File created: 17-Jul-90 10:48:08 **{DSK}<users>sybalsky>XCLC-ENV-CTXT.;1**

changes to: (IL:STRUCTURES ENV)
(IL:VARS IL:XCLC-ENV-CTXTCOMS)
(IL:VARIABLES *HOST-ARCHITECTURE* *TARGET-ARCHITECTURE*)

previous date: 23-May-90 13:00:38 {PELE:MV:ENVOS}<LISPCORE>SOURCES>XCLC-ENV-CTXT.;2

Read Table: XCL

Package: COMPILER

Format: XCCS


```
(IL:RPAQQ IL:XCLC-ENV-CTXTCOMS
          (
```

;;; Contexts and Environments

```
           (IL:STRUCTURES CONTEXT ENV)
           (IL:FUNCTIONS LOCAL-CONSTANT-P PRINT-CONTEXT PRINT-ENV)
           (IL:FUNCTIONS MAKE-CHILD-ENV ENV-BIND-VARIABLE ENV-BIND-FUNCTION ENV-ADD-DECLS ENV-DECL-P
                  ENV-ALLOW-INLINES ENV-DISALLOW-INLINES ENV-INLINE-ALLOWED ENV-INLINE-DISALLOWED
                  ENV-PROCLAIM-SPECIAL ENV-PROCLAIMED-SPECIAL-P ENV-PROCLAIM-GLOBAL ENV-PROCLAIMED-GLOBAL-P
                  ENV-DECLARE-SPECIALS ENV-DECLARE-GLOBALS ENV-DECLARE-A-SPECIAL ENV-DECLARE-A-GLOBAL)
           (IL:FUNCTIONS FIND-TOP-ENVIRONMENT RESOLVE-VARIABLE-REFERENCE RESOLVE-VARIABLE-BINDING
                  VALUE-FOLDABLE-P CHECK-GLOBAL-CONSTANT CONSTANT-VALUE SET-CONSTANT-VALUE LOCAL-CONSTANT-P)
           (IL:SETFS CONSTANT-VALUE)
           (IL:VARIABLES *CONSTANTS-HASH-TABLE*)
           (IL:VARIABLES *ENVIRONMENT* *CONTEXT* *ARGUMENT-CONTEXT* *EFFECT-CONTEXT* *NULL-CONTEXT*
                  *PREDICATE-CONTEXT*)
           ;; External interface to environments
           (IL:FUNCTIONS ENV-BOUNDP ENV-FBOUNDP COPY-ENV-WITH-FUNCTION COPY-ENV-WITH-VARIABLE MAKE-EMPTY-ENV)
           ;; Describe the machine we're running on and the target for hte compiled code
           (IL:VARIABLES *HOST-ARCHITECTURE* *TARGET-ARCHITECTURE*)
           ;; Arrange for the correct compiler to be used.
           (IL:PROP IL:FILETYPE IL:XCLC-ENV-CTXT)
           ;; Arrange for the correct reader environment.
           (IL:PROP IL:MAKEFILE-ENVIRONMENT IL:XCLC-ENV-CTXT)))
```

;;; Contexts and Environments

```
(DEFSTRUCT (CONTEXT (:PRINT-FUNCTION PRINT-CONTEXT)
                    (:INLINE T))
```

;;; TOP-LEVEL-P is non-nil iff we are analyzing a top-level form.

;;; VALUES-USED is either :unknown or the non-negative number of values expected for the result of the current form.

;;; PREDICATE-P is non-nil iff the value of the current form will only be used in a nil/non-nil test.

;;; APPLIED-CONTEXT is either nil or the context in which the result of applying the current form as a function would be used. If NIL, then assume the
;;; *null-context*.

```
   (TOP-LEVEL-P NIL)
   (VALUES-USED :UNKNOWN)
   (PREDICATE-P NIL)
   (APPLIED-CONTEXT NIL))
```

```
(DEFSTRUCT (ENV (:PRINT-FUNCTION PRINT-ENV)
                (:INLINE T))
```

;;; Structure for maintaining the compiler's idea of the environment of a given form.

;;; PARENT is either NIL, meaning that this environment is at the top, or another ENV structure.

;;; VENV is an AList associating the symbol that is the name of the variable with a list of two elements, the SCOPE of the variable (one of :CONSTANT
;;; or :LEXICAL) and the VARIABLE or LITERAL structure corresponding to the variable.

;;; FENV is an AList associating the symbol that is the name of the function with a list of two elements: the KIND of the symbol (either :MACRO or
;;; :FUNCTION) and either an expansion function (iff KIND = :MACRO) or a VARIABLE structure (iff KIND = :FUNCTION). When KIND = :FUNCTION,
;;; the VARIABLE structure may be omitted.

;;; The information about INLINE and NOTINLINE declarations is maintained by two lists, ALLOWED-INLINES and DISALLOWED-INLINES. If a symbol
;;; is on DISALLOWED-INLINES, the compiler will not expand calls to it inline; nor will any optimizers on that symbol be applied. If a symbol is on
;;; ALLOWED-INLINES, the compiler will try harder to expand calls to the symbol inline. A given symbol should not appear on both lists.

;;; DECL-SPECIFIERS is a list of non-standard declaration-specifiers to be allowed in the compilation. Such specifiers are created by (declare
;;; (declaration foo)) forms.

```
;;; DECLARED-SPECIALS is a list of those symbols that were DECLAREd SPECIAL at this contour.  In the top-most environment, the list is used for
;;; PROCLAIMed variables.

;;; DECLARED-GLOBALS is just like DECLARED-SPECIALS but for globals.

    ;; TARGET-ARCHITECTURE is a list analogous to *FEATURES* that describes the architecture of the target machine we're compiling to.  This is
    ;; intended initially to support multiple instruction sets (for the 3-byte-symbol change).
    (PARENT NIL)
    (VENV NIL)
    (FENV NIL)
    (ALLOWED-INLINES NIL)
    (DISALLOWED-INLINES NIL)
    (DECL-SPECIFIERS NIL)
    (DECLARED-SPECIALS NIL)
    (DECLARED-GLOBALS NIL)
    (TARGET-ARCHITECTURE *TARGET-ARCHITECTURE*))


(DEFUN LOCAL-CONSTANT-P (SYMBOL)
   (GETHASH SYMBOL *CONSTANTS-HASH-TABLE*))


(DEFUN PRINT-CONTEXT (STRUCT STREAM DEPTH)

;;; Print almost all contexts in a more readable form, interpreting the various combinations of the fields.

   (DECLARE (IGNORE DEPTH))
   (LET ((TL (CONTEXT-TOP-LEVEL-P STRUCT))
         (VALS (CONTEXT-VALUES-USED STRUCT))
         (PRED (CONTEXT-PREDICATE-P STRUCT))
         (APP (CONTEXT-APPLIED-CONTEXT STRUCT)))
        (MACROLET ((OUTPUT (STRING &REST ARGS)
                           `(FORMAT STREAM ,(CONCATENATE 'STRING "~:[" STRING "~;#<Ctxt--" STRING ">~]")
                                    *PRINT-ESCAPE*
                                    ,@ARGS)))
                  (LET ((VALUE (COND
                                   ((AND (EQ TL NIL)
                                         (EQ VALS :UNKNOWN)
                                         (EQ PRED NIL)
                                         (EQ APP NIL))
                                    (OUTPUT "Null"))
                                   ((AND (EQ TL T)
                                         (EQ VALS 0)
                                         (EQ PRED NIL)
                                         (EQ APP NIL))
                                    (OUTPUT "Top-level form"))
                                   ((AND (EQ TL NIL)
                                         (EQ VALS 0)
                                         (EQ PRED NIL)
                                         (EQ APP NIL))
                                    (OUTPUT "Effect"))
                                   ((AND (EQ TL NIL)
                                         (EQ VALS 1)
                                         (EQ PRED NIL)
                                         (EQ APP NIL))
                                    (OUTPUT "Argument"))
                                   ((AND (EQ TL NIL)
                                         (EQ VALS 1)
                                         (EQ PRED T)
                                         (EQ APP NIL))
                                    (OUTPUT "Predicate"))
                                   ((AND (EQ TL NIL)
                                         (EQ PRED NIL)
                                         (EQ APP NIL))
                                    (OUTPUT "~S values" VALS))
                                   (T (FORMAT STREAM "#<Ctxt--TL: ~S--Vals: ~S--Pred: ~S--App: ~A>" TL VALS PRED APP
                                              ))))))))


(DEFUN PRINT-ENV (STRUCT STREAM DEPTH)
   (DECLARE (IGNORE DEPTH))
   (FORMAT STREAM "#<Compiler Environment @ ~O,~O>" (IL:\\HILOC STRUCT)
          (IL:\\LOLOC STRUCT)))


(DEFUN MAKE-CHILD-ENV (PARENT)
   (MAKE-ENV :PARENT PARENT))


(DEFUN ENV-BIND-VARIABLE (ENV NAME STRUCT)
   (PUSH (CONS NAME STRUCT)
         (ENV-VENV ENV)))


(DEFUN ENV-BIND-FUNCTION (ENV NAME KIND &OPTIONAL EXPN-OR-VAR)
   (PUSH (LIST NAME KIND EXPN-OR-VAR)
```

```
         (ENV-FENV ENV)))

(DEFUN ENV-ADD-DECLS (ENV SPECIFIERS)
   (SETF (ENV-DECL-SPECIFIERS ENV)
         (APPEND SPECIFIERS (ENV-DECL-SPECIFIERS ENV))))

(DEFUN ENV-DECL-P (ENV SPECIFIER)
   (OR (MEMBER SPECIFIER (ENV-DECL-SPECIFIERS ENV))
       (LET ((PARENT (ENV-PARENT ENV)))
            (AND (ENV-P PARENT)
                 (ENV-DECL-P PARENT SPECIFIER)))))

(DEFUN ENV-ALLOW-INLINES (ENV NAMES)
   (SETF (ENV-DISALLOWED-INLINES ENV)
         (SET-DIFFERENCE (ENV-DISALLOWED-INLINES ENV)
                NAMES))
   (SETF (ENV-ALLOWED-INLINES ENV)
         (UNION (ENV-ALLOWED-INLINES ENV)
                NAMES)))

(DEFUN ENV-DISALLOW-INLINES (ENV NAMES)
   (SETF (ENV-ALLOWED-INLINES ENV)
         (SET-DIFFERENCE (ENV-ALLOWED-INLINES ENV)
                NAMES))
   (SETF (ENV-DISALLOWED-INLINES ENV)
         (UNION (ENV-DISALLOWED-INLINES ENV)
                NAMES)))

(DEFUN ENV-INLINE-ALLOWED (ENV NAME)
   (COND
      ((MEMBER NAME (ENV-ALLOWED-INLINES ENV)
               :TEST
               'EQ)
       T)
      ((MEMBER NAME (ENV-DISALLOWED-INLINES ENV)
               :TEST
               'EQ)
       NIL)
      (T (LET ((PARENT (ENV-PARENT ENV)))
              (IF (ENV-P PARENT)
                  (ENV-INLINE-ALLOWED PARENT NAME)
                  ;; We don't currently have a way to note globally inline-able functions.  Thus, if you run out of environments, you don't have
                  ;; permission to inline it.
                  NIL)))))

(DEFUN ENV-INLINE-DISALLOWED (ENV NAME)
   (COND
      ((MEMBER NAME (ENV-DISALLOWED-INLINES ENV)
               :TEST
               'EQ)
       T)
      ((MEMBER NAME (ENV-ALLOWED-INLINES ENV)
               :TEST
               'EQ)
       NIL)
      (T (LET ((PARENT (ENV-PARENT ENV)))
              (IF (ENV-P PARENT)
                  (ENV-INLINE-DISALLOWED PARENT NAME)
                  (XCL::GLOBALLY-NOTINLINE-P NAME))))))

(DEFUN ENV-PROCLAIM-SPECIAL (ENV NAME)
   (PUSH NAME (ENV-DECLARED-SPECIALS (FIND-TOP-ENVIRONMENT ENV)))
   NAME)

(DEFUN ENV-PROCLAIMED-SPECIAL-P (ENV NAME)
   (MEMBER NAME (ENV-DECLARED-SPECIALS (FIND-TOP-ENVIRONMENT ENV))
          :TEST
          'EQ))

(DEFUN ENV-PROCLAIM-GLOBAL (ENV NAME)
   (PUSH NAME (ENV-DECLARED-GLOBALS (FIND-TOP-ENVIRONMENT ENV)))
   NAME)

(DEFUN ENV-PROCLAIMED-GLOBAL-P (ENV NAME)
   (MEMBER NAME (ENV-DECLARED-GLOBALS (FIND-TOP-ENVIRONMENT ENV))
          :TEST
```

```
         'EQ))

(DEFUN ENV-DECLARE-SPECIALS (ENV SPECIALS)
   (SETF (ENV-DECLARED-SPECIALS ENV)
         (APPEND SPECIALS (ENV-DECLARED-SPECIALS ENV))))

(DEFUN ENV-DECLARE-GLOBALS (ENV GLOBALS)
   (SETF (ENV-DECLARED-GLOBALS ENV)
         (APPEND GLOBALS (ENV-DECLARED-GLOBALS ENV))))

(DEFUN ENV-DECLARE-A-SPECIAL (ENV VAR)
   (PUSH VAR (ENV-DECLARED-SPECIALS ENV)))

(DEFUN ENV-DECLARE-A-GLOBAL (ENV VAR)
   (PUSH VAR (ENV-DECLARED-GLOBALS ENV)))

(DEFUN FIND-TOP-ENVIRONMENT (ENV)
   (IL:|until| (NOT (ENV-P (ENV-PARENT ENV))) IL:|do| (SETQ ENV (ENV-PARENT ENV)))
   ENV)

(DEFUN RESOLVE-VARIABLE-REFERENCE (CURRENT-ENV SYMBOL &OPTIONAL (SETQ-P NIL))
   (DECLARE (SPECIAL IL:SPECVARS IL:LOCALVARS IL:LOCALFREEVARS IL:GLOBALVARS))
   (LET ((OBJ ;; Check up the chain of environments for bindings or local declarations.
              (DO ((ENV CURRENT-ENV (ENV-PARENT ENV))
                   TEMP)
                  ((OR (EQ ENV NIL)
                       (EQ ENV T))
                   ;; If we hit the end of the chain, then look for proclamations and check LOCALVARS, SPECVARS and GLOBALVARS.
                   (COND
                      ((AND SETQ-P (OR (LOCAL-CONSTANT-P SYMBOL)
                                       (CONSTANTP SYMBOL)))
                       (WARN "Attempt to SETQ a declared constant: ~S" SYMBOL)
                       (MAKE-VARIABLE :NAME SYMBOL :SCOPE :GLOBAL :KIND :VARIABLE))
                      ((LOCAL-CONSTANT-P SYMBOL)
                       (MAKE-LITERAL :VALUE (CONSTANT-VALUE SYMBOL)))
                      ((CONSTANTP SYMBOL)
                       (MULTIPLE-VALUE-BIND (VALUE FOLDABLE?)
                           (CHECK-GLOBAL-CONSTANT SYMBOL)
                         (IF (NOT FOLDABLE?)
                             (MAKE-VARIABLE :NAME SYMBOL :SCOPE :GLOBAL :KIND :VARIABLE)
                             (MAKE-LITERAL :VALUE (SETF (CONSTANT-VALUE SYMBOL)
                                                        VALUE)))))
                      ((OR (IL:VARIABLE-GLOBAL-P SYMBOL)
                           (MEMBER SYMBOL IL:GLOBALVARS :TEST 'EQ))
                       (MAKE-VARIABLE :NAME SYMBOL :SCOPE :GLOBAL :KIND :VARIABLE))
                      ((OR (AND (EQ IL:SPECVARS T)
                                (NOT (MEMBER SYMBOL IL:LOCALVARS :TEST 'EQ)))
                           (MEMBER SYMBOL IL:SPECVARS :TEST 'EQ)
                           (MEMBER SYMBOL IL:LOCALFREEVARS :TEST 'EQ)
                           (IL:VARIABLE-GLOBALLY-SPECIAL-P SYMBOL))
                       (MAKE-VARIABLE :NAME SYMBOL :SCOPE :SPECIAL :KIND :VARIABLE))
                      (T (UNLESS (MEMBER SYMBOL *AUTOMATIC-SPECIAL-DECLARATIONS* :TEST 'EQ)
                             (WARN "The variable ~S was unknown and has been declared SPECIAL." SYMBOL)
                             (PUSH SYMBOL *AUTOMATIC-SPECIAL-DECLARATIONS*))
                         (ENV-DECLARE-A-SPECIAL CURRENT-ENV SYMBOL)
                         (MAKE-VARIABLE :NAME SYMBOL :SCOPE :SPECIAL :KIND :VARIABLE))))
                   ;; In each environment, look for bindings or declarations that involve this variable.
                   (COND
                      ((SETQ TEMP (ASSOC SYMBOL (ENV-VENV ENV)))
                       (RETURN (CDR TEMP)))
                      ((MEMBER SYMBOL (ENV-DECLARED-SPECIALS ENV)
                               :TEST
                               'EQ)
                       (RETURN (MAKE-VARIABLE :NAME SYMBOL :SCOPE :SPECIAL :KIND :VARIABLE)))
                      ((MEMBER SYMBOL (ENV-DECLARED-GLOBALS ENV)
                               :TEST
                               'EQ)
                       (RETURN (MAKE-VARIABLE :NAME SYMBOL :SCOPE :GLOBAL :KIND :VARIABLE)))))))
         ;; SETQ's want a bare VARIABLE, not a VAR-REF.
         (IF (AND (NULL SETQ-P)
                  (VARIABLE-P OBJ))
             (MAKE-VAR-REF :VARIABLE OBJ)
             OBJ)))

(DEFUN RESOLVE-VARIABLE-BINDING (ENV SYMBOL)
   (DECLARE (SPECIAL *NEW-GLOBALS* *NEW-SPECIALS* IL:SPECVARS IL:LOCALVARS IL:LOCALFREEVARS IL:GLOBALVARS))
```

```
   (COND
      ((OR (LOCAL-CONSTANT-P SYMBOL)
           (CONSTANTP SYMBOL))
       (CERROR "Make a lexical binding anyway." "The symbol ~S is declared as a constant and thus cannot be
              bound." SYMBOL)
       :LEXICAL)
      ((OR (MEMBER SYMBOL *NEW-GLOBALS* :TEST 'EQ)
           (ENV-PROCLAIMED-GLOBAL-P ENV SYMBOL)
           (IL:VARIABLE-GLOBAL-P SYMBOL)
           (MEMBER SYMBOL IL:GLOBALVARS :TEST 'EQ))
       (CERROR "Make a lexical binding anyway." "The symbol ~S is declared as a global and thus cannot be
              bound." SYMBOL)
       :LEXICAL)
      ((OR (MEMBER SYMBOL *NEW-SPECIALS* :TEST 'EQ)
           (ENV-PROCLAIMED-SPECIAL-P ENV SYMBOL)
           (IL:VARIABLE-GLOBALLY-SPECIAL-P SYMBOL)
           (MEMBER SYMBOL IL:LOCALFREEVARS :TEST 'EQ)
           (IF (EQ IL:SPECVARS T)
               (NOT (MEMBER SYMBOL IL:LOCALVARS :TEST 'EQ))
               (MEMBER SYMBOL IL:SPECVARS :TEST 'EQ)))
       :SPECIAL)
      (T :LEXICAL)))
```

(DEFUN **VALUE-FOLDABLE-P** (VALUE)

;;; Should we replace a reference to a constant variable with the given value? Be careful: this predicate must imply FASL:VALUE-DUMPABLE-P. Also ;;; be careful about allowing folding of objects with components due to the EQ-ness problem.

```
   (TYPEP VALUE '(OR SYMBOL NUMBER CHARACTER)))
```

(DEFUN **CHECK-GLOBAL-CONSTANT** (SYMBOL)

;;; Find the value for the globally-declared constant SYMBOL and decide whether or not it should be folded into code that references it. Return two ;;; values, the constant value and a boolean which is true iff the value should be used.

```
   (LET ((LOOKUP (GETHASH SYMBOL IL:COMPVARMACROHASH)))
        (COND
           ((NULL LOOKUP)
            ;; The symbol isn't in the COMPVARMACROHASH.  If it's bound, then use its value.  This is useful for keywords, for example.
            (IF (BOUNDP SYMBOL)
                (VALUES (SYMBOL-VALUE SYMBOL)
                        T)
                (ERROR "BUG: ~S is declared as a constant, but no value for it is known." SYMBOL)))
           ((OR (ATOM LOOKUP)
                (NOT (EQ (CAR LOOKUP)
                         'IL:CONSTANT))
                (NULL (CDR LOOKUP))
                (NOT (NULL (CDDR LOOKUP))))
            (ERROR "BUG: The value of ~S in the constants hash table, ~S, has an illegal form." SYMBOL LOOKUP))
           (T (LET* ((VALUE-EXPR (CADR LOOKUP))
                     (VALUE (EVAL VALUE-EXPR)))
                ;; Unless the VALUE-EXPR is the same as the SYMBOL (as it will be for all Common Lisp constants), we have no way of
                ;; getting the value of the constant other than folding it in now.  For the cases where the VALUE-EXPR is the same as the
                ;; SYMBOL, we can afford to use the more conservative VALUE-FOLDABLE-P test.
                (VALUES VALUE (OR (NOT (EQ VALUE-EXPR SYMBOL))
                                  (VALUE-FOLDABLE-P VALUE)))))))
```

(DEFUN **CONSTANT-VALUE** (SYMBOL)

```
   (LET ((VALUE (GETHASH SYMBOL *CONSTANTS-HASH-TABLE*)))
        (ASSERT VALUE NIL "~S is not a known constant" SYMBOL)
        (CAR VALUE)))
```

(DEFUN **SET-CONSTANT-VALUE** (SYMBOL VALUE)

```
   (CAR (SETF (GETHASH SYMBOL *CONSTANTS-HASH-TABLE*)
              (LIST VALUE))))
```

(DEFUN **LOCAL-CONSTANT-P** (SYMBOL)

```
   (GETHASH SYMBOL *CONSTANTS-HASH-TABLE*))
```

(DEFSETF **CONSTANT-VALUE** SET-CONSTANT-VALUE)

(DEFVAR ***CONSTANTS-HASH-TABLE*** NIL

;;; Hash-table for keeping track of the constants defined in a given file.

```
   )
```

```
(DEFVAR *ENVIRONMENT* NIL

;;; The current environment of declarations, bindings, etc.  Rebound at several places within the compiler.

   )


(DEFVAR *CONTEXT* NIL
   "The evaluation context of the current form.  Rebound at several places within the compiler.")


(DEFCONSTANT *ARGUMENT-CONTEXT* (MAKE-CONTEXT :VALUES-USED 1)
                                "Context structure to be shared among all evaluations in return
                                position.")


(DEFCONSTANT *EFFECT-CONTEXT* (MAKE-CONTEXT :VALUES-USED 0)
                              "Context structure to be shared among all evaluations for effect.")


(DEFCONSTANT *NULL-CONTEXT* (MAKE-CONTEXT)
                            "Context structure to be shared among all expressions in a position without any
                            contextual information.")


(DEFCONSTANT *PREDICATE-CONTEXT* (MAKE-CONTEXT :VALUES-USED 1 :PREDICATE-P T)
                                 "Context structure to be shared among all evaluations as predicates.")

;; External interface to environments


(DEFUN ENV-BOUNDP (ENV NAME)

;;; Only used by clients outside the compiler (i.e., macros and optimizers).

   (LET ((LOOKUP (ASSOC NAME (ENV-VENV ENV))))
        (COND
           (LOOKUP (LET ((SCOPE-OR-STRUCT (CDR LOOKUP)))
                        (IF (VARIABLE-P SCOPE-OR-STRUCT)
                            (VARIABLE-SCOPE SCOPE-OR-STRUCT)
                            SCOPE-OR-STRUCT)))
           ((MEMBER NAME (ENV-DECLARED-SPECIALS ENV)
                    :TEST
                    'EQ)
            :SPECIAL)
           ((MEMBER NAME (ENV-DECLARED-GLOBALS ENV)
                    :TEST
                    'EQ)
            :GLOBAL)
           (T (LET ((PARENT (ENV-PARENT ENV)))
                   (AND (ENV-P PARENT)
                        (ENV-BOUNDP PARENT NAME)))))))


(DEFUN ENV-FBOUNDP (ENV NAME &KEY (LEXICAL-ONLY NIL))

;;; Return two values: the KIND of the given NAME (either :MACRO or :FUNCTION) and, iff KIND = :MACRO, the expansion function for the macro.

;;; When LEXICAL-ONLY is true, we're only supposed to tell the user about lexically apparent functions and macros.  The environment chain ends in one
;;; representing the various top-level objects in the file.  In particular, top-level DEFMACRO's are in there.  Thus, in our search here, we must be careful
;;; to avoid looking in the top environment.  We can distinguish such environments because their PARENT is T.

   (LABELS ((FIND-FN (ENV)
                     (LET ((PARENT (ENV-PARENT ENV)))
                          (UNLESS (AND LEXICAL-ONLY (EQ PARENT T))
                              (LET ((LOOKUP (ASSOC NAME (ENV-FENV ENV)
                                                   :TEST
                                                   'EQ)))
                                   (IF (NULL LOOKUP)
                                       (AND PARENT (NOT (EQ PARENT T))
                                            (FIND-FN PARENT))
                                       (VALUES-LIST (CDR LOOKUP))))))))
     (FIND-FN ENV)))


(DEFUN COPY-ENV-WITH-FUNCTION (ENV FN &OPTIONAL (KIND :FUNCTION)
                                   EXP-FN)
   (LET ((NEW-ENV (IF ENV
                      (COPY-ENV ENV)
                      (MAKE-EMPTY-ENV))))
        (ENV-BIND-FUNCTION NEW-ENV FN KIND EXP-FN)
        NEW-ENV))


(DEFUN COPY-ENV-WITH-VARIABLE (ENV VAR &OPTIONAL (KIND :LEXICAL))
   (LET ((NEW-ENV (IF ENV
                      (COPY-ENV ENV)
```

```
                    (MAKE-EMPTY-ENV))))
        (ENV-BIND-VARIABLE NEW-ENV VAR KIND)
        NEW-ENV))

(DEFUN MAKE-EMPTY-ENV ()
   (MAKE-ENV))
```

;; Describe the machine we're running on and the target for hte compiled code

```
(DEFVAR *HOST-ARCHITECTURE* NIL)

(DEFVAR *TARGET-ARCHITECTURE* NIL)
```

;; Arrange for the correct compiler to be used.

```
(IL:PUTPROPS IL:XCLC-ENV-CTXT IL:FILETYPE COMPILE-FILE)
```

;; Arrange for the correct reader environment.

```
(IL:PUTPROPS IL:XCLC-ENV-CTXT IL:MAKEFILE-ENVIRONMENT (:READTABLE "XCL" :PACKAGE (DEFPACKAGE "COMPILER"
                                                                                    (:USE "LISP" "XCL"))))

(IL:PUTPROPS IL:XCLC-ENV-CTXT IL:COPYRIGHT ("Venue & Xerox Corporation" 1986 1987 1988 1990))
```

FUNCTION INDEX

VARIABLE INDEX

CONSTANT INDEX

STRUCTURE INDEX

PROPERTY INDEX

SETF INDEX