*File created:* 23-May-90 13:29:17 **{DSK}<usr>local>lde>lispcore>sources>XCLC-TREES.;2**

*changes to:* (IL:VARS IL:XCLC-TREESCOMS)

*previous date:* 7-Jan-88 14:50:18 {DSK}<usr>local>lde>lispcore>sources>XCLC-TREES.;1

*Read Table:* XCL

*Package:* COMPILER

*Format:* XCCS



```
(IL:RPAQQ IL:XCLC-TREESCOMS
          (

;;; Program trees

            (IL:DEFINE-TYPES NODES)
            (IL:DECLARE\: IL:EVAL@LOAD IL:EVAL@COMPILE IL:DOCOPY (IL:FUNCTIONS NODE-TYPE-NAME
                                                                               CONSTRUCT-COMPILER-SYMBOL)
                   (IL:VARIABLES *NODE-TYPES*))
            (IL:FUNCTIONS MAKE-NODE-METHOD)
            (IL:FUNCTIONS DEFNODE)
            (IL:STRUCTURES NODE BLIPPER CALLER SEGMENT VARIABLE-STRUCT)
            (NODES BLOCK-NODE CALL-NODE CATCH-NODE GO-NODE IF-NODE LABELS-NODE LAMBDA-NODE LITERAL-NODE
                   MV-CALL-NODE MV-PROG1-NODE OPCODES-NODE PROGN-NODE PROGV-NODE RETURN-NODE SETQ-NODE
                   TAGBODY-NODE THROW-NODE UNWIND-PROTECT-NODE VAR-REF-NODE)
            (IL:VARIABLES *LITERALLY-NIL* *LITERALLY-T*)
            (IL:FUNCTIONS MAKE-REFERENCE-TO-VARIABLE)
            (IL:FUNCTIONS NODE-DISPATCH)
            ;; Eliminating tree circularities
            (IL:FUNCTIONS RELEASE-TREE)
            (IL:FUNCTIONS DELETEF DELETEF-1 DELETEF-2)
            (IL:FUNCTIONS RELEASE-BLOCK RELEASE-CALL RELEASE-CATCH RELEASE-GO RELEASE-IF RELEASE-LABELS
                   RELEASE-LAMBDA RELEASE-LITERAL RELEASE-MV-CALL RELEASE-MV-PROG1 RELEASE-OPCODES RELEASE-PROGN
                   RELEASE-PROGV RELEASE-RETURN RELEASE-SETQ RELEASE-TAGBODY RELEASE-THROW RELEASE-UNWIND-PROTECT
                   RELEASE-VAR-REF)
            ;; Copying tree structure
            (IL:FUNCTIONS COPY-CODE COPY-NODES)
            (IL:VARIABLES *COPY-NODE-TABLE*)
            (IL:FUNCTIONS COPY-NODE-BLOCK COPY-NODE-CALL COPY-NODE-CATCH COPY-NODE-GO COPY-NODE-IF
                   COPY-NODE-LABELS COPY-NODE-LAMBDA COPY-NODE-LITERAL COPY-NODE-MV-CALL COPY-NODE-MV-PROG1
                   COPY-NODE-OPCODES COPY-NODE-PROGN COPY-NODE-PROGV COPY-NODE-RETURN COPY-NODE-SETQ
                   COPY-NODE-TAGBODY COPY-NODE-THROW COPY-NODE-UNWIND-PROTECT COPY-NODE-VAR-REF)
            (IL:FUNCTIONS COPY-NODE-LIST COPY-VARIABLE FIND-COPIED-VARIABLE)
            ;; Arrange for the correct compiler to be used.
            (IL:PROP IL:FILETYPE IL:XCLC-TREES)
            ;; Arrange for the correct makefile-environment
            (IL:PROP IL:MAKEFILE-ENVIRONMENT IL:XCLC-TREES)))

;;; Program trees

(DEF-DEFINE-TYPE NODES "XCL compiler tree node types")

(IL:DECLARE\: IL:EVAL@LOAD IL:EVAL@COMPILE IL:DOCOPY

(DEFUN NODE-TYPE-NAME (TRUE-NAME)
   (CONSTRUCT-COMPILER-SYMBOL TRUE-NAME "-NODE"))

(DEFUN CONSTRUCT-COMPILER-SYMBOL (&REST PARTS)
   (INTERN (APPLY 'CONCATENATE 'STRING (MAPCAR 'STRING PARTS))
          "COMPILER"))

(DEFVAR *NODE-TYPES* NIL
   "List of the names of the various kinds of parse-tree nodes.  Names are put on this list by DEFNODE.")
)

(DEFMACRO MAKE-NODE-METHOD (PREFIX)

;;; Used only during compiler development, this is only useful inside of SEdit, when I can type
    ;; (MAKE-NODE-METHOD FOO)

;;; and then hit Meta-X to get the list of function names associated with the new FOO method on nodes.  Someday, this will change to cons up the
;;; names of PCL methods.
```

```
   (IL:SORT (IL:FOR F IL:IN *NODE-TYPES* IL:COLLECT (CONSTRUCT-COMPILER-SYMBOL PREFIX "-" F))))

(DEFDEFINER (DEFNODE (:NAME (LAMBDA (WHOLE)
                                    (LET* ((NAME-AND-OPTIONS (SECOND WHOLE))
                                           (TRUE-NAME (IF (CONSP NAME-AND-OPTIONS)
                                                          (FIRST NAME-AND-OPTIONS)
                                                          NAME-AND-OPTIONS)))
                                          (NODE-TYPE-NAME TRUE-NAME)))))
    NODES (TRUE-NAME &REST DEFSTRUCT-BODY)
   (LET ((PARENT 'NODE)
         OPTIONS)
        (WHEN (CONSP TRUE-NAME)
            (PSETQ TRUE-NAME (CAR TRUE-NAME)
                   OPTIONS
                   (CDR TRUE-NAME))
            (IL:FOR OPTION IL:IN OPTIONS IL:DO (ECASE (FIRST OPTION)
                                                      ((:PARENT) (SETQ PARENT (SECOND OPTION))))))
        `(PROGN (EVAL-WHEN (COMPILE LOAD EVAL)
                           (PUSHNEW ',TRUE-NAME *NODE-TYPES*))
                (DEFSTRUCT (,(NODE-TYPE-NAME TRUE-NAME)
                            (:CONC-NAME ,(CONSTRUCT-COMPILER-SYMBOL TRUE-NAME "-"))
                            (:PREDICATE ,(CONSTRUCT-COMPILER-SYMBOL TRUE-NAME "-P"))
                            (:COPIER NIL)
                            (:CONSTRUCTOR ,(CONSTRUCT-COMPILER-SYMBOL "MAKE-" TRUE-NAME))
                            (:INCLUDE ,PARENT)
                            (:INLINE NIL))
                   ,@DEFSTRUCT-BODY))))
```

```
(DEFSTRUCT (NODE (:INLINE T))
```

;;; METAP is non-NIL if and only if the tree below this point has already been meta-evaluated. If a given node has this bit set, then every node below ;;; should have it set as well.

;;; SUBST-P is non-NIL if and only if this node was substituted in for a variable during meta-evaluation. See META-CALL-LAMBDA-SUBSTITUTE.

;;; EFFECTS is either :NONE, :CONS, :ANY, or a list of variables representing the side effects possible in the subtree below this node.

;;; AFFECTED is like EFFECTS but describes the side-effects that can affect the computation of the subtree below this node.

```
   (META-P NIL)
   (SUBST-P NIL)
   (EFFECTS NIL)
   (AFFECTED NIL))
```

```
(DEFSTRUCT (BLIPPER (:INCLUDE NODE)
                    (:INLINE T))
```

;;; REFERENCES is a list of the GO or RETURN structures whose reference will be cut off if this blipper is made into a separate frame.

;;; CLOSED-OVER-P is non-NIL if this blipper has dynamically remote references.

;;; NEW-FRAME-P is non-NIL if this blipper must be a separate frame.

```
   REFERENCES
   CLOSED-OVER-P
   NEW-FRAME-P)
```

```
(DEFSTRUCT (CALLER (:INCLUDE NODE)
                   (:INLINE T))
```

;;; Shared parent of CALL and MV-CALL.

;;; NOT-INLINE is non-NIL iff this call should not be inline-expanded.

```
   (NOT-INLINE NIL))
```

```
(DEFSTRUCT (SEGMENT (:INLINE T))
```

;;; TAGS is a list of symbols which are tags for the forms in STMTS.

;;; STMTS is a list of structures for the forms tagged by the symbols in TAGS.

;;; CLOSED-OVER-P is non-NIL if this segment may be referred to from another frame.

;;; LOCAL-TAG is the LAP tag to which local GOs should point.

;;; REMOTE-TAG is the LAP tag to which non-local GOs should point."

```
   TAGS
   STMTS
   CLOSED-OVER-P
   LOCAL-TAG
   REMOTE-TAG)
```

```
(DEFSTRUCT (VARIABLE-STRUCT (:CONC-NAME VARIABLE-)
                            (:CONSTRUCTOR MAKE-VARIABLE)
                            (:COPIER NIL)
                            (:PREDICATE VARIABLE-P)
                            (:INLINE T))
    ;; SCOPE is one of :lexical, :special or :global.
    ;; KIND is one of :variable or :function.
    ;; NAME is a string (for :lexical names) or symbol (for the others) giving the programmer's name for the variable.
    ;; BINDER is the LAMBDA or LABELS structure that binds this variable.
    ;; LAP-VAR is the LAP-code variable corresponding to this one.
    ;; CLOSED-OVER is non-NIL if this variable might be referred to from a distance.
    ;; READ-REFS and WRITE-REFS are lists of references to this variable in VAR-REF's and SETQ's, respectively.
    ;; The defaults are set up to allow the easy generation of anonymous temporaries, for example during the meta-evaluation of called lambdas.
    (SCOPE :LEXICAL)
    (KIND :VARIABLE)
    (NAME "Anonymous")
    (BINDER NIL)
    (READ-REFS NIL)
    (WRITE-REFS NIL)
    (LAP-VAR NIL)
    (CLOSED-OVER NIL))

(DEFNODE (BLOCK (:PARENT BLIPPER))
;;; NAME is the symbol which names the block.
;;; STMT is the structure representing the form or forms making up the body of the block.
;;; CONTEXT is the evaluation context of the block, for use by any RETURN-FROM's for this block.
;;; CLOSED-OVER-VARS is a list of lexical VARIABLEs whose storage should be allocated on entry to this block.
;;; FRAME is the value of *current-frame* for the body of block.
;;; BLIP-VAR is the LAP variable containing the value of the blip associated with this block.
;;; END-TAG is the LAP tag pointing to the end of the code for this block.
;;; STK-NUM is the LAP stack-level number for the context of this block.
    NAME
    STMT
    CONTEXT
    CLOSED-OVER-VARS
    FRAME
    BLIP-VAR
    END-TAG
    STK-NUM)

(DEFNODE (CALL (:PARENT CALLER))
;;; FN is the value representing the function to be applied
;;; ARGS is a list of structures for the arguments
    FN
    ARGS)

(DEFNODE (CATCH (:PARENT BLIPPER))
;;; TAG is the structure representing the form to be evaluated to get the catch-tag.
;;; STMT is the structure representing the form or forms to be evaluated inside the catch.
;;; CLOSED-OVER-VARS is a list of lexical VARIABLEs whose storage should be allocated on entry to the catch body.  It need not be allocated before
;;; evaluating the tag, however.
    TAG
    STMT
    CLOSED-OVER-VARS)

(DEFNODE GO
;;; TAGBODY is the structure representing the tagbody form containing the target of this go.
;;; TAG is the label in that tagbody to which this go goes.
```

```
   TAGBODY
   TAG)
```

(DEFNODE **IF**

;;; PRED is the structure representing the predicate form.

;;; THEN is the structure representing the consequent form.

;;; ELSE is the structure representing the alternative form.

```
   PRED
   THEN
   ELSE)
```

(DEFNODE **LABELS**

;;; FUNS is an alist mapping the VARIABLE structures representing the names of the functions to the LAMBDA structures representing the functions ;;; themselves.

;;; BODY is the structure representing the forms in the body of the LABELS.

;;; CLOSED-OVER is a list of lexical VARIABLEs whose storage should be allocated on entry to this labels.

```
   FUNS
   BODY
   CLOSED-OVER-VARS)
```

(DEFNODE **LAMBDA**

;;; NAME is the string or symbol to be used to name this lambda.

;;; ARG-TYPE is the Interlisp ARGTYPE of this LAMBDA or NIL if it's Common Lisp.

;;; NO-SPREAD-NAME is the symbol naming the parameter of this LAMBDA if it's an Interlisp LAMBDA-NO-SPREAD, otherwise NIL.

;;; REQUIRED is a list of VARIABLEs representing the required parameters of the lambda-form.

;;; OPTIONAL is a list of values representing the optional parameters of the lambda-form. Each value is a list of up to three items: the VARIABLE, the ;;; structure representing the init-form, and an optional VARIABLE representing the supplied-p parameter.

;;; REST is either NIL or a VARIABLE representing the &rest parameter of the lambda-form.

;;; KEYWORD is a list of lists, each one representing a keyword-parameter to the lambda-form. Each list has up to four elements: 1) The keyword to be ;;; recognized for the parameter, 2) the VARIABLE to be bound, 3) a structure representing the init-form, and 4) an optional VARIABLE representing any ;;; supplied-p parameter.

;;; ALLOW-OTHER-KEYS is T if and only if &allow-other-keys was specified in the lambda-list.

;;; BODY is a structure representing the form or forms of the body of the lambda-form.

;;; APPLIED-EFFECTS and APPLIED-AFFECTED are the side-effects of this lambda when applied.

;;; CLOSED-OVER-VARS is a list of lexical VARIABLEs to be allocated storage on entry to this lambda.

;;; NEW-FRAME-P is non-NIL if this LAMBDA is to be compiled as a separate frame. Set during frame annotation and used during other annotations ;;; and code generation.

;;; TAIL-CALL-TAG is, if non-NIL, a tag number to be used at the top of the body of the lambda as a target for tail-recursive jumps.

```
   NAME
   ARG-TYPE
   NO-SPREAD-NAME
   REQUIRED
   OPTIONAL
   REST
   KEYWORD
   ALLOW-OTHER-KEYS
   BODY
   APPLIED-EFFECTS
   APPLIED-AFFECTED
   CLOSED-OVER-VARS
   NEW-FRAME-P
   TAIL-CALL-TAG)
```

(DEFNODE **LITERAL**

;;; VALUE is the actual Lisp value of the literal.

```
   VALUE)
```

(DEFNODE (**MV-CALL** (:PARENT CALLER))

;;; FN is a structure representing the function to be called with the values.

;;; ARG-EXPRS is a list of structures representing the forms to be evaluated to generate the values.

```
   FN
   ARG-EXPRS)
```

(DEFNODE **MV-PROG1**

;;; STMTS is a list of structures representing the forms in the body of the multiple-value-prog1. (car stmts) is the structure for the form whose values are
;;; the values of this expression.

```
   STMTS)
```

(DEFNODE **OPCODES**

;;; BYTES is the list of bytes to be generated.

```
   BYTES)
```

(DEFNODE **PROGN**

;;; STMTS is a list of the structures representing the forms of the PROGN.

```
   STMTS)
```

(DEFNODE **PROGV**

;;; SYMS-EXPR is the structure representing the form to be evaluated to get the list of symbols to be bound.

;;; VALS-EXPR is the structure representing the form to be evaluated to get the list of values ot be bound to the symbols.

;;; STMT is the structure representing the form or forms in the body of the progv.

```
   SYMS-EXPR
   VALS-EXPR
   STMT)
```

(DEFNODE **RETURN**

;;; BLOCK is the structure for the block from which this return-from returns.

;;; VALUE is the structure for the form to be evaluated for the returned value.

```
   BLOCK
   VALUE)
```

(DEFNODE **SETQ**

;;; VAR is the VARIABLE structure representing the variable being set.

;;; VALUE is the structure for the form whose value will be used.

```
   VAR
   VALUE)
```

(DEFNODE (**TAGBODY** (:PARENT BLIPPER))

;;; SEGMENTS is a list of SEGMENT structures representing the tags and forms of the tagbody.

;;; CLOSED-OVER-VARS is a list of lexical VARIABLEs to be allocated storage on entry to this tagbody.

;;; FRAME is the value of *CURRENT-FRAME* at the top level of this tagbody.

;;; BLIP-VAR is the LAP variable containing the control blip for this tagbody.

;;; STK-NUM is the stack-state number for the top level of this tagbody.

```
   SEGMENTS
   CLOSED-OVER-VARS
   FRAME
   BLIP-VAR
   STK-NUM)
```

(DEFNODE **THROW**

;;; TAG is the structure for the form whose value will be the catch-tag to be thrown.

```
;;; VALUE is the structure for the form whose values will be thrown to the tag.

   TAG
   VALUE)


(DEFNODE UNWIND-PROTECT

;;; STMT is the structure for the form to be protected.

;;; CLEANUP is the structure for the cleanup form or forms.

   STMT
   CLEANUP)


(DEFNODE VAR-REF

;;; The wrapper for a variable reference.  VARIABLE is the VARIABLE structure being referenced.

   VARIABLE)


(DEFCONSTANT *LITERALLY-NIL* (MAKE-LITERAL :VALUE NIL)
                             "The LITERAL structure to be used for all occurences of NIL, in order to save
                             allocations.")


(DEFCONSTANT *LITERALLY-T* (MAKE-LITERAL :VALUE T)
                           "The LITERAL structure to be used for all occurences of T, to save allocations")


(DEFMACRO MAKE-REFERENCE-TO-VARIABLE (&REST ARGS)
   `(MAKE-VAR-REF :VARIABLE (MAKE-VARIABLE ,@ARGS)))


(DEFMACRO NODE-DISPATCH (PREFIX NODE &REST ARGS)

;;; Expands into a ETYPECASE stmt dispatching on the given node to a call on the function named <PREFIX>-<TYPE> with the NODE and the other
;;; ARGS as arguments.  The node expression is evaluated only once.

   `(LET (($$NODE$$ ,NODE))
         (ETYPECASE $$NODE$$
             (IL:\\\,@ (MAPCAR #'(LAMBDA (TRUE-NAME)
                                         `((,(NODE-TYPE-NAME TRUE-NAME))
                                           (,(CONSTRUCT-COMPILER-SYMBOL PREFIX "-" TRUE-NAME)
                                            $$NODE$$
                                            ,@ARGS)))
                               *NODE-TYPES*)))))

;; Eliminating tree circularities


(DEFUN RELEASE-TREE (TREE)

;;; Release-Tree methods should arrange for their sub-tree to be removed from the program tree.  Any circularities should be removed and the results of
;;; analysis should be fixed up.  However, those kinds of nodes that are shared among multiple uses in a single tree (such as variables), should not
;;; destroy any fields that other uses are counting on.

   (WHEN (NOT (NULL TREE))
       (SETF (NODE-EFFECTS TREE)
             NIL)
       (SETF (NODE-AFFECTED TREE)
             NIL)
       (NODE-DISPATCH RELEASE TREE)))


(DEFMACRO DELETEF (ITEM PLACE)
   `(DELETEF-1 ,PLACE ,ITEM))


(DEFINE-MODIFY-MACRO DELETEF-1 (ITEM) DELETEF-2)


(DEFMACRO DELETEF-2 (PLACE ITEM)
   `(DELETE ,ITEM ,PLACE))


(DEFUN RELEASE-BLOCK (NODE)
   (SETF (BLOCK-FRAME NODE)
         NIL)
   (SETF (BLOCK-REFERENCES NODE)
         NIL)
   (RELEASE-TREE (BLOCK-STMT NODE)))


(DEFUN RELEASE-CALL (NODE)
```

```
   (RELEASE-TREE (CALL-FN NODE))
   (MAPC #'RELEASE-TREE (CALL-ARGS NODE)))


(DEFUN RELEASE-CATCH (NODE)
   (SETF (CATCH-REFERENCES NODE)
         NIL)
   (RELEASE-TREE (CATCH-TAG NODE))
   (RELEASE-TREE (CATCH-STMT NODE)))


(DEFUN RELEASE-GO (NODE)
   (SETF (GO-TAGBODY NODE)
         NIL))


(DEFUN RELEASE-IF (NODE)
   (RELEASE-TREE (IF-PRED NODE))
   (RELEASE-TREE (IF-THEN NODE))
   (RELEASE-TREE (IF-ELSE NODE)))


(DEFUN RELEASE-LABELS (NODE)
   (IL:|for| BINDING IL:|in| (LABELS-FUNS NODE) IL:|do| (SETF (VARIABLE-BINDER (CAR BINDING))
                                                             NIL)
                                                       (RELEASE-TREE (CDR BINDING)))
   (RELEASE-TREE (LABELS-BODY NODE)))


(DEFUN RELEASE-LAMBDA (NODE)                                    ;
   (SETF (LAMBDA-APPLIED-EFFECTS NODE)
         NIL)
   (SETF (LAMBDA-APPLIED-AFFECTED NODE)
         NIL)
   (IL:FOR VAR IL:IN (LAMBDA-REQUIRED NODE) IL:DO (SETF (VARIABLE-BINDER VAR)
                                                        NIL))
   (IL:FOR OPT-VAR IL:IN (LAMBDA-OPTIONAL NODE) IL:DO (SETF (VARIABLE-BINDER (FIRST OPT-VAR))
                                                             NIL)
                                                       (RELEASE-TREE (SECOND OPT-VAR))
                                                       (WHEN (THIRD OPT-VAR)
                                                           (SETF (VARIABLE-BINDER (THIRD OPT-VAR))
                                                                 NIL)))
   (WHEN (LAMBDA-REST NODE)
       (SETF (VARIABLE-BINDER (LAMBDA-REST NODE))
             NIL))
   (IL:FOR KEY-VAR IL:IN (LAMBDA-KEYWORD NODE) IL:DO (SETF (VARIABLE-BINDER (SECOND KEY-VAR))
                                                            NIL)
                                                      (RELEASE-TREE (THIRD KEY-VAR))
                                                      (WHEN (FOURTH KEY-VAR)
                                                          (SETF (VARIABLE-BINDER (FOURTH KEY-VAR))
                                                                NIL)))
   (RELEASE-TREE (LAMBDA-BODY NODE)))


(DEFUN RELEASE-LITERAL (NODE)
   NIL)


(DEFUN RELEASE-MV-CALL (NODE)
   (RELEASE-TREE (MV-CALL-FN NODE))
   (MAPC #'RELEASE-TREE (MV-CALL-ARG-EXPRS NODE)))


(DEFUN RELEASE-MV-PROG1 (NODE)
   (MAPC #'RELEASE-TREE (MV-PROG1-STMTS NODE)))


(DEFUN RELEASE-OPCODES (NODE)
   NIL)


(DEFUN RELEASE-PROGN (NODE)
   (MAPC #'RELEASE-TREE (PROGN-STMTS NODE)))


(DEFUN RELEASE-PROGV (NODE)
   (RELEASE-TREE (PROGV-SYMS-EXPR NODE))
   (RELEASE-TREE (PROGV-VALS-EXPR NODE))
   (RELEASE-TREE (PROGV-STMT NODE)))


(DEFUN RELEASE-RETURN (NODE)
   (RELEASE-TREE (RETURN-VALUE NODE))
   (SETF (RETURN-BLOCK NODE)
         NIL))
```

```
(DEFUN RELEASE-SETQ (NODE)

;;; Remove the WRITE-REF we're getting rid of.

   (DELETEF NODE (VARIABLE-WRITE-REFS (SETQ-VAR NODE)))
   (RELEASE-TREE (SETQ-VALUE NODE)))


(DEFUN RELEASE-TAGBODY (NODE)                                   ;
   (SETF (TAGBODY-REFERENCES NODE)
         NIL)
   (SETF (TAGBODY-FRAME NODE)
         NIL)
   (IL:|for| SEGMENT IL:|in| (TAGBODY-SEGMENTS NODE) IL:|do| (IL:|for| STMT IL:|in| (SEGMENT-STMTS SEGMENT)
                                                              IL:|do| (RELEASE-TREE STMT))))


(DEFUN RELEASE-THROW (NODE)
   (RELEASE-TREE (THROW-TAG NODE))
   (RELEASE-TREE (THROW-VALUE NODE)))


(DEFUN RELEASE-UNWIND-PROTECT (NODE)
   (RELEASE-TREE (UNWIND-PROTECT-STMT NODE))
   (RELEASE-TREE (UNWIND-PROTECT-CLEANUP NODE)))


(DEFUN RELEASE-VAR-REF (NODE)

;;; The binder field is cleared out in the binder itself, since that's when we can be sure that no more uses exist.

   (DELETEF NODE (VARIABLE-READ-REFS (VAR-REF-VARIABLE NODE)))
   (SETF (VAR-REF-VARIABLE NODE)
         NIL))

;; Copying tree structure


(DEFUN COPY-CODE (TREE)
   (LET ((*COPY-NODE-TABLE* (MAKE-HASH-TABLE)))
        (COPY-NODES TREE)))


(DEFUN COPY-NODES (TREE)

;;; COPY-NODE methods return a subtree with the same structure as the one they're given, but without any of the analysis information filled in.

   (AND TREE (NODE-DISPATCH COPY-NODE TREE)))


(DEFVAR *COPY-NODE-TABLE* NIL

;;; A hashtable mapping nodes and other structures into their copied counterparts.  Used in various COPY-NODE methods.

   )


(DEFUN COPY-NODE-BLOCK (NODE)
   (LET ((NEW-BLOCK (MAKE-BLOCK :NAME (BLOCK-NAME NODE))))
        (SETF (GETHASH NODE *COPY-NODE-TABLE*)
              NEW-BLOCK)
        (SETF (BLOCK-STMT NEW-BLOCK)
              (COPY-NODES (BLOCK-STMT NODE)))
       NEW-BLOCK))


(DEFUN COPY-NODE-CALL (NODE)
   (MAKE-CALL :FN (COPY-NODES (CALL-FN NODE))
          :ARGS
          (COPY-NODE-LIST (CALL-ARGS NODE))))


(DEFUN COPY-NODE-CATCH (NODE)
   (MAKE-CATCH :TAG (COPY-NODES (CATCH-TAG NODE))
          :STMT
          (COPY-NODES (CATCH-STMT NODE))))


(DEFUN COPY-NODE-GO (NODE)
   (LET ((TAGBODY (GETHASH (GO-TAGBODY NODE)
                           *COPY-NODE-TABLE*)))
        (MAKE-GO :TAGBODY (IF (NULL TAGBODY)                    ; This GO is to a TAGBODY not being copied.
                              (GO-TAGBODY NODE)
                              TAGBODY)
                 :TAG
                 (GO-TAG NODE))))
```

```
(DEFUN COPY-NODE-IF (NODE)
   (MAKE-IF :PRED (COPY-NODES (IF-PRED NODE))
          :THEN
          (COPY-NODES (IF-THEN NODE))
          :ELSE
          (COPY-NODES (IF-ELSE NODE))))

(DEFUN COPY-NODE-LABELS (NODE)

;;; Make one pass through the functions copying the variables and storing them in the hash table, then do the actual copying of the function bodies and
;;; the LABELS body.

   (LET* ((NEW-LABELS (MAKE-LABELS)))
         (SETF (LABELS-FUNS NEW-LABELS)
               (IL:FOR FUN IL:IN (LABELS-FUNS NODE) IL:AS NEW-VAR IL:IN (IL:FOR FUN IL:IN (LABELS-FUNS NODE)
                                                                                 IL:COLLECT (COPY-VARIABLE (CAR FUN)
                                                                                                           NEW-LABELS))
                  IL:COLLECT (CONS NEW-VAR (COPY-NODE-LAMBDA (CDR FUN)))))
         (SETF (LABELS-BODY NEW-LABELS)
               (COPY-NODES (LABELS-BODY NODE)))
         NEW-LABELS))

(DEFUN COPY-NODE-LAMBDA (NODE)
   (LET ((NEW-LAMBDA (MAKE-LAMBDA :NAME (LAMBDA-NAME NODE)
                                  :ARG-TYPE
                                  (LAMBDA-ARG-TYPE NODE)
                                  :NO-SPREAD-NAME
                                  (LAMBDA-NO-SPREAD-NAME NODE)
                                  :ALLOW-OTHER-KEYS
                                  (LAMBDA-ALLOW-OTHER-KEYS NODE))))
        (SETF (LAMBDA-REQUIRED NEW-LAMBDA)
              (IL:FOR VAR IL:IN (LAMBDA-REQUIRED NODE) IL:COLLECT (COPY-VARIABLE VAR NEW-LAMBDA)))
        (SETF (LAMBDA-OPTIONAL NEW-LAMBDA)
              (IL:FOR OPT-VAR IL:IN (LAMBDA-OPTIONAL NODE) IL:COLLECT (LIST (COPY-VARIABLE (FIRST OPT-VAR)
                                                                                           NEW-LAMBDA)
                                                                            (COPY-NODES (SECOND OPT-VAR))
                                                                            (COPY-VARIABLE (THIRD OPT-VAR)
                                                                                           NEW-LAMBDA))))
        (SETF (LAMBDA-REST NEW-LAMBDA)
              (COPY-VARIABLE (LAMBDA-REST NODE)
                             NEW-LAMBDA))
        (SETF (LAMBDA-KEYWORD NEW-LAMBDA)
              (IL:FOR KEY-VAR IL:IN (LAMBDA-KEYWORD NODE) IL:COLLECT (LIST (FIRST KEY-VAR)
                                                                           (COPY-VARIABLE (SECOND KEY-VAR)
                                                                                          NEW-LAMBDA)
                                                                           (COPY-NODES (THIRD KEY-VAR))
                                                                           (COPY-VARIABLE (FOURTH KEY-VAR)
                                                                                          NEW-LAMBDA))))
        (SETF (LAMBDA-BODY NEW-LAMBDA)
              (COPY-NODES (LAMBDA-BODY NODE)))
        NEW-LAMBDA))

(DEFUN COPY-NODE-LITERAL (NODE)

;;; Even lowly literals are copied, since their META-P field can be important.

   (MAKE-LITERAL :VALUE (LITERAL-VALUE NODE)))

(DEFUN COPY-NODE-MV-CALL (NODE)
   (MAKE-MV-CALL :FN (COPY-NODES (MV-CALL-FN NODE))
          :ARG-EXPRS
          (COPY-NODE-LIST (MV-CALL-ARG-EXPRS NODE))))

(DEFUN COPY-NODE-MV-PROG1 (NODE)
   (MAKE-MV-PROG1 :STMTS (COPY-NODE-LIST (MV-PROG1-STMTS NODE))))

(DEFUN COPY-NODE-OPCODES (NODE)

;;; Copy the byte-list just in case somebody wants to do a transformation on it later (ugh!).

   (MAKE-OPCODES :BYTES (COPY-LIST (OPCODES-BYTES NODE))))

(DEFUN COPY-NODE-PROGN (NODE)
   (MAKE-PROGN :STMTS (COPY-NODE-LIST (PROGN-STMTS NODE))))

(DEFUN COPY-NODE-PROGV (NODE)
   (MAKE-PROGV :SYMS-EXPR (COPY-NODES (PROGV-SYMS-EXPR NODE))
          :VALS-EXPR
```

```
          (COPY-NODES (PROGV-VALS-EXPR NODE))
          :STMT
          (COPY-NODES (PROGV-STMT NODE))))

(DEFUN COPY-NODE-RETURN (NODE)
   (LET ((BLOCK (GETHASH (RETURN-BLOCK NODE)
                         *COPY-NODE-TABLE*)))
        (MAKE-RETURN :BLOCK (IF (NULL BLOCK)                    ; This is a RETURN from a BLOCK not being copied.
                                (RETURN-BLOCK NODE)
                                BLOCK)
                     :VALUE
                     (COPY-NODES (RETURN-VALUE NODE)))))

(DEFUN COPY-NODE-SETQ (NODE)
   (MAKE-SETQ :VAR (FIND-COPIED-VARIABLE (SETQ-VAR NODE))
          :VALUE
          (COPY-NODES (SETQ-VALUE NODE))))

(DEFUN COPY-NODE-TAGBODY (NODE)
   (LET ((NEW-TAGBODY (MAKE-TAGBODY)))
        (SETF (GETHASH NODE *COPY-NODE-TABLE*)
              NEW-TAGBODY)
        (SETF (TAGBODY-SEGMENTS NEW-TAGBODY)
              (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:COLLECT (MAKE-SEGMENT :TAGS (SEGMENT-TAGS SEGMENT)
                                                                                      :STMTS
                                                                                      (COPY-NODE-LIST (SEGMENT-STMTS
                                                                                                       SEGMENT)))))
        NEW-TAGBODY))

(DEFUN COPY-NODE-THROW (NODE)
   (MAKE-THROW :TAG (COPY-NODES (THROW-TAG NODE))
          :VALUE
          (COPY-NODES (THROW-VALUE NODE))))

(DEFUN COPY-NODE-UNWIND-PROTECT (NODE)
   (MAKE-UNWIND-PROTECT :STMT (COPY-NODES (UNWIND-PROTECT-STMT NODE))
          :CLEANUP
          (COPY-NODES (UNWIND-PROTECT-CLEANUP NODE))))

(DEFUN COPY-NODE-VAR-REF (NODE)
   (MAKE-VAR-REF :VARIABLE (FIND-COPIED-VARIABLE (VAR-REF-VARIABLE NODE))))

(DEFUN COPY-NODE-LIST (NODES)
   (IL:FOR NODE IL:IN NODES IL:COLLECT (COPY-NODES NODE)))

(DEFUN COPY-VARIABLE (VAR BINDER)
   (AND VAR (SETF (GETHASH VAR *COPY-NODE-TABLE*)
                  (MAKE-VARIABLE :NAME (VARIABLE-NAME VAR)
                         :SCOPE
                         (VARIABLE-SCOPE VAR)
                         :KIND
                         (VARIABLE-KIND VAR)
                         :BINDER BINDER))))

(DEFUN FIND-COPIED-VARIABLE (VAR)
   (IF (EQ :LEXICAL (VARIABLE-SCOPE VAR))
       (OR (GETHASH VAR *COPY-NODE-TABLE*)
           VAR)
       (COPY-VARIABLE VAR NIL)))

;; Arrange for the correct compiler to be used.

(IL:PUTPROPS IL:XCLC-TREES IL:FILETYPE :COMPILE-FILE)

;; Arrange for the correct makefile-environment

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

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

## FUNCTION INDEX



## NODE INDEX



## MACRO INDEX



## STRUCTURE INDEX



## VARIABLE INDEX



## CONSTANT INDEX



## PROPERTY INDEX



## DEFINE-TYPE INDEX



## DEFINER INDEX