))
                                                                    (MAPC BODY (FUNCTION CL:EVAL)))
                                (AND (OR (FMEMB 'LOAD OPTIONS)
                                         (FMEMB 'CL:LOAD OPTIONS))
                                     `(PROGN ,@BODY))))))

```

(DEFINEQ

```

(CL:EVAL-WHEN
  (NLAMBDA OPTIONS.BODY
    (AND (OR (FMEMB 'CL:EVAL (CAR OPTIONS.BODY))
              (FMEMB 'EVAL (CAR OPTIONS.BODY)))
      (MAPC (CDR OPTIONS.BODY)
        (FUNCTION \\EVAL))))))
(* |Imm| " 1-Jun-86 15:16")
)

```

```

(DEFINE-SPECIAL-FORM CL:EVAL-WHEN (CL::TAGS &REST CL::BODY &ENVIRONMENT CL::ENV)
  (AND (OR (CL:MEMBER 'CL:EVAL CL::TAGS)
            (CL:MEMBER 'EVAL CL::TAGS))
    (\\EVAL-PROGN CL::BODY CL::ENV)))

```

(DEFINE-SPECIAL-FORM DECLARE FALSE)

```

(DEFMACRO CL:LOCALLY (&BODY BODY)
  `(LET NIL ,@BODY))

```

;; Interlisp version on LLINTERP

(DEFINE-SPECIAL-FORM PROGN \\EVAL-PROGN)

(DEFINEQ

```

(\\EVAL-PROGN
  (LAMBDA (BODY ENVIRONMENT)
    (|if| (CDR BODY)
      |then| (CL:EVAL (CAR BODY)
                     ENVIRONMENT)
      (\\EVAL-PROGN (CDR BODY)
                     ENVIRONMENT)
    |else| (CL:EVAL (CAR BODY)
                   ENVIRONMENT))))
; Edited 12-Feb-87 20:25 by Pavel
)

```

;; Confused because currently Interlisp special form, fixing MACRO-FUNCTION is complex

;; The Interlisp function is on LLINTERP

```

(DEFINE-SPECIAL-FORM PROG1 (CL:FIRST &REST CL:REST &ENVIRONMENT CL::ENV)
  (LET ((CL::VAL (CL:EVAL CL:FIRST CL::ENV)))
    (CL:TAGBODY PROG1 (CL:IF CL:REST
                             (PROGN (CL:EVAL (CAR CL:REST)
                                             CL::ENV)
                                     (CL:SETQ CL:REST (CDR CL:REST)))
                             (CL:RETURN-FROM PROG1 CL::VAL))
      (GO PROG1))))

```

```

(DEFMACRO PROG1 (CL:FIRST &REST CL:REST)
  `(LET ((SI::$PROG1-FIRST-EXPRESSION$ ,CL:FIRST))
    (DECLARE (LOCALVARS SI::$PROG1-FIRST-EXPRESSION$))
    ,@CL:REST SI::$PROG1-FIRST-EXPRESSION$))

```

```

(DEFINE-SPECIAL-FORM LET* (CL::VARS &REST CL::BODY &ENVIRONMENT CL::ENV)
  (CL:MULTIPLE-VALUE-BIND (CL::BODY CL::SPECIALS)
    (\\REMOVE-DECLS CL::BODY (CL:SETQ CL::ENV (\\MAKE-CHILD-ENVIRONMENT CL::ENV)))
    (\\LET*-RECURSION CL::VARS CL::SPECIALS CL::ENV CL::BODY)))

(DEFINE-SPECIAL-FORM LET (CL::VARS &BODY CL::BODY &ENVIRONMENT CL::ENV &AUX CL::\\NEW-ENV)
  ;; Initializes the variables, binding them to new values all at once, then executes the remaining forms as in a PROG.
  (CL:MULTIPLE-VALUE-BIND (CL::\\BODY CL::SPECIALS)
    (\\REMOVE-DECLS CL::BODY (CL:SETQ CL::\\NEW-ENV (\\MAKE-CHILD-ENVIRONMENT CL::ENV)))
    ;; Note that since remove decls side-effects the environment, variables which are declared special inside this scope will cause references inside
    ;; the variable value forms to do special reference.
    (LET ((CL::ENV-VARS (ENVIRONMENT-VARS CL::\\NEW-ENV))
          CL::SPECVARS CL::SPECVALS CL::VALUE)
      (FOR CL::VAR IN CL::VARS DO (COND
        ((CL:CONSP CL::VAR)
          ;; NEW-ENV current has all of the new specials, but none of the new lexicals. This is the right
          ;; environment to eval in.
          (CL:SETQ CL::VALUE (CL:EVAL (CADR CL::VAR)
                                     CL::\\NEW-ENV))
          (CL:SETQ CL::VAR (CAR CL::VAR)))
        (T (CL:SETQ CL::VALUE NIL)))
      (CHECK-BINDABLE CL::VAR)
      (IF (OR (FMEMB CL::VAR CL::SPECIALS)
              (VARIABLE-GLOBALLY-SPECIAL-P CL::VAR))
          THEN (CL:PUSH CL::VAR CL::SPECVARS)
              (CL:PUSH CL::VALUE CL::SPECVALS)
          ELSE (CL:SETQ CL::ENV-VARS (LIST* CL::VAR CL::VALUE CL::ENV-VARS))))
    (CL:SETF (ENVIRONMENT-VARS CL::\\NEW-ENV)
      CL::ENV-VARS)
    (CL:IF CL::SPECVARS
      (CL:PROGV CL::SPECVARS CL::SPECVALS
        (\\EVAL-PROGN CL::\\BODY CL::\\NEW-ENV))
      (\\EVAL-PROGN CL::\\BODY CL::\\NEW-ENV))))

(PUTPROPS LET MACRO (X (|\\LETtran| X)))

(PUTPROPS LET* MACRO (X (|\\LETtran| X T)))

(DEFINEQ
  (\\LET*-RECURSION
    (LAMBDA (VARS $$LET*-SPECIALS $$LET*-ENV $$LET*-BODY)
      (DECLARE (LOCALVARS . T)) ; Edited 7-Apr-88 16:09 by amd
      (|bind| VAR VALUE |for| $$LET*-TAIL |on| VARS |eachtime| (SETQ VAR (CAR $$LET*-TAIL))
        |do| (COND
          ((CL:CONSP VAR)
            (SETQ VALUE (CL:EVAL (CADR VAR)
                                $$LET*-ENV))
            (SETQ VAR (CAR VAR)))
          (T (SETQ VALUE NIL)))
          (CHECK-BINDABLE VAR)
          (CL:IF (OR (FMEMB VAR $$LET*-SPECIALS)
                    (VARIABLE-GLOBALLY-SPECIAL-P VAR))
            (RETURN (CL:PROGV (LIST VAR)
                              (LIST VALUE)
                              (\\LET*-RECURSION (CDR $$LET*-TAIL)
                                                $$LET*-SPECIALS $$LET*-ENV $$LET*-BODY)))
            (CL:SETF (ENVIRONMENT-VARS $$LET*-ENV)
              (LIST* VAR VALUE (ENVIRONMENT-VARS $$LET*-ENV))))
        |finally| (RETURN (\\EVAL-PROGN $$LET*-BODY $$LET*-ENV))))))

(|\\LETtran|
  (LAMBDA (LETTAIL SEQUENTIALP) ; Edited 23-Dec-86 16:23 by Imm
    ;; Interlisp version of LET/LET*/PROG*
    (PROG ((VARS (MAPCAR (CAR LETTAIL)
      (FUNCTION (LAMBDA (BINDENTRY)
        (|if| (LISTP BINDENTRY)
          |then| (CAR BINDENTRY)
          |else| BINDENTRY))))))
      (VALS (MAPCAR (CAR LETTAIL)
        (FUNCTION (LAMBDA (BINDENTRY)
          (|if| (LISTP BINDENTRY)
            |then| (CADR BINDENTRY)
            |else| NIL))))))
      (BODY (CDR LETTAIL))
      (DECLS NIL))
    (CL:MULTIPLE-VALUE-SETQ (BODY DECLS)
      (PARSE-BODY BODY NIL))
    (RETURN (|if| (NOT SEQUENTIALP)

```

```

|then| `((, 'LAMBDA , VARS
        , @DECLS
        , @BODY)
        , @VALS)
|elseif| (NULL (CDR VARS))
|then| (SELECTQ SEQUENTIALP
        (PROG* `(PROG , @LETTAIL))
        `((, 'LAMBDA , VARS
            , @DECLS
            , @BODY)
            , @VALS))
|else|                                     ; in the sequential case, all declarations must be included in each
        (|if| (EQ SEQUENTIALP 'PROG*)
        |then| (SETQ BODY (LIST (LIST* 'PROG NIL BODY))))
        (|for| VAR |in| (REVERSE (CDR VARS)) |as| VAL |in| (REVERSE (CDR VALS))
        |do| (SETQ BODY `((, 'LAMBDA (, VAR
                                , @DECLS
                                , @BODY)
                                , VAL))))
        `(, 'LAMBDA (, (CAR VARS)
            , @DECLS
            , @BODY)
            , (CAR VALS))))))
)

```

```

(DEFINE-SPECIAL-FORM COND (&REST CL::COND-CLAUSES &ENVIRONMENT CL::ENVIRONMENT)
  (PROG NIL
    CL::CONDDOOP
    (COND
      (NULL CL::COND-CLAUSES)
      (RETURN NIL))
    (NULL (CDAR CL::COND-CLAUSES))
    (RETURN (OR (CL:EVAL (CAAR CL::COND-CLAUSES)
                        CL::ENVIRONMENT)
                (PROGN (CL:SETQ CL::COND-CLAUSES (CDR CL::COND-CLAUSES))
                        (GO CL::CONDDOOP))))))
    ((CL:EVAL (CAAR CL::COND-CLAUSES)
              CL::ENVIRONMENT)
     (RETURN (\\EVAL-PROGN (CDAR CL::COND-CLAUSES)
                           CL::ENVIRONMENT)))
    (T (CL:SETQ CL::COND-CLAUSES (CDR CL::COND-CLAUSES))
      (GO CL::CONDDOOP))))

```

```

(DEFMACRO COND (&REST CL::TAIL)
  (CL:IF CL::TAIL
    (CL:IF (NULL (CDAR CL::TAIL))
      (CL:IF (CDR CL::TAIL)
        (LET ((VAR (CL:GENTEMP)))
          `(LET ((,VAR , (CAAR CL::TAIL)))
              (CL:IF ,VAR
                    ,VAR
                    (COND
                      ,@(CDR CL::TAIL))))))
          `(CL:VALUES , (CAAR CL::TAIL)))
        `(CL:IF , (CAAR CL::TAIL)
          , (MKPROGN (CDAR CL::TAIL))
          ,@(CL:IF (CDR CL::TAIL)
              (LIST (CL:IF (EQ (CAADR CL::TAIL)
                            T)
                          (CL:IF (NULL (CDADR CL::TAIL))
                                T
                                (MKPROGN (CDADR CL::TAIL))))
                    , (COND
                      ,@(CDR CL::TAIL))))))))))

```

```

(DEFINEQ

```

```

(CL:IF
  (NLAMBDA (CL::TEST CL::THEN CL::ELSE)
    (DECLARE (LOCALVARS . T))
    (CL:IF (\\EVAL CL::TEST)
      (\\EVAL CL::THEN)
      (\\EVAL CL::ELSE))))
)

```

; Edited 12-Feb-87 20:27 by Pavel

```

(DEFINE-SPECIAL-FORM CL:IF (CL::TEST CL::THEN &OPTIONAL CL::ELSE &ENVIRONMENT CL::ENVIRONMENT)
  (CL:IF (CL:EVAL CL::TEST CL::ENVIRONMENT)
    (CL:EVAL CL::THEN CL::ENVIRONMENT)
    (CL:EVAL CL::ELSE CL::ENVIRONMENT)))

(PUTPROPS CL:IF DMACRO COMP IF)

```

;; Interlisp NLAMBDA definitions on LLINTERP
 ;; both special form and macro

```
(DEFMACRO AND (&REST CL::FORMS)
  (COND
    ((CDR CL::FORMS)
     `(CL:IF , (CAR CL::FORMS)
              (AND ,@ (CDR CL::FORMS))))
    (CL::FORMS (CAR CL::FORMS))
    (T T)))
```

```
(DEFMACRO OR (&REST CL::FORMS)
  (CL:IF (NULL (CDR CL::FORMS))
    (CAR CL::FORMS)
    `(LET ((SI::*OR-GENTEMP* , (CAR CL::FORMS)))
      (DECLARE (LOCALVARS SI::*OR-GENTEMP*))
      (CL:IF SI::*OR-GENTEMP*
        SI::*OR-GENTEMP*
        (OR ,@ (CDR CL::FORMS)))))))
```

```
(DEFINE-SPECIAL-FORM AND (&REST CL::AND-CLAUSES &ENVIRONMENT CL::ENV)
  (CL:LOOP (COND
    ((NULL CL::AND-CLAUSES)
     (RETURN T))
    ((NULL (CDR CL::AND-CLAUSES))
     (RETURN (CL:EVAL (CAR CL::AND-CLAUSES)
                       CL::ENV)))
    (T (CL:IF (CL:EVAL (CAR CL::AND-CLAUSES)
                       CL::ENV)
               (CL:POP CL::AND-CLAUSES)
               (RETURN NIL))))))
```

```
(DEFINE-SPECIAL-FORM OR (&REST CL::TAIL &ENVIRONMENT CL::ENV)
  (BIND CL::VAL FOR OLD CL::TAIL ON CL::TAIL (COND
    ((NULL (CDR CL::TAIL))
     (RETURN (CL:EVAL (CAR CL::TAIL)
                       CL::ENV)))
    ((CL:SETQ CL::VAL (CL:EVAL (CAR CL::TAIL)
                                CL::ENV))
     (RETURN CL::VAL))))
```

;; BLOCK and RETURN go together

```
(DEFINEQ
```

```
  (CL:BLOCK
```

```
    (NLAMBDA CL::TAIL
      (\\EVPROGN (CDR CL::TAIL))))
```

; Edited 12-Feb-87 20:31 by Pavel

```
)
```

```
(PUTPROPS CL:BLOCK DMACRO COMP.BLOCK)
```

```
(DEFINE-SPECIAL-FORM CL:BLOCK (CL::NAME &REST CL::\\BODY &ENVIRONMENT CL::ENVIRONMENT)
```

;; Syntax is (BLOCK name . body). The body is evaluated as a PROGN, but it is possible to exit the block using (RETURN-FROM name value).
 ;; The RETURN-FROM must be lexically contained within the block.

```
(LET* ((CL::BLIP (CONS NIL NIL))
      (CL::\\NEW-ENV (\\MAKE-CHILD-ENVIRONMENT CL::ENVIRONMENT :BLOCK (CL::NAME CL::BLIP))))
  (CL:CATCH CL::BLIP (\\EVAL-PROGN CL::\\BODY CL::\\NEW-ENV)))
```

```
(DEFMACRO RETURN (CL::VALUE)
  `(CL:RETURN-FROM NIL ,CL::VALUE))
```

```
(DEFINEQ
```

```
  (CL:RETURN-FROM
```

```
    (NLAMBDA (CL::RETFROM-TAG CL::RETFROM-VALUE)
```

```
      (DECLARE (LOCALVARS . T))
```

; Edited 12-Feb-87 20:35 by Pavel

```
      (LET ((CL::RETVALUES (CL:MULTIPLE-VALUE-LIST (\\EVAL CL::RETFROM-VALUE))))
```

```
        (LET ((CL::FRAME (STKNTH 1))))
```

```
        (WHILE CL::FRAME DO (CL:IF (OR (AND (NULL CL::RETFROM-TAG)
```

```
                                          (EQ (STKNAME CL::FRAME)
```

```
                                              '\\PROG0))
```

```
                                          (AND (EQ (STKNAME CL::FRAME)
```

```
                                              'CL:BLOCK)
```

```
                                          (EQ (CAR (STKARG 1 CL::FRAME))
```

```
                                              CL::RETFROM-TAG)))
```

```
        (RETVALUES CL::FRAME CL::RETVALUES T)
```

```
        (CL:SETQ CL::FRAME (STKNTH 1 CL::FRAME CL::FRAME)))
```



```

    FINALLY (CL:ERROR 'ILLEGAL-RETURN :TAG CL::RETFROM-TAG))))))
)

(DEFINE-SPECIAL-FORM CL:RETURN-FROM (CL::BLOCK-NAME CL::EXPR &ENVIRONMENT CL::ENV)
  (LET ((CL::BLIP (AND CL::ENV (CL:GETF (ENVIRONMENT-BLOCKS CL::ENV)
                                         CL::BLOCK-NAME))))
    (CL:IF (AND CL::BLOCK-NAME (NULL CL::BLIP))
      (CL:ERROR 'ILLEGAL-RETURN :TAG CL::BLOCK-NAME)
      (LET ((CL::\BLK CL::BLOCK-NAME)
            (CL::VALS (CL:MULTIPLE-VALUE-LIST (CL:EVAL CL::EXPR CL::ENV))))
        (COND
          (CL::BLIP ; This is a CL RETURN-FROM, so do the throw.
            (HANDLER-BIND ((ILLEGAL-THROW #'(CL:LAMBDA (CL::C)
                                                         (DECLARE (IGNORE CL::C))
                                                         (CL:ERROR 'ILLEGAL-RETURN :TAG CL::\BLK))))
              (CL:THROW CL::BLIP (CL:VALUES-LIST CL::VALS))))
          (T ; This is an IL RETURN, so return from the closest enclosing
            (RETVALUES (STKPOS '\PROG0)
                       CL::VALS T)))))))

```

;; IL and CL versions of FUNCTION.

```

(DEFINEQ
  (CL:FUNCTION
    (NLAMBDA (CL::FN) ; Edited 30-Jan-87 19:07 by Pavel

```

;;; Fake CL:FUNCTION for Interlisp --- no lexical closures

```

  (CL:IF (CL:SYMBOLP CL::FN)
    (CL:SYMBOL-FUNCTION CL::FN)
    (CL::FN))
)

(PUTPROPS CL:FUNCTION DMACRO (DEFMACRO (X) (COND
  ((CL:SYMBOLP X)
   (CL:SYMBOL-FUNCTION ',X))
  (T (FUNCTION ,X)))))

```

```

(DEFINE-SPECIAL-FORM CL:FUNCTION (CL::FN &ENVIRONMENT CL::ENVIRONMENT)
  (COND
    ((CL:SYMBOLP CL::FN)
     (LET (CL::FN-DEFN)
       (COND
         ((OR (NULL CL::ENVIRONMENT)
              (NULL (CL:SETQ CL::FN-DEFN (CL:GETF (ENVIRONMENT-FUNCTIONS CL::ENVIRONMENT)
                                                    CL::FN))))
          (CL:SYMBOL-FUNCTION CL::FN))
         ((EQ (CAR CL::FN-DEFN)
              :FUNCTION)
          (CDR CL::FN-DEFN))
         (T (CL:ERROR "The lexical macro ~S is not a legal argument to ~S." CL::FN 'CL:FUNCTION))))))
    ((OR (NULL CL::ENVIRONMENT)
         (AND (FOR CL::VALUE IN (CDR (ENVIRONMENT-VARS CL::ENVIRONMENT)) BY CDDR
              ALWAYS (EQ CL::VALUE *SPECIAL-BINDING-MARK*))
              (NULL (ENVIRONMENT-FUNCTIONS CL::ENVIRONMENT))
              (NULL (ENVIRONMENT-BLOCKS CL::ENVIRONMENT))
              (NULL (ENVIRONMENT-TAGBODIES CL::ENVIRONMENT))))
     ;; Environment is empty: don't have to make a closure.
     CL::FN)
    (T (MAKE-CLOSURE :FUNCTION (COND
      ((EQ (CAR CL::FN)
           'LAMBDA)
       (CL:LAMBDA (&OPTIONAL ,@(CADR CL::FN)
                    &REST IGNORE)
        ,@(CDDR CL::FN)))
      (T CL::FN))
      :ENVIRONMENT
      (\COPY-ENVIRONMENT CL::ENVIRONMENT) ; environment is copied so that forms that side-effect it (such as
                                           ; LET*) will work correctly.
    ))))
)

```

```

(DEFINE-SPECIAL-FORM FUNCTION (FN &OPTIONAL FUNARGP &ENVIRONMENT ENVIRONMENT)
  ;; Interlisp FUNCTION in Common Lisp interpreter:
  ;; like CL:FUNCTION except that (FUNCTION FOO) just returns FOO and not its definition.
  (COND
    (FUNARGP (CL:FUNCALL #'FUNCTION FN FUNARGP))

```

```

((CL:SYMBOLP FN)
 (LET (FN-DEFN)
  (COND
   ((OR (NULL ENVIRONMENT)
        (NULL (SETQ FN-DEFN (CL:GETF (ENVIRONMENT-FUNCTIONS ENVIRONMENT)
                                       FN))))
    FN)
   ((EQ (CAR FN-DEFN)
        :FUNCTION)
    (CDR FN-DEFN))
   (T (CL:ERROR "The lexical macro ~S is not a legal argument to ~S." FN 'FUNCTION)))))
((OR (NULL ENVIRONMENT)
  (AND (FOR VALUE IN (CDR (ENVIRONMENT-VARS ENVIRONMENT)) BY CDDR ALWAYS (EQ VALUE
                                                                              *SPECIAL-BINDING-MARK*))
        )
  (NULL (ENVIRONMENT-FUNCTIONS ENVIRONMENT))
  (NULL (ENVIRONMENT-BLOCKS ENVIRONMENT))
  (NULL (ENVIRONMENT-TAGBODIES ENVIRONMENT))))
FN)
(T (MAKE-CLOSURE :FUNCTION (COND
  ((EQ (CAR FN)
        'LAMBDA)
   (CL:LAMBDA (&OPTIONAL ,@(CADR FN)
                &REST IGNORE)
    ,@(CDDR FN)))
  (T FN))
  :ENVIRONMENT ENVIRONMENT))))

(CL:DEFUN CL:FUNCTIONP (CL::FN)
 (AND (OR (CL:SYMBOLP CL::FN)
          (CL:COMPILED-FUNCTION-P CL::FN)
          (AND (CL:CONSP CL::FN)
                (EQ (CAR CL::FN)
                    'CL:LAMBDA))
          (CLOSURE-P CL::FN))
  T))

(CL:DEFUN CL:COMPILED-FUNCTION-P (CL::FN)
 (OR (TYPEP CL::FN 'COMPILED-CLOSURE)
  (AND (ARRAYP CL::FN)
        (EQ (|fetch| (ARRAYP TYP) |of| CL::FN)
            \\ST.CODE))))

(DEFINE-SPECIAL-FORM CL:MULTIPLE-VALUE-CALL (CL::FN &REST CL::ARGS &ENVIRONMENT CL::ENV)
 ;; for interpreted calls only. The macro inserts a MVLIST call after the computation of TAIL
 (CL:APPLY (CL:EVAL CL::FN CL::ENV)
  (FOR CL::X IN CL::ARGS JOIN (\\MVLIST (CL:EVAL CL::X CL::ENV)))))

(DEFINE-SPECIAL-FORM CL:MULTIPLE-VALUE-PROG1 (CL::FORM &REST CL::OTHER-FORMS &ENVIRONMENT CL::ENV)
 (CL:VALUES-LIST (PROG1 (CL:MULTIPLE-VALUE-LIST (CL:EVAL CL::FORM CL::ENV))
  (FOR CL::X IN CL::OTHER-FORMS DO (CL:EVAL CL::X CL::ENV)))))

(DEFINEQ
 (COMP.CL-EVAL
  (LAMBDA (EXP)
    (COMP.SPREAD (CDR ,@EXP)
      '*EVAL-ARGUMENT-COUNT*
      (CAR ,@EXP)
      '((CL:EVAL ENVIRONMENT)))))
 )
 (* |lmm| " 5-Jun-86 00:44")

(CL:DEFUN CL:EVALHOOK (CL::FORM CL::EVALHOOKFN CL::APPLYHOOKFN &OPTIONAL CL::ENV)
 "Evaluates Form with *Evalhook* bound to Evalhookfn and *Applyhook* bound to applyhookfn. Ignores these
 hooks once, for the top-level evaluation of Form."
 (LET ((*EVALHOOK* CL::EVALHOOKFN)
      (CL::*SKIP-EVALHOOK* T)
      (*APPLYHOOK* CL::APPLYHOOKFN)
      (CL::*SKIP-APPLYHOOK* NIL))
  (CL:EVAL CL::FORM CL::ENV)))

(CL:DEFUN CL:APPLYHOOK (CL:FUNCTION CL::ARGS CL::EVALHOOKFN CL::APPLYHOOKFN &OPTIONAL CL::ENV)
 "Evaluates Form with *Evalhook* bound to Evalhookfn and *Applyhook* bound to applyhookfn. Ignores these
 hooks once, for the top-level evaluation of Form."
 (DECLARE (IGNORE CL::ENV))
 ;; the env argument is not used as agreed on the Common Lisp mailing list. (Arguments have already been evaluated.)
 (LET ((*EVALHOOK* CL::EVALHOOKFN)
      (CL::*SKIP-EVALHOOK* T)

```

```

(*APPLYHOOK* CL::APPLYHOOKFN)
(CL::*SKIP-APPLYHOOK* NIL))
(CL:APPLY CL:FUNCTION CL::ARGS)))

```

```
(CL:DEFVAR *EVALHOOK* NIL)
```

```
(CL:DEFVAR *APPLYHOOK* NIL)
```

```
(CL:DEFVAR CL::*SKIP-EVALHOOK* NIL
"Used with non-null *EVALHOOK* to suppress the use of the hook-function
for one level of eval.")
```

```
(CL:DEFVAR CL::*SKIP-APPLYHOOK* NIL
"Used with non-null *APPLYHOOK* to suppress the use of the hook function
for one level of eval.")
```

```
:: CONSTANTS mechanism
```

```
(DEFINEQ
```

```
(CL:CONSTANTP
```

```
(* |vanMelle| "19-Nov-86 21:43")
```

```

(LAMBDA (OBJECT ENVIRONMENT)
  (CL:TYPECASE OBJECT
    (CL:NUMBER T)
    (CL:CHARACTER T)
    (STRING T)
    (CL:BIT-VECTOR T)
    (CL:SYMBOL (OR (EQ OBJECT NIL)
                    (EQ OBJECT T)
                    (CL:KEYWORDP OBJECT)
                    (AND COMPVARMACROHASH (SETQ OBJECT (GETHASH OBJECT COMPVARMACROHASH))
                        (CL:CONSTANTP OBJECT)))))
    (CONS (CASE (CAR OBJECT)
              ((CONSTANT QUOTE) T)
              (CL:OTHERWISE (COND
                             ((FMEMB (CAR OBJECT)
                                       CONSTANTFOLDFNS)
                              (EVERY (CDR OBJECT)
                                       (FUNCTION CL:CONSTANTP)))
                             (T (CL:MULTIPLE-VALUE-BIND (NEW-FORM EXPANDED)
                                                           (CL:MACROEXPAND OBJECT ENVIRONMENT)
                                                           (AND EXPANDED (CL:CONSTANTP NEW-FORM))))))))))

```

```
)
```

```
(CL:DEFSETF CL:CONSTANTP XCL::SET-CONSTANTP)
```

```
(CL:DEFUN XCL::SET-CONSTANTP (CL:SYMBOL XCL::FLAG)
  (CL:IF (NOT (NULL XCL::FLAG))
    (CL:SETF (GETHASH CL:SYMBOL COMPVARMACROHASH)
              `(CONSTANT ,CL:SYMBOL))
    (CL:WHEN (TYPEP COMPVARMACROHASH 'CL:HASH-TABLE)
              (REMHASH CL:SYMBOL COMPVARMACROHASH))))
```

```
:: Interlisp SETQ for Common Lisp and vice versa
```

```
(DEFINE-SPECIAL-FORM CL:SETQ (&REST CL::TAIL &ENVIRONMENT CL::ENV)
  (LET (CL::VALUE)
    (WHILE CL::TAIL DO (CL:SETQ CL:VALUE (SET-SYMBOL (CL:POP CL::TAIL)
                                                         (CL:EVAL (CL:POP CL::TAIL)
                                                         CL::ENV)
                                                         CL::ENV)))
    CL::VALUE))
```

```
(DEFINE-SPECIAL-FORM SETQ (VAR VALUE &ENVIRONMENT ENV)
  (SET-SYMBOL VAR (CL:EVAL VALUE ENV)
    ENV))
```

```
(PUTPROPS CL:SETQ DMACRO (DEFMACRO (X Y &REST CL:REST) `(PROGN (SETQ ,X ,Y)
                                                                    ,@ (AND CL:REST `((CL:SETQ ,@CL:REST))))))
```

```
:: An lambda definition for cl:setq so cmldefer may use cl:setq will run in the init
```

```
(DEFINEQ
```

```
(CL:SETQ
```

```

(NLAMBDA CL::TAIL
  (LET ((CL::VALUE NIL))

```

```
; Edited 15-Nov-87 17:34 by jop
```

```

    (CL:LOOP (CL:IF (NULL CL::TAIL)
                    (RETURN CL::VALUE))
      (CL:SETQ CL::VALUE (SET (CL:POP CL::TAIL)
                              (CL:IF (NOT (BOUNDP *EVALHOOK*))
                                      (PROGN ;; CMLEVAL Init-forms not yet run
                                           (EVAL (CL:POP CL::TAIL)))
                                      (CL:EVAL (CL:POP CL::TAIL)))))))
  )

```

```

(DEFMACRO SETQ (VAR &REST VALUE-FORMS)
  (COND
    ((NULL VALUE-FORMS)
     `(CL:SETQ ,VAR NIL))
    ((NULL (CDR VALUE-FORMS))
     `(CL:SETQ ,VAR , (CAR VALUE-FORMS)))
    (T `(CL:SETQ ,VAR (PROG1 ,@VALUE-FORMS))))

```

```
(DEFINEQ
```

```
(SET-SYMBOL
```

```
(LAMBDA (CL:SYMBOL VALUE ENVIRONMENT)
```

```
; Edited 7-Jan-87 15:37 by gbn
```

```
(CL:BLOCK SET-SYMBOL
```

```
(|if| ENVIRONMENT
```

```
|then| (SETQ ENVIRONMENT (ENVIRONMENT-VARS ENVIRONMENT))
```

```
(WHILE ENVIRONMENT DO (IF (EQ CL:SYMBOL (CAR ENVIRONMENT))
```

```
THEN ;; found a binding for this symbol
```

```
(IF (EQ (CAR (SETQ ENVIRONMENT (CDR ENVIRONMENT)))
      *SPECIAL-BINDING-MARK*)
```

```
THEN ;; it is a special binding, or a mark that we are using the special value
```

```
(RETURN NIL)
```

```
; return from WHILE
```

```
)
```

```
(RPLACA ENVIRONMENT VALUE)
```

```
;; smash new value in
```

```
(CL:RETURN-FROM SET-SYMBOL VALUE)
```

```
ELSE (SETQ ENVIRONMENT (CDDR ENVIRONMENT))))))
```

```
;; no environment, or not found
```

```
(SETQ ENVIRONMENT (\\STKSCAN CL:SYMBOL))
```

```
(COND
```

```
((EQ (\\HILOC ENVIRONMENT)
```

```
\\STACKHI)
```

```
(\\PUTBASEPTR ENVIRONMENT 0 VALUE))
```

```
(T (\\RPLPTR ENVIRONMENT 0 VALUE)))
```

```
VALUE)))
```

```
)
```

```
(DEFMACRO CL:PSETQ (&REST TAIL)
```

```
(AND TAIL `(PROGN (SETQ ,(|pop| TAIL)
```

```
, (CL:IF (CDR TAIL)
```

```
`(PROG1 , (POP TAIL)
```

```
(CL:PSETQ ,@TAIL))
```

```
(CAR TAIL)))
```

```
NIL)))
```

```
(DEFMACRO SETQQ (SYMBOL VALUE)
```

```
; so common lisp interpreter will know about it
```

```
`(SETQ ,SYMBOL ',VALUE))
```

```
(DEFINE-SPECIAL-FORM CL:CATCH (CL::CATCH-TAG &REST CL::\\CATCH-FORMS &ENVIRONMENT CL::\\CATCH-ENV)
```

```
(CL:CATCH (CL:EVAL CL::CATCH-TAG CL::\\CATCH-ENV)
```

```
(\\EVAL-PROGN CL::\\CATCH-FORMS CL::\\CATCH-ENV)))
```

```
(DEFINE-SPECIAL-FORM CL:THROW (CL::TAG CL::VALUE &ENVIRONMENT CL::ENV)
```

```
(CL:THROW (CL:EVAL CL::TAG CL::ENV)
```

```
(CL:EVAL CL::VALUE CL::ENV)))
```

```
(DEFINE-SPECIAL-FORM CL:UNWIND-PROTECT (CL::\\FORM &REST CL::\\CLEANUPS &ENVIRONMENT CL::\\ENV)
```

```
(CL:UNWIND-PROTECT
```

```
(CL:EVAL CL::\\FORM CL::\\ENV)
```

```
(\\EVAL-PROGN CL::\\CLEANUPS CL::\\ENV)))
```

```
(DEFINEQ
```

```
(CL:THROW
```

```
(NLAMBDA (THROW-TAG THROW-VALUE)
```

```
(DECLARE (LOCALVARS . T))
(CL:THROW (\\EVAL THROW-TAG)
  (\\EVAL THROW-VALUE)))
```

(* |Imm| "30-May-86 00:09")

(CL:CATCH

```
(NLAMBDA \\CATCH-FORMS
  (CL:CATCH (\\EVAL (CAR \\CATCH-FORMS))
    (\\EVPROGN (CDR \\CATCH-FORMS)))))
```

; Edited 7-Apr-88 16:53 by amd

(CL:UNWIND-PROTECT

```
(NLAMBDA \\UNWIND-FORMS
  (CL:UNWIND-PROTECT
    (\\EVAL (CAR \\UNWIND-FORMS))
    (\\EVPROGN (CDR \\UNWIND-FORMS)))))
```

; Edited 7-Apr-88 16:54 by amd

)

(DEFMACRO PROG

```
(VARS &BODY (BODY DECLS))
  (CL:BLOCK NIL
    (LET ,VARS ,@DECLS (CL:TAGBODY ,@BODY))))
```

(DEFMACRO PROG*

```
(VARS &BODY (BODY DECLS))
  (CL:BLOCK NIL
    (LET* ,VARS ,@DECLS (CL:TAGBODY ,@BODY))))
```

```
(DEFINE-SPECIAL-FORM GO (CL::\\TAG &ENVIRONMENT CL::ENV)
```

```
(BIND CL::TAIL FOR CL::TAGBODIES ON (AND CL::ENV (ENVIRONMENT-TAGBODIES CL::ENV)) BY CDDR
  WHEN (CL:SETQ CL::TAIL (CL:MEMBER CL::\\TAG (CAR CL::TAGBODIES)))
```

;; MUST use EQL, as tags may be integers.

```
DO (HANDLER-BIND ((ILLEGAL-THROW #'(CL:LAMBDA (CL::C)
  (CL:ERROR 'ILLEGAL-GO :TAG CL::\\TAG))))
  (CL:THROW (CADR CL::TAGBODIES)
    CL::TAIL))
  FINALLY (CL:ERROR 'ILLEGAL-GO :TAG CL::\\TAG)))
```

```
(DEFINE-SPECIAL-FORM CL:TAGBODY (&REST CL::\\TAGBODY-TAIL &ENVIRONMENT CL::ENV)
```

```
(LET* ((CL::BLIP (CONS NIL NIL))
  (CL::\\NEW-ENV (\\MAKE-CHILD-ENVIRONMENT CL::ENV :TAGBODY (CL::\\TAGBODY-TAIL CL::BLIP))))
  (WHILE (CL:SETQ CL::\\TAGBODY-TAIL (CL:CATCH CL::BLIP
    (FOR CL::X IN CL::\\TAGBODY-TAIL UNLESS (CL:SYMBOLP CL::X)
      DO (CL:EVAL CL::X CL::\\NEW-ENV)))))
```

(DEFINEQ

(CL:TAGBODY

```
(NLAMBDA TAIL
```

(* |Imm| "23-May-86 16:05")

(* |like| PROG |with| |no| |variables|)

```
(LET ((TL (CONS NIL TAIL)))
  (\\PROG0 TL TL)))
```

)

;; for macro caching

(DEFINEQ

(CACHEMACRO

```
(LAMBDA (FN BODY ENV)
```

; Edited 25-Sep-87 18:32 by jop

;;; We want to cache the expansion unless

;; 1) the env is not an interpreted env (including NIL),

;; 2) there are lexical macros in force, OR

;; 3) There is a compiler-let in force.

```
(CL:IF (OR (NOT (TYPEP ENV 'ENVIRONMENT))
  (AND ENV (FOR FN IN (CDR (ENVIRONMENT-FUNCTIONS ENV)) BY CDDR THEREIS (EQ (CAR FN)
    :MACRO))))
```

IN-COMPILER-LET)

(CL:FUNCALL FN BODY ENV)

(OR (GETHASH BODY CLISPARRAY)

```
(PUTHASH BODY (CL:FUNCALL FN BODY ENV)
  CLISPARRAY))))
```

)

(CL:DEFPARAMETER *MACROEXPAND-HOOK* 'CACHEMACRO)

(RPAQQ *IN-COMPILER-LET* NIL)

:: PROCLAIM and friends.

:: Needs to come first because DEFVARs put it out. With package code in the init, also need this here rather than CMLEVAL

```
(CL:DEFUN CL:PROCLAIM (CL::PROCLAMATION)
  ;; PROCLAIM is a top-level form used to pass assorted information to the compiler. This interpreter ignores proclamations except for those declaring
  ;; variables to be SPECIAL.
  (CL:WHEN (CL:CONSP CL::PROCLAMATION)
    (CASE (CAR CL::PROCLAMATION)
      (CL:SPECIAL (FOR CL::X IN (CDR CL::PROCLAMATION) DO (CL:SETF (VARIABLE-GLOBALLY-SPECIAL-P CL::X)
                                                                    T)
                                                                    (CL:SETF (VARIABLE-GLOBAL-P CL::X)
                                                                    NIL)
                                                                    (CL:SETF (CL:CONSTANTP CL::X)
                                                                    NIL)))
      (GLOBAL (FOR CL::X IN (CDR CL::PROCLAMATION) DO (CL:SETF (VARIABLE-GLOBAL-P CL::X)
                                                                T)
                                                                (CL:SETF (VARIABLE-GLOBALLY-SPECIAL-P CL::X)
                                                                NIL)
                                                                (CL:SETF (CL:CONSTANTP CL::X)
                                                                NIL)))
      (SI::CONSTANT (FOR CL::X IN (CDR CL::PROCLAMATION) DO (CL:SETF (CL:CONSTANTP CL::X)
                                                                    T)
                                                                    (CL:SETF (VARIABLE-GLOBAL-P CL::X)
                                                                    NIL)
                                                                    (CL:SETF (VARIABLE-GLOBALLY-SPECIAL-P CL::X)
                                                                    NIL)))
      (CL:DECLARATION (FOR CL::X IN (CDR CL::PROCLAMATION) DO (CL:SETF (XCL::DECL-SPECIFIER-P CL::X)
                                                                    T)))
      (CL:NOTINLINE (FOR CL::X IN (CDR CL::PROCLAMATION) DO (CL:SETF (XCL::GLOBALLY-NOTINLINE-P CL::X)
                                                                    T)))
      (CL:INLINE (FOR CL::X IN (CDR CL::PROCLAMATION) DO (CL:SETF (XCL::GLOBALLY-NOTINLINE-P CL::X)
                                                                    NIL))))))

;; used by the codewalker, too

(DECLARE\ : EVAL@COMPILE

(PUTPROPS VARIABLE-GLOBALLY-SPECIAL-P MACRO ((VARIABLE)
                                              (GET VARIABLE 'GLOBALLY-SPECIAL)))

(PUTPROPS VARIABLE-GLOBAL-P MACRO ((VARIABLE)
                                     (GET VARIABLE 'GLOBALVAR)))
)

(CL:DEFUN XCL::DECL-SPECIFIER-P (CL:SYMBOL)
  (GET CL:SYMBOL 'SI::DECLARATION-SPECIFIER))

(CL:DEFUN XCL::SET-DECL-SPECIFIER-P (XCL::SPEC XCL::VAL)
  (CL:SETF (GET XCL::SPEC 'SI::DECLARATION-SPECIFIER)
    XCL::VAL))

(CL:DEFUN XCL::GLOBALLY-NOTINLINE-P (XCL::FN)
  (GET XCL::FN 'SI::GLOBALLY-NOTINLINE))

(CL:DEFUN XCL::SET-GLOBALLY-NOTINLINE-P (XCL::FN XCL::VAL)
  (CL:SETF (GET XCL::FN 'SI::GLOBALLY-NOTINLINE)
    XCL::VAL))

(CL:DEFSETF XCL::DECL-SPECIFIER-P XCL::SET-DECL-SPECIFIER-P)

(CL:DEFSETF XCL::GLOBALLY-NOTINLINE-P XCL::SET-GLOBALLY-NOTINLINE-P)

(PUTPROPS GLOBALLY-SPECIAL PROPTYPE IGNORE)

(PUTPROPS GLOBALVAR PROPTYPE IGNORE)

(PUTPROPS SI::DECLARATION-SPECIFIER PROPTYPE IGNORE)

(PUTPROPS SI::GLOBALLY-NOTINLINE PROPTYPE IGNORE)

(PUTPROPS SPECIAL-FORM PROPTYPE IGNORE)

(PUTPROPS CMLEVAL FILETYPE BCOMPL)

(PUTPROPS CMLEVAL MAKEFILE-ENVIRONMENT (:READTABLE "XCL" :PACKAGE "INTERLISP"))

(DECLARE\ : EVAL@COMPILE DONTCOPY
```

```

(DEFOPTIMIZER CL-EVAL-FN3-CALL (ARG1 ARG2 &ENVIRONMENT ENV)
  ;; Emit a call to FN3 after pushing only 2 arguments (the other having been pushed by
  ;; IL::COMPILER-SPREAD-ARGUMENTS. earlier in the game). Used in CL:EVAL.
  (COND
    ((FMEMB :4-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE ENV))
      `((OPCODES FN3 0 0 0 (FN . \\EVAL-INVOKE-LAMBDA)
          RETURN)
        ,ARG1
        ,ARG2))
    ((FMEMB :3-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE ENV))
      `((OPCODES FN3 0 0 (FN . \\EVAL-INVOKE-LAMBDA)
          RETURN)
        ,ARG1
        ,ARG2)))
    (T `((OPCODES FN3 0 (FN . \\EVAL-INVOKE-LAMBDA)
          RETURN)
        ,ARG1
        ,ARG2))))
)

(DECLARE\ : DONTVAL@LOAD DOEVAL@COMPILE DONTCOPY

(DECLARE\ : DOEVAL@COMPILE DONTCOPY

(LOCALVARS . T)
)
)

(DECLARE\ : DONTVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILERVERS

(ADDTOVAR NLAMA CL:TAGBODY CL:UNWIND-PROTECT CL:CATCH CL:SETQ CL:BLOCK CL:EVAL-WHEN CL:COMPILER-LET COMMON-LISP
)

(ADDTOVAR NLAML CL:THROW CL:FUNCTION CL:RETURN-FROM CL:IF)

(ADDTOVAR LAMA CL:APPLY CL:FUNCALL)
)

```

FUNCTION INDEX

CL:APPLY	11	CL:EQUAL	2	SET-SYMBOL	20
CL:APPLYHOOK	18	CL:EQUALP	2	CL:SETQ	19
CL:BLOCK	16	CL:EVAL	6	CL:SPECIAL-FORM-P	4
CACHEMACRO	21	CL:EVAL-WHEN	13	CL:TAGBODY	21
CL:CATCH	21	CL:EVALHOOK	18	CL:THROW	20
CHECK-BINDABLE	10	CL:FUNCALL	11	CL:UNWIND-PROTECT	21
CHECK-KEYWORDS	10	CL:FUNCTION	17	\\EVAL-VOKE-LAMBDA	7
COMMON-LISP	4	CL:FUNCTIONP	18	\\EVAL-PROGN	13
COMP.CL-EVAL	18	XCL::GLOBALLY-NOTINLINE-P	22	\\INTERPRET-ARGUMENTS	8
COMP.COMPILER-LET	11	CL:IF	15	\\INTERPRETER-LAMBDA	10
CL:COMPILED-FUNCTION-P	18	CL:PROCLAIM	22	\\LET*-RECURSION	14
CL:COMPILER-LET	11	CL:RETURN-FROM	16	\\LETtran 	14
CL:CONSTANTP	19	XCL::SET-CONSTANTP	19	\\REMOVE-DECLS	3
XCL::DECL-SPECIFIER-P	22	XCL::SET-DECL-SPECIFIER-P	22	\\TRANSLATE-CL\\:LAMBDA	4
DECLARED-SPECIAL	10	XCL::SET-GLOBALLY-NOTINLINE-P	22		

MACRO INDEX

.COMPILER-SPREAD-ARGUMENTS.	10	CL:FUNCALL	11	CL:PSETQ	20
AND	16	CL:FUNCTION	17	RETURN	16
CL:APPLY	11	CL:IF	15	CL:SETQ	19
ARG-REF	10	INTERLISP	4	SETQ	20
CL:BLOCK	16	LET	14	SETQQ	20
COMMON-LISP	4	LET*	14	THE	13
CL:COMPILER-LET	11	CL:LOCALLY	13	VARIABLE-GLOBAL-P	22
COND	15	OR	16	VARIABLE-GLOBALLY-SPECIAL-P	22
CL:EQUAL	3	PROG	21	\\MAKE-CHILD-ENVIRONMENT	6
CL:EQUALP	3	PROG*	21		
CL:EVAL-WHEN	13	PROG1	13		

SPECIAL-FORM INDEX

AND	16	CL:FUNCTION	17	CL:MACROLET	12	CL:SETQ	19
CL:BLOCK	16	FUNCTION	17	CL:MULTIPLE-VALUE-CALL	18	SETQ	19
CL:CATCH	20	GO	21	CL:MULTIPLE-VALUE-PROG1	18	CL:TAGBODY	21
CL:COMPILER-LET	12	CL:IF	15	OR	16	THE	12
COND	15	INTERLISP	4	PROG1	13	CL:THROW	20
DECLARE	13	CL:LABELS	12	PROGN	13	CL:UNWIND-PROTECT	20
CL:EVAL-WHEN	13	LET	14	QUOTE	12		
CL:FLET	12	LET*	14	CL:RETURN-FROM	17		

VARIABLE INDEX

APPLYHOOK	19	*IN-COMPILER-LET*	21	CL::*SKIP-EVALHOOK*	19
CHECK-ARGUMENT-COUNTS	5	*MACROEXPAND-HOOK*	21	*SPECIAL-BINDING-MARK*	5
EVALHOOK	19	CL::*SKIP-APPLYHOOK*	19	LAMBDA\$PLST	4

PROPERTY INDEX

CMLEVAL	22	SI::GLOBALLY-NOTINLINE	22	SPECIAL-FORM	22
SI::DECLARATION-SPECIFIER	22	GLOBALLY-SPECIAL	22		
CL:EQUAL	3	GLOBALVAR	22		

SETF INDEX

CL:CONSTANTP	19	XCL::DECL-SPECIFIER-P	22	XCL::GLOBALLY-NOTINLINE-P	22
--------------------	----	-----------------------------	----	---------------------------------	----

CONSTANT INDEX

CL:CALL-ARGUMENTS-LIMIT	5	CL:LAMBDA-LIST-KEYWORDS	5	CL:LAMBDA-PARAMETERS-LIMIT	5
-------------------------------	---	-------------------------------	---	----------------------------------	---

STRUCTURE INDEX

CLOSURE	5	ENVIRONMENT	6
---------------	---	-------------------	---

{MEDLEY}<sources>CMLEVAL.;1

OPTIMIZER INDEX

CL-EVAL-FN3-CALL23
