File created: 9-Dec-91 03:12:25 **{PELE:MV:ENVOS}<LISPCORE>SOURCES>XCLC-META-EVAL.;5**

changes to: (IL:FUNCTIONS META-EVAL-LABELS)

previous date: 16-Aug-91 18:47:02 {PELE:MV:ENVOS}<LISPCORE>SOURCES>XCLC-META-EVAL.;4

Read Table: XCL

Package: COMPILER

Format: XCCS


```
(IL:RPAQQ IL:XCLC-META-EVALCOMS
          (
```

;;; Meta-evaluation

```
            (IL:FUNCTIONS META-EVALUATE)
            (IL:FUNCTIONS MEVAL MEVAL-LIST REDO-MEVAL)
            (IL:FUNCTIONS REMOVE-NESTED-PROGNS EXPAND-NESTED-PROGNS)
            (IL:FUNCTIONS GLOBAL-FUNCTION-P)
            (IL:FUNCTIONS CONSTRUCT-PROG1-TREE)
            (IL:VARIABLES *MADE-CHANGES* *REDOING-ANALYSIS*)
            (IL:FUNCTIONS META-EVAL-BLOCK META-EVAL-CALL META-EVAL-CATCH META-EVAL-GO META-EVAL-IF
                   META-EVAL-LABELS META-EVAL-LAMBDA META-EVAL-LITERAL META-EVAL-MV-CALL META-EVAL-MV-PROG1
                   META-EVAL-OPCODES META-EVAL-PROGN META-EVAL-PROGV META-EVAL-RETURN META-EVAL-SETQ
                   META-EVAL-TAGBODY META-EVAL-THROW META-EVAL-UNWIND-PROTECT META-EVAL-VAR-REF)
            (IL:FUNCTIONS META-CALL-LAMBDA META-CALL-LAMBDA-SIMPLIFY-PARAMETERS CONSTRUCT-LIST)
```

;;; Meta-substitution

```
            (IL:FUNCTIONS META-CALL-LAMBDA-SUBSTITUTE META-CALL-LABELS META-SUBSTITUTE META-SUBST)
            (IL:FUNCTIONS MSUBST MSUBST-LIST)
            (IL:FUNCTIONS SUBSTITUTABLE-P)
            (IL:FUNCTIONS EFFECTLESS EFFECTLESS-EXCEPT-CONS NULL-INTERSECTION NULL-EFFECTS-INTERSECTION
                   NULL-EFFECTS-INTERSECTION-EXCEPT-CONS PASSABLE NONLOCAL-VARIABLE-EFFECT-P
                   COLLECT-NONLOCAL-VAR-EFFECTS)
            (IL:VARIABLES *SUBST-VAR* *SUBST-EXPR* *SUBST-OCCURRED*)
            (IL:FUNCTIONS META-SUBST-BLOCK META-SUBST-CALL META-SUBST-CATCH META-SUBST-GO META-SUBST-IF
                   META-SUBST-LABELS META-SUBST-LAMBDA META-SUBST-LITERAL META-SUBST-MV-CALL META-SUBST-MV-PROG1
                   META-SUBST-OPCODES META-SUBST-PROGN META-SUBST-PROGV META-SUBST-RETURN META-SUBST-SETQ
                   META-SUBST-TAGBODY META-SUBST-THROW META-SUBST-UNWIND-PROTECT META-SUBST-VAR-REF)
            (IL:FUNCTIONS META-SUBST-ANY-CALL META-SUBST-STMTS)
            ;; Testing meta-evaluation
            (IL:FUNCTIONS TEST-META-EVAL)
            ;; Arrange to use the correct compiler
            (IL:PROP IL:FILETYPE IL:XCLC-META-EVAL)
            ;; Arrange for the proper makefile-environment
            (IL:PROP IL:MAKEFILE-ENVIRONMENT IL:XCLC-META-EVAL)))
```

;;; Meta-evaluation

(DEFUN **META-EVALUATE** (TREE &OPTIONAL (CONTEXT :ARGUMENT))

;;; Each meta-evaluation method takes in a subtree and mungs on it for a while, returning a (possibly identical) subtree. If the method makes no changes or can otherwise guarantee that no further meta-evaluation will be useful, it should set the meta-p field to the current context. This field must never be set if any node lower in the tree does not have it set. You can count on the fact that no node gets returned by meta-evaluate unless it has that field set.

;;; The CONTEXT argument has one of the following values:

;;; :effect -- No values of this expression are used.

;;; :argument -- Exactly one of the values of this expression is used.

;;; :mv -- More than one of the values of this expression may be used.

;;; :return -- The value(s) of this expression will be returned to the caller of the current function.

```
   (WHEN *REDOING-ANALYSIS*
       (SETF (NODE-META-P TREE)
             NIL))
   (IL:UNTIL (EQ CONTEXT (NODE-META-P TREE)) IL:DO (SETQ TREE (NODE-DISPATCH META-EVAL TREE CONTEXT)))
   TREE)
```

(DEFINE-MODIFY-MACRO **MEVAL** (CONTEXT) META-EVALUATE)

(DEFMACRO **MEVAL-LIST** (LIST-EXPR CONTEXT &OPTIONAL KEY-FN)

```
   `(IL:FOR TAIL IL:ON ,LIST-EXPR IL:DO (MEVAL ,(IF KEY-FN
                                                     `(,KEY-FN (CAR TAIL))
                                                     '(CAR TAIL))
                                                 ,CONTEXT)))


(DEFMACRO REDO-MEVAL (NODE-EXPR CONTEXT)
   `(LET ((*REDOING-ANALYSIS* T))
         (MEVAL ,NODE-EXPR ,CONTEXT)))


(DEFINE-MODIFY-MACRO REMOVE-NESTED-PROGNS () EXPAND-NESTED-PROGNS
```

;;; Replace the list of nodes in a given place with another list that has had its nested progns spliced inline. It is assumed that the nested PROGN nodes ;;; will not be referred to afterwards; this operation uses NCONC to splice the lists together.

```
   )


(DEFUN EXPAND-NESTED-PROGNS (NODE-LIST)
   (IL:FOR NODE IL:IN NODE-LIST IL:JOIN (IF (PROGN-P NODE)
                                            (PROGN-STMTS NODE)
                                            (LIST NODE))))


(DEFINLINE GLOBAL-FUNCTION-P (NODE)
   (AND (VAR-REF-P NODE)
        (LET ((VAR (VAR-REF-VARIABLE NODE)))
             (AND (EQ :GLOBAL (VARIABLE-SCOPE VAR))
                  (EQ :FUNCTION (VARIABLE-KIND VAR))))))


(DEFUN CONSTRUCT-PROG1-TREE (FIRST REST)
```

;;; Return a tree that represents a PROG1 in which FIRST's values are the ones that are wanted:

```
;;;      ((lambda (x) ,@REST x) FIRST)

   (LET* ((PROG1-VAR (MAKE-VARIABLE))
          (PROG1-VAR-REF (MAKE-VAR-REF :VARIABLE PROG1-VAR))
          (NEW-LAMBDA (MAKE-LAMBDA :REQUIRED (LIST PROG1-VAR)
                                   :BODY
                                   (MAKE-PROGN :STMTS (APPEND REST (LIST PROG1-VAR-REF))))))
         (SETF (VARIABLE-BINDER PROG1-VAR)
               NEW-LAMBDA)
         (PUSH PROG1-VAR-REF (VARIABLE-READ-REFS PROG1-VAR))
         (MAKE-CALL :FN NEW-LAMBDA :ARGS (LIST FIRST))))


(DEFVAR *MADE-CHANGES* NIL
```

;;; This variable is rebound in a few places in meta-evaluation in order to keep track of whether or not a large collection of routines has had any effect on ;;; the program tree.

```
   )


(DEFVAR *REDOING-ANALYSIS* NIL
```

;;; Bound to T when meta-evaluation is being redone. This forces the analysis to descend all the way back down to the bottom again.

```
   )


(DEFUN META-EVAL-BLOCK (NODE CONTEXT)
   (SETF (BLOCK-CONTEXT NODE)
         CONTEXT)
   (MEVAL (BLOCK-STMT NODE)
          CONTEXT)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)


(DEFUN META-EVAL-CALL (NODE CONTEXT)
   ;; First, meta-eval the subtrees.
   (MEVAL (CALL-FN NODE)
          :ARGUMENT)
   (MEVAL-LIST (CALL-ARGS NODE)
          :ARGUMENT)
   (ANALYZE-TREE NODE)
   (LET
    ((FN (CALL-FN NODE))
     TRANSFORM ARGS EVAL-WHEN-LOAD? LIST-OF-VALUES)
    (COND
```

```
        ;; Then, if it's a lambda-call, try hacking on it.
        ((LAMBDA-P FN)
         ;; But first, re-meta-evaluate the lambda-body since we now know the context in which it will be evaluated.
         (UNLESS (EQ CONTEXT :RETURN)
             (MEVAL (LAMBDA-BODY FN)
                    CONTEXT)
             (ANALYZE-TREE FN))
         (SETQ NODE (META-CALL-LAMBDA NODE CONTEXT)))
        ;; If it's a call to a side-effect-less function in effect context, blow it away.
        ;; Fix this to find local function side-effects.
        ((AND (EQ CONTEXT :EFFECT)
              (NOT (CALLER-NOT-INLINE NODE))
              (GLOBAL-FUNCTION-P FN)
              (LET ((EFFECTS (CAR (SIDE-EFFECTS (VARIABLE-NAME (VAR-REF-VARIABLE FN))))))
                   (OR (EQ EFFECTS :NONE)
                       (EQUAL EFFECTS '(:CONS)))))
         (SETQ NODE (PROG1 (MAKE-PROGN :STMTS (CALL-ARGS NODE))
                           (SETF (CALL-ARGS NODE)
                                 NIL)                          ; Detach the args, since they're still in the tree.
                           (RELEASE-TREE NODE))))
        ;; If it's a constant-foldable function and the arguments are all literals, fold it.  We must be careful because some of the literals might be
        ;; EVAL-WHEN-LOAD objects.  In that case, we bring the entire call inside a new EVAL-WHEN-LOAD.  This accomplishes, in effect, load-time
        ;; constant-folding.
        ((AND (NOT (CALLER-NOT-INLINE NODE))
              (GLOBAL-FUNCTION-P FN)
              (EQUAL '(:NONE . :NONE)
                     (SIDE-EFFECTS (VARIABLE-NAME (VAR-REF-VARIABLE FN))))
              (LISTP (SETQ ARGS (IL:FOR ARG IL:IN (CALL-ARGS NODE)
                                   IL:COLLECT (IF (NOT (LITERAL-P ARG))
                                                  (RETURN T)
                                                  (LET ((VALUE (LITERAL-VALUE ARG)))
                                                       (WHEN (EVAL-WHEN-LOAD-P VALUE)
                                                             (SETQ EVAL-WHEN-LOAD? T))
                                                       VALUE)))))
              (OR (EQ CONTEXT :ARGUMENT)
                  (NOT EVAL-WHEN-LOAD?)))
         (LET ((FN-NAME (VARIABLE-NAME (VAR-REF-VARIABLE FN))))
              (RELEASE-TREE NODE)
              (COND
                 ((EQ CONTEXT :ARGUMENT)
                  (SETQ NODE (MAKE-LITERAL
                              :VALUE
                              (IF EVAL-WHEN-LOAD?
                                  (MAKE-EVAL-WHEN-LOAD
                                   :FORM
                                   `(,FN-NAME ,@(IL:FOR ARG IL:IN ARGS
                                                   IL:COLLECT (IF (EVAL-WHEN-LOAD-P ARG)
                                                                  (EVAL-WHEN-LOAD-FORM ARG)
                                                                  `',ARG))))
                                  (APPLY FN-NAME ARGS)))))
                    ;; We're in either :RETURN or :MV context and the expression doesn't include any EVAL-WHEN-LOAD forms.
                    ((NULL (CDR (SETQ LIST-OF-VALUES (MULTIPLE-VALUE-LIST (APPLY FN-NAME ARGS)))))
                     ;; The form does not return multiple values.
                     (SETQ NODE (MAKE-LITERAL :VALUE (CAR LIST-OF-VALUES))))
                    (T ;; The form does return multiple values.  Turn it into a call on VALUES-LIST.
                       (SETQ NODE (MAKE-CALL :FN (MAKE-REFERENCE-TO-VARIABLE :NAME 'VALUES-LIST :KIND :FUNCTION
                                                                             :SCOPE :GLOBAL)
                                             :ARGS
                                             (LIST (MAKE-LITERAL :VALUE LIST-OF-VALUES))))))))
        ;; If there's a function-specific transformation defined for it, give it a try.
        ((AND (NOT (CALLER-NOT-INLINE NODE))
              (GLOBAL-FUNCTION-P FN)
              (SETQ TRANSFORM (GET (VARIABLE-NAME (VAR-REF-VARIABLE FN))
                                   'TRANSFORM)))
         (SETQ NODE (FUNCALL TRANSFORM NODE CONTEXT)))
        ;; Nothing to do, so we must be done.
        (T (SETF (NODE-META-P NODE)
                 CONTEXT)))
    NODE))

(DEFUN META-EVAL-CATCH (NODE CONTEXT)
   (MEVAL (CATCH-TAG NODE)
          :ARGUMENT)
   (MEVAL (CATCH-STMT NODE)
          CONTEXT)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
```

```
         CONTEXT)
   NODE)

(DEFUN META-EVAL-GO (NODE CONTEXT)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-IF (NODE CONTEXT)
   (MEVAL (IF-PRED NODE)
          :ARGUMENT)
   (MEVAL (IF-THEN NODE)
          CONTEXT)
   (MEVAL (IF-ELSE NODE)
          CONTEXT)
   (ANALYZE-TREE NODE)
   ;; If the predicate is a literal, we can eliminate one of the arms.
   (WHEN (LITERAL-P (IF-PRED NODE))
       (RETURN-FROM META-EVAL-IF (COND
                                     ((LITERAL-VALUE (IF-PRED NODE))
                                      (RELEASE-TREE (IF-ELSE NODE))
                                      (IF-THEN NODE))
                                     (T (RELEASE-TREE (IF-THEN NODE))
                                        (IF-ELSE NODE)))))
   ;; If both arms turned into literals and we're in effect context, then the only thing left is the predicate.
   (WHEN (AND (EQ CONTEXT :EFFECT)
              (LITERAL-P (IF-THEN NODE))
              (LITERAL-P (IF-ELSE NODE)))
       (RETURN-FROM META-EVAL-IF (IF-PRED NODE)))
   ;; If both arms have no side-effects and we're in effect context, then the only thing left is the predicate.
   (WHEN (AND (EQ CONTEXT :EFFECT)
              (EFFECTLESS-EXCEPT-CONS (NODE-EFFECTS (IF-THEN NODE)))
              (EFFECTLESS-EXCEPT-CONS (NODE-EFFECTS (IF-ELSE NODE))))
       (RETURN-FROM META-EVAL-IF (IF-PRED NODE)))
   ;; If the IF has the form (IF (IF pred NIL T) then else) then reduce to (IF pred else then).
   (LET ((PRED (IF-PRED NODE)))
        (WHEN (AND (IF-P PRED)
                   (LITERAL-P (IF-THEN PRED))
                   (EQ NIL (LITERAL-VALUE (IF-THEN PRED)))
                   (LITERAL-P (IF-ELSE PRED))
                   (EQ T (LITERAL-VALUE (IF-ELSE PRED))))
             (SETF (IF-PRED NODE)
                   (IF-PRED PRED))
             (ROTATEF (IF-THEN NODE)
                      (IF-ELSE NODE))))
   ;; Nothing worked, so give up.
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-LABELS (NODE CONTEXT)
   ;; First, meta-eval the subtrees:  The bodies of the FLET/LABELS-d functions must be meta-evaled first, so we get the side-effects info for
   ;; meta-evaluating the body correctly (e.g., to detect effect-free FLETd fns in effect context).  This fixes AR 11447 JDS 12/8/91
   (MEVAL-LIST (LABELS-FUNS NODE)
          :ARGUMENT CDR)
   (MEVAL (LABELS-BODY NODE)
          CONTEXT)
   (ANALYZE-TREE NODE)
   ;; Now try to substitute the local functions into the body.
   (SETQ NODE (META-CALL-LABELS NODE CONTEXT))
   NODE)

(DEFUN META-EVAL-LAMBDA (NODE CONTEXT)
   (WHEN (EQ CONTEXT :EFFECT)                                    ; LAMBDA's have no side effects.
       (RELEASE-TREE NODE)
       (RETURN-FROM META-EVAL-LAMBDA *LITERALLY-NIL*))
   (MEVAL (LAMBDA-BODY NODE)
          :RETURN)
   (MEVAL-LIST (LAMBDA-OPTIONAL NODE)
          :ARGUMENT SECOND)
   (MEVAL-LIST (LAMBDA-KEYWORD NODE)
          :ARGUMENT THIRD)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
```

```
   NODE)

(DEFUN META-EVAL-LITERAL (NODE CONTEXT)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-MV-CALL (NODE CONTEXT)

;;; Come back to this.  If all of the arg-exprs can be guaranteed to return a single value each, we can transform this in to a normal function call.
;;; Unfortunately, the information about how many values something might return is not passed up the tree.

   (MEVAL (MV-CALL-FN NODE)
          :ARGUMENT)
   (WHEN (LAMBDA-P (MV-CALL-FN NODE))
      ;; If the function is a LAMBDA, we should redo the analysis of the body because we now know the context in which it will be evaluated.
      (MEVAL (LAMBDA-BODY (MV-CALL-FN NODE))
             CONTEXT)
      (ANALYZE-TREE (MV-CALL-FN NODE)))
   (MEVAL-LIST (MV-CALL-ARG-EXPRS NODE)
          :MV)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-MV-PROG1 (NODE CONTEXT)
   (LET ((STMTS (MV-PROG1-STMTS NODE)))
        (MEVAL (FIRST STMTS)
               CONTEXT)
        (MEVAL-LIST (REST STMTS)
               :EFFECT)
        (REMOVE-NESTED-PROGNS (REST STMTS))
        (ECASE CONTEXT
            (:EFFECT                                                ; In effect context, we transform the MV-PROG1 into a simple
                                                                    ; PROGN.
               (MAKE-PROGN :STMTS STMTS))
            (:ARGUMENT                                              ; If multiple-values aren't wanted, turn this into a normal PROG1:
               (CONSTRUCT-PROG1-TREE (FIRST STMTS)
                      (REST STMTS)))
            ((:RETURN :MV)                                          ; Oh, well, it's MV context after all.  We can still try one last thing,
                                                                    ; though.
               (SETF (NODE-META-P NODE)
                     CONTEXT)
               (WHEN (PROGN-P (FIRST STMTS))                        ; Transform MV-PROG1 of PROGN into PROGN of MV-PROG1
                   (ROTATEF NODE (FIRST STMTS)
                            (CAR (LAST (PROGN-STMTS (FIRST STMTS))))))
               NODE))))

(DEFUN META-EVAL-OPCODES (NODE CONTEXT)

;;; Go ahead.  Meta-evaluate these opcodes.  I dare you.

   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-PROGN (NODE CONTEXT)

;;; Meta-evaluate the subtrees and then eliminate any nested PROGN's.

   (IL:FOR TAIL IL:ON (PROGN-STMTS NODE) IL:DO (IF (NULL (CDR TAIL))
                                                   (MEVAL (CAR TAIL)
                                                          CONTEXT)
                                                   (MEVAL (CAR TAIL)
                                                          :EFFECT)))
   (ANALYZE-TREE NODE)
   (REMOVE-NESTED-PROGNS (PROGN-STMTS NODE))
   ;; Eliminate any effect-context literals and reduce (PROGN <exp>) to <exp>.
   (LET ((NEW-STMTS (IL:FOR TAIL IL:ON (PROGN-STMTS NODE) IL:WHEN (OR (NOT (LITERAL-P (CAR TAIL)))
                                                                      (NULL (CDR TAIL)))
                       IL:COLLECT (CAR TAIL))))
        (COND
           ((NULL NEW-STMTS)
            *LITERALLY-NIL*)
           ((NULL (CDR NEW-STMTS))
            (SETQ NODE (CAR NEW-STMTS)))
           (T (SETF (PROGN-STMTS NODE)
```

```
                    NEW-STMTS))))
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-PROGV (NODE CONTEXT)
   (MEVAL (PROGV-SYMS-EXPR NODE)
          :ARGUMENT)
   (MEVAL (PROGV-VALS-EXPR NODE)
          :ARGUMENT)
   (MEVAL (PROGV-STMT NODE)
          CONTEXT)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-RETURN (NODE CONTEXT)
   (MEVAL (RETURN-VALUE NODE)
          (BLOCK-CONTEXT (RETURN-BLOCK NODE)))
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-SETQ (NODE CONTEXT)
   (MEVAL (SETQ-VALUE NODE)
          :ARGUMENT)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-TAGBODY (NODE CONTEXT)
   (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:DO (MEVAL-LIST (SEGMENT-STMTS SEGMENT)
                                                                   :EFFECT))
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-THROW (NODE CONTEXT)
   (MEVAL (THROW-TAG NODE)
          :ARGUMENT)
   (MEVAL (THROW-VALUE NODE)
          :MV)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-UNWIND-PROTECT (NODE CONTEXT)

;;; This is fucked up right now.

   (MEVAL (UNWIND-PROTECT-STMT NODE)
          :ARGUMENT)
   (MEVAL (UNWIND-PROTECT-CLEANUP NODE)
          :ARGUMENT)
   (ANALYZE-TREE NODE)
   (SETF (NODE-META-P NODE)
         CONTEXT)
   NODE)

(DEFUN META-EVAL-VAR-REF (NODE CONTEXT)
   (COND
      ((EQ CONTEXT :EFFECT)                                    ; Variable references have no side-effects.
       (RELEASE-VAR-REF NODE)
       *LITERALLY-NIL*)
      (T (ANALYZE-TREE NODE)
         (SETF (NODE-META-P NODE)
               CONTEXT)
         NODE)))

(DEFUN META-CALL-LAMBDA (NODE CONTEXT)
   ;; This is a directly-called LAMBDA.  Here are the steps to meta-evaluate it:
   ;;    -- Try to eliminate all of the non-required arguments
   ;;    -- Try to substitute in all of the required arguments
```

```
;;    -- Try to eliminate all arguments to unreferenced parameters
;;    -- Beta-convert the now-argument-less LAMBDA
(LET ((*MADE-CHANGES* NIL)
      (FN (CALL-FN NODE)))
     ;; We can't meta-evaluate direct calls to Interlisp's LAMBDA no-spread's.
     (WHEN (EQ 2 (LAMBDA-ARG-TYPE FN))
           (RETURN-FROM META-CALL-LAMBDA NODE))
     ;; If there are non-required parameters, try to get rid of them.
     (WHEN (OR (LAMBDA-OPTIONAL FN)
               (LAMBDA-KEYWORD FN)
               (LAMBDA-REST FN)
               (/= (LENGTH (CALL-ARGS NODE))
                   (LENGTH (LAMBDA-REQUIRED FN))))          ; Attempt to eliminate all of the non-required parameters.  Also,
                                                            ; catch wrong number of arguments errors.
         (WHEN (NULL (META-CALL-LAMBDA-SIMPLIFY-PARAMETERS NODE))
             (WHEN (NULL *MADE-CHANGES*)
                 (SETF (NODE-META-P NODE)
                       CONTEXT))
             (RETURN-FROM META-CALL-LAMBDA NODE))
         ;; Some changes were made.  We need to re-meta-evaluate the call in order to make sure it's up to date.
         (MEVAL (CALL-FN NODE)
                :ARGUMENT)
         (MEVAL-LIST (CALL-ARGS NODE)
                :ARGUMENT))
     ;; Now there are only required parameters.  Try to substitute arguments for parameters where appropriate.
     (META-CALL-LAMBDA-SUBSTITUTE NODE)
     ;; Now we can get rid of any parameters that aren't referenced.
     (LET ((NEW-PARAMS NIL)
           (NEW-ARGS NIL)
           (CURRENT-PROG1-TAIL NIL)
           (CURRENT-PROGN NIL))
          ;; If the first parameter is not referenced, we can turn (fn arg1 . args) into (progn arg1 (fn' . args)) where fn' is just like fn but doesn't
          ;; have the first parameter.
          ;; If the first is referenced, but the second is not, we can turn (fn arg1 arg2 . args) into (fn' (prog1 arg1 arg2) .args), where fn' is just like
          ;; fn but without the second parameter and (prog1 (a) . b) is really ((lambda (anon) ,@b anon) (a)) .
          (IL:FOR PARAM IL:IN (LAMBDA-REQUIRED FN) IL:AS ARG IL:IN (CALL-ARGS NODE)
             IL:DO (COND
                       ((OR (EQ :SPECIAL (VARIABLE-SCOPE PARAM))
                            (NOT (NULL (VARIABLE-READ-REFS PARAM)))
                            (NOT (NULL (VARIABLE-WRITE-REFS PARAM))))
                                                            ; This one's used.  Leave it.
                        (PUSH PARAM NEW-PARAMS)
                        (PUSH ARG NEW-ARGS)
                        (SETQ CURRENT-PROG1-TAIL NIL))
                       (T                                   ; This one is not used.  Can it.
                          (SETQ *MADE-CHANGES* T)
                          (COND
                             ((NODE-SUBST-P ARG)            ; The corresponding argument has been substituted into the
                                                            ; body.  We need not save it at all.
                              (RELEASE-TREE ARG))
                             ((NULL NEW-ARGS)               ; It's an early one.  Stick it into a PROGN.
                              (PUSH ARG CURRENT-PROGN))
                             ((NULL CURRENT-PROG1-TAIL)     ; We need to set up a PROG1 using the previous argument.
                                                            ; After putting it together, we make CURRENT-PROG1-TAIL
                                                            ; hold the last CONS cell in the body of the LAMBDA of the
                                                            ; PROG1.  This is to allow splicing new stmts into that body.
                              (LET* ((NEW-PROG1 (CONSTRUCT-PROG1-TREE (CAR NEW-ARGS)
                                                                      (LIST ARG))))
                                    (SETQ CURRENT-PROG1-TAIL (CDR (PROGN-STMTS (LAMBDA-BODY (CALL-FN NEW-PROG1
                                                                                                          )))))
                                    (SETF (CAR NEW-ARGS)
                                          NEW-PROG1)))
                             (T                             ; There's already a PROG1 set up for us.
                                (IL:RPLNODE CURRENT-PROG1-TAIL ARG (LIST (CAR CURRENT-PROG1-TAIL)))
                                (SETQ CURRENT-PROG1-TAIL (CDR CURRENT-PROG1-TAIL)))))))
          ;; If there aren't any arguments left, then we can beta-reduce.
          (COND
             ((NULL NEW-ARGS)
              (SETQ NODE (LAMBDA-BODY FN))
              (SETF (LAMBDA-BODY FN)
                    NIL)
              (SETF (LAMBDA-REQUIRED FN)
                    NIL)
              (RELEASE-TREE FN)
              (SETQ *MADE-CHANGES* T))
             (T (SETF (LAMBDA-REQUIRED FN)
                      (NREVERSE NEW-PARAMS))
                (SETF (CALL-ARGS NODE)
```

```
                        (NREVERSE NEW-ARGS))))
             (WHEN (NOT (NULL CURRENT-PROGN))
                 (SETQ NODE (MAKE-PROGN :STMTS (NREVERSE (CONS NODE CURRENT-PROGN))))))
         (WHEN (NULL *MADE-CHANGES*)
             (SETF (NODE-META-P NODE)
                   CONTEXT))
         NODE))
```

(DEFUN **META-CALL-LAMBDA-SIMPLIFY-PARAMETERS** (NODE)

;; Attempt to eliminate all of the non-required parameters from the given lambda-call. Return non-nil iff we can get rid of all of them. Also, check for ;; wrong number of arguments.

```
(LET* ((FN (CALL-FN NODE))
       (ARGS (CALL-ARGS NODE))
       (INNERMOST-CALL NODE)
       (INNERMOST-LAMBDA FN)
       (NEW-ARGS NIL)
       (NEW-PARAMS NIL)
       (KEY-PARAMS NIL))
      (LABELS ((ADD-PARAM (PARAM ARG)
```

;; Match up the given argument with the given parameter in the current lambda.

```
                       (PUSH PARAM NEW-PARAMS)
                       (PUSH ARG NEW-ARGS)
                       (SETF (VARIABLE-BINDER PARAM)
                             INNERMOST-LAMBDA))
               (CLOSE-LAMBDA NIL
```

;; Close off the current lambda.

```
                       (SETF (LAMBDA-REQUIRED INNERMOST-LAMBDA)
                             (NREVERSE NEW-PARAMS))
                       (SETF (CALL-ARGS INNERMOST-CALL)
                             (NREVERSE NEW-ARGS))
                       (SETQ NEW-ARGS NIL)
                       (SETQ NEW-PARAMS NIL))
               (NEW-LAMBDA NIL
```

;; Close off the old lambda and add a new one inside of it.

```
                       (CLOSE-LAMBDA)
                       (LET* ((NEW-LAMBDA (MAKE-LAMBDA :BODY (LAMBDA-BODY INNERMOST-LAMBDA)))
                              (NEW-CALL (MAKE-CALL :FN NEW-LAMBDA)))
                             (SETF (LAMBDA-BODY INNERMOST-LAMBDA)
                                   NEW-CALL)
                             (SETQ INNERMOST-LAMBDA NEW-LAMBDA)
                             (SETQ INNERMOST-CALL NEW-CALL)))
               (OUTER-LAMBDA-P NIL
```

;; Is the current lambda the outermost one?

```
                       (EQ INNERMOST-LAMBDA FN))
               (ENSURE-INNER-LAMBDA NIL
```

;; Make sure that the current lambda is not the outermost one.

```
                       (WHEN (OUTER-LAMBDA-P)
                             (NEW-LAMBDA))))
```

;; Handle the required parameters

```
(IL:FOR TAIL IL:ON (LAMBDA-REQUIRED FN) IL:DO (COND
                                                  ((NULL ARGS)
                                                   (CERROR "Use NIL for the remaining
                                                          parameters." "Too few arguments given
                                                          for explicit LAMBDA call.")
                                                   (IL:FOR PARAM IL:IN TAIL
                                                      IL:DO (ADD-PARAM PARAM *LITERALLY-NIL*))
                                                   (RETURN))
                                                  (T (ADD-PARAM (CAR TAIL)
                                                                (POP ARGS)))))
```

;; Handle the optional parameters

```
(WHEN (NOT (NULL (LAMBDA-OPTIONAL FN)))
    (IL:FOR OPT-VAR IL:IN (LAMBDA-OPTIONAL FN) IL:DO (COND
                                                         ((NULL ARGS)
```

;; No arguments left. Wrap the body in a LET binding the optional parameter to its default value.
;; Also bind the supplied-p parameter, if any.

```
                                                          (NEW-LAMBDA)
                                                          (ADD-PARAM (FIRST OPT-VAR)
                                                                 (SECOND OPT-VAR))
                                                          (WHEN (THIRD OPT-VAR)
                                                          ; There's a supplied-p
                                                              (ADD-PARAM (THIRD OPT-VAR)
                                                                     *LITERALLY-NIL*)))
                                                         (T
```

;; There are arguments left, so match one up with this optional and match up T
;; with the supplied-p parameter, if any.

```
                                                             (ADD-PARAM (FIRST OPT-VAR)
```

```
                                                                    (POP ARGS))
                                                               (RELEASE-TREE (SECOND OPT-VAR))
                                                               (WHEN (THIRD OPT-VAR)
                                                          ; There's a supplied-p
                                                                    (ADD-PARAM (THIRD OPT-VAR)
                                                                           *LITERALLY-T*)))))
    (SETF (LAMBDA-OPTIONAL FN)
          NIL)                                            ; All of the optionals are gone now.
    (SETF (NODE-META-P FN)
          NIL)
    (SETQ *MADE-CHANGES* T))
```

;; We can't go any further if there are keyword parameters and we can't tell what the corresponding keyword arguments are.

```
(WHEN (AND (NOT (NULL (LAMBDA-KEYWORD FN)))
           (NOT (IL:FOR ARG IL:IN ARGS IL:BY CDDR IL:ALWAYS (LITERAL-P ARG))))
    (CLOSE-LAMBDA)
    (RETURN-FROM META-CALL-LAMBDA-SIMPLIFY-PARAMETERS NIL))
```

;; If there are keyword parameters, we need to bind the remaining arguments to new, anonymous variables for any more
;; processing.

```
(WHEN (NOT (NULL (LAMBDA-KEYWORD FN)))
    (ENSURE-INNER-LAMBDA)
    (SETQ KEY-PARAMS (IL:IN ARGS IL:COLLECT (MAKE-VARIABLE :BINDER FN)))
    (WHEN (NOT (NULL KEY-PARAMS))
        (SETF (LAMBDA-REQUIRED FN)
              (APPEND (LAMBDA-REQUIRED FN)
                      KEY-PARAMS))
        (SETF (CALL-ARGS NODE)
              (APPEND (CALL-ARGS NODE)
                      ARGS)))
    (SETF (NODE-META-P FN)
          NIL)
    (SETQ *MADE-CHANGES* T))
```

;; Handle the &rest parameter, if any. If there are keyword parameters, then we bind the rest-var to a list of the anonymous
;; variables used in that translation. If not, we transform the arguments into a single call on LIST and bind that to the rest-var.

```
(LET ((REST-VAR (LAMBDA-REST FN)))
     (WHEN (NOT (NULL REST-VAR))
         (COND
            ((NOT (NULL KEY-PARAMS))                  ; There are keyword parameters.
             (ADD-PARAM REST-VAR (CONSTRUCT-LIST
                                  (MAPCAR #'(LAMBDA (PARAM)
                                                    (LET ((REF (MAKE-VAR-REF :VARIABLE PARAM)))
                                                         (PUSH REF (VARIABLE-READ-REFS PARAM))
                                                        REF))
                                          KEY-PARAMS))))
            (T                                        ; There are no keyword parameters.
               (COND
                  ((NULL ARGS)                        ; There aren't any arguments left, either.
                   (ADD-PARAM REST-VAR *LITERALLY-NIL*))
                  ((OUTER-LAMBDA-P)                   ; We're still in the outer lambda, so we can just bind the rest-var
                                                      ; to a list of the arguments.
                   (ADD-PARAM REST-VAR (CONSTRUCT-LIST ARGS)))
                  (T                                  ; Sigh. This is the messiest case. We're in an inner lambda, so
                                                      ; we have to add an anonymous variable to the outer one, bind
                                                      ; that to the list of arguments, and then bind the rest-var to that in
                                                      ; the current lambda.
                     (LET* ((ANON-VAR (MAKE-VARIABLE :BINDER FN))
                            (ANON-VAR-REF (MAKE-VAR-REF :VARIABLE ANON-VAR)))
                           (SETF (LAMBDA-REQUIRED FN)
                                 (NCONC (LAMBDA-REQUIRED FN)
                                        (LIST ANON-VAR)))
                           (SETF (CALL-ARGS NODE)
                                 (NCONC (CALL-ARGS NODE)
                                        (LIST (CONSTRUCT-LIST ARGS))))
                           (ADD-PARAM REST-VAR ANON-VAR-REF)
                           (PUSH ANON-VAR-REF (VARIABLE-READ-REFS ANON-VAR)))))
               (SETQ ARGS NIL)                        ; All of the arguments have been handled.

               ))
         (SETF (LAMBDA-REST FN)
               NIL)                                   ; The &rest parameter is no more.
         (SETF (NODE-META-P FN)
               NIL)
         (SETQ *MADE-CHANGES* T)))
```

;; Handle the keyword parameters. All of the keyword-position arguments are literals; we can thus determine which arguments go
;; with which keyword parameters. Thus, we can turn them all into required ones.

```
(WHEN (NOT (NULL (LAMBDA-KEYWORD FN)))
    (IL:FOR KEY-VAR IL:IN (LAMBDA-KEYWORD FN)
       IL:DO (LET* ((KEYWORD (FIRST KEY-VAR))
                    (ANON-VAR (IL:FOR ARG IL:IN ARGS IL:BY CDDR IL:AS TAIL IL:ON KEY-PARAMS
                                      IL:BY CDDR IL:WHEN (EQ KEYWORD (LITERAL-VALUE ARG))
                                      IL:DO (RETURN (SECOND TAIL))))
                    (ANON-VAR-REF (MAKE-VAR-REF :VARIABLE ANON-VAR)))
                   (COND
                      ((NULL ANON-VAR)
```

```
                                        ;; This keyword isn't present in the call; treat the same as an unsupplied optional.
                                        (NEW-LAMBDA)
                                        (ADD-PARAM (SECOND KEY-VAR)
                                               (THIRD KEY-VAR))
                                        (WHEN (FOURTH KEY-VAR)
                                            (ADD-PARAM (FOURTH KEY-VAR)
                                                   *LITERALLY-NIL*)))
                                       (T ;; There is a matching keyword in the argument list, so we bind the parameter to the
                                          ;; corresponding anonymous one.
                                          (ENSURE-INNER-LAMBDA)
                                          (ADD-PARAM (SECOND KEY-VAR)
                                                 ANON-VAR-REF)
                                          (PUSH ANON-VAR-REF (VARIABLE-READ-REFS ANON-VAR))
                                          (RELEASE-TREE (THIRD KEY-VAR))
                                          (WHEN (FOURTH KEY-VAR)
                                              (ADD-PARAM (FOURTH KEY-VAR)
                                                     *LITERALLY-T*))))))
                    (SETF (LAMBDA-KEYWORD FN)
                          NIL)                                    ; All of the keyword parameters are gone now.
                    (SETQ ARGS NIL)                               ; And we've managed to use up all of the arguments.

                    )
               ;; Make one final check that there weren't too many arguments.
               (WHEN (NOT (NULL ARGS))
                   (CERROR "Ignore the extra arguments" "Too many arguments were given to an inline LAMBDA
                           call.")
                   (IL:FOR ARG IL:IN ARGS IL:DO (RELEASE-TREE ARG)))
               ;; We're done now. Close up the inner lambda machinery and return a sign of success.
               (CLOSE-LAMBDA)
               T)))
```

(DEFUN **CONSTRUCT-LIST** (ARGS)

;;; ARGS is a non-empty list of nodes. Return an tree that computes the result of LIST applied to the results of those nodes. A simple implementation
;;; would just make an actual call to the function LIST, but this is inefficient. Instead, we make a nested series of calls to the function CONS.

```
   (LET ((NODE *LITERALLY-NIL*))
        (IL:MAPC (REVERSE ARGS)
               (IL:FUNCTION (IL:LAMBDA (ARG)
                             (SETQ NODE (MAKE-CALL :FN (MAKE-REFERENCE-TO-VARIABLE :KIND :FUNCTION :SCOPE
                                                                                   :GLOBAL :NAME 'CONS)
                                                   :ARGS
                                                   (LIST ARG NODE))))))
        NODE))
```

;;; Meta-substitution

```
(DEFUN META-CALL-LAMBDA-SUBSTITUTE (NODE)
   (LET* ((FN (CALL-FN NODE))
          (REQUIRED-ARGS (LAMBDA-REQUIRED FN))
          (NON-LOCAL-EFFECTS (WITH-COLLECTION (DOLIST (VAR REQUIRED-ARGS)
                                                      (UNLESS (EQ (VARIABLE-SCOPE VAR)
                                                                  :LEXICAL)
                                                          (COLLECT (EFFECTS-REPRESENTATION VAR))))))
          (*SUBST-OCCURRED* NIL))                                  ; Bind *SUBST-OCCURRED* just so that
                                                                   ; META-SUBST-VAR-REF has a binding to set even when
                                                                   ; nobody cares.
         (IL:FOR VAR IL:IN REQUIRED-ARGS IL:AS TAIL IL:ON (CALL-ARGS NODE)
            IL:WHEN (AND (EQ (VARIABLE-SCOPE VAR)
                             :LEXICAL)
                         (SUBSTITUTABLE-P (CAR TAIL)
                                 VAR)
                         (DOLIST (NON-LOCAL-EFFECT NON-LOCAL-EFFECTS T)
                             (UNLESS (NULL-EFFECTS-INTERSECTION NON-LOCAL-EFFECT (NODE-AFFECTED (CAR TAIL)))
                                     (RETURN NIL)))
                         (DOLIST (LATER-ARG (CDR TAIL)
                                            T)
                             (WHEN (NOT (PASSABLE (CAR TAIL)
                                                  LATER-ARG))
                                   (RETURN NIL))))
            IL:DO (SETF (LAMBDA-BODY FN)
                        (META-SUBSTITUTE (CAR TAIL)
                              VAR
                              (LAMBDA-BODY FN))))
         (WHEN (NULL (NODE-META-P (LAMBDA-BODY FN)))
             (SETF (NODE-META-P FN)
                   NIL)
             (SETQ *MADE-CHANGES* T))))
```

```
(DEFUN META-CALL-LABELS (NODE CONTEXT)
    ;; This is similar to META-CALL-LAMBDA, but we have some extra information. There are only required arguments, and we have the correct number
    ;; of them.
    (LET ((*MADE-CHANGES* NIL))
         ;; First, substitute the functions wherever possible.
         (DOLIST (FN-PAIR (LABELS-FUNS NODE)
                        (WHEN (NULL (NODE-META-P (LABELS-BODY NODE)))
                            (SETF (NODE-META-P NODE)
                                  NIL)
                            (SETQ *MADE-CHANGES* T)))
             (WHEN (SUBSTITUTABLE-P (CDR FN-PAIR)
                                    (CAR FN-PAIR))
                 (LET ((*SUBST-OCCURRED* NIL))
                      ;; First try substituting into the body.
                      (SETF (LABELS-BODY NODE)
                            (META-SUBSTITUTE (CDR FN-PAIR)
                                   (CAR FN-PAIR)
                                   (LABELS-BODY NODE)))
                      (WHEN (NOT *SUBST-OCCURRED*)
                           ;; Wasn't in the body - try the other functions.
                           (DOLIST (TARGET-PAIR (LABELS-FUNS NODE))
                               (UNLESS (EQ TARGET-PAIR FN-PAIR)
                                   (SETF (CDR TARGET-PAIR)
                                         (META-SUBSTITUTE (CDR FN-PAIR)
                                                 (CAR FN-PAIR)
                                                 (CDR TARGET-PAIR)))
                                   (WHEN *SUBST-OCCURRED*          ; Found it, we can stop now.
                                       (SETF (NODE-META-P NODE)
                                             NIL)
                                       (SETQ *MADE-CHANGES* T)
                                       (RETURN)))))
                      ;; May need to reanalyze the node, since things might have changed.  Note that reanalyzing the parts of the node this way
                      ;; means the the state in the enclosing loop is not lost.
                      (DOLIST (FNS (LABELS-FUNS NODE))
                          (MEVAL (CDR FNS)
                                 :ARGUMENT))
                      (MEVAL (LABELS-BODY NODE)
                             :RETURN))))
         ;; Now remove any functions that aren't referenced.
         (DOLIST (FN-PAIR (PROG1 (LABELS-FUNS NODE)
                                 (SETF (LABELS-FUNS NODE)
                                       NIL)))
             (COND
                ((NULL (VARIABLE-READ-REFS (CAR FN-PAIR)))
                 (RELEASE-TREE (CDR FN-PAIR))
                 (SETQ *MADE-CHANGES* T))
                (T (PUSH FN-PAIR (LABELS-FUNS NODE)))))
         ;; If there aren't any functions left, replace the node with its body.
         (WHEN (NULL (LABELS-FUNS NODE))
             (LET ((BODY (LABELS-BODY NODE)))
                  (SETF (LABELS-BODY NODE)
                        NIL)
                  (RELEASE-TREE NODE)
                  (SETQ NODE BODY *MADE-CHANGES* T)))
         ;; Finally, set the meta-p flag if everythings OK.
         (IF (NULL *MADE-CHANGES*)
             (SETF (NODE-META-P NODE)
                   CONTEXT)
             (SETF (NODE-META-P NODE)
                   NIL)))
   NODE)


(DEFUN META-SUBSTITUTE (*SUBST-EXPR* *SUBST-VAR* TREE)
   (META-SUBST TREE))


(DEFUN META-SUBST (NODE)
   (NODE-DISPATCH META-SUBST NODE))


(DEFINE-MODIFY-MACRO MSUBST (CONTEXT) META-SUBST)


(DEFMACRO MSUBST-LIST (LIST-EXPR CONTEXT &OPTIONAL KEY-FN)
   `(IL:FOR TAIL IL:ON ,LIST-EXPR IL:DO (MSUBST ,(IF KEY-FN
                                                     `(,KEY-FN (CAR TAIL))
                                                     '(CAR TAIL))
                                                ,CONTEXT)))
```

```
(DEFUN SUBSTITUTABLE-P (ARG VAR)

;;; Should ARG be substituted for all of the references to VAR?

;;; This test is very conservative, but still catches an enormous number of cases in practice.

;;; NOTEZ BIEN: If you change this test, be sure to look carefully at the various substitution methods and at the code in META-CALL-LAMBDA.  Some of
;;; it depends upon the precise test being made; in particular, it matters if more side effects are allowed to be substituted.

   (AND (NULL (VARIABLE-WRITE-REFS VAR))                              ; The variable is never SETQ'd in the body,
        (NOT (NULL (VARIABLE-READ-REFS VAR)))                         ; the variable is read at least once,
                                                                      ; and either
        (OR (VAR-REF-P ARG)                                           ;  -- the arg is a variable reference,
            (AND (LITERAL-P ARG)
                 (NOT (EVAL-WHEN-LOAD-P (LITERAL-VALUE ARG))))        ;  -- the arg is a literal

            )))


(DEFUN EFFECTLESS (EFFECTS)
   (OR (NULL EFFECTS)
       (EQ EFFECTS :NONE)))


(DEFUN EFFECTLESS-EXCEPT-CONS (EFFECTS)
   (OR (NULL EFFECTS)
       (EQ EFFECTS :NONE)
       (EQUAL EFFECTS '(:CONS))))


(DEFUN NULL-INTERSECTION (LIST-1 LIST-2)
   (OR (NULL LIST-1)
       (NULL LIST-2)
       (DOLIST (X LIST-1 T)
           (WHEN (MEMBER X LIST-2 :TEST 'EQ)
                 (RETURN NIL)))))


(DEFUN NULL-EFFECTS-INTERSECTION (EFFECTS-1 EFFECTS-2)
   (OR (NULL EFFECTS-1)
       (NULL EFFECTS-2)
       (EQ EFFECTS-1 :NONE)
       (EQ EFFECTS-2 :NONE)
       (COND
          ((EQ EFFECTS-1 :ANY)
           NIL)
          ((EQ EFFECTS-2 :ANY)
           NIL)
          (T (NULL-INTERSECTION EFFECTS-1 EFFECTS-2)))))


(DEFUN NULL-EFFECTS-INTERSECTION-EXCEPT-CONS (EFFECTS-1 EFFECTS-2)
   (COND
      ((OR (NULL EFFECTS-1)
           (NULL EFFECTS-2)
           (EQ EFFECTS-1 :NONE)
           (EQ EFFECTS-2 :NONE))
       T)
      ((EQ EFFECTS-1 :ANY)
       (IL:EQUAL (IL:MKLIST EFFECTS-2 '(:CONS))))
      ((EQ EFFECTS-2 :ANY)
       (IL:EQUAL (IL:MKLIST EFFECTS-1 '(:CONS))))
      (T (DOLIST (EFFECT EFFECTS-1 T)
             (WHEN (AND (NOT (EQ EFFECT :CONS))
                        (MEMBER EFFECT EFFECTS-2 :TEST 'EQ))
                   (RETURN NIL))))))


(DEFUN PASSABLE (NODE-1 NODE-2)

;;; This predicate is true if and only if the two given nodes can be executed in either order.

   (AND (NULL-EFFECTS-INTERSECTION (NODE-EFFECTS NODE-1)
               (NODE-AFFECTED NODE-2))
        (NULL-EFFECTS-INTERSECTION (NODE-AFFECTED NODE-1)
               (NODE-EFFECTS NODE-2))
        (NULL-EFFECTS-INTERSECTION-EXCEPT-CONS (NODE-EFFECTS NODE-1)
               (NODE-EFFECTS NODE-2))))


(DEFINLINE NONLOCAL-VARIABLE-EFFECT-P (EFFECT-REP)
   (AND (SYMBOLP EFFECT-REP)
        (NOT (KEYWORDP EFFECT-REP))))
```

```
(DEFUN COLLECT-NONLOCAL-VAR-EFFECTS (NODE)
   (LET ((VARS NIL))
        (DOLIST (EFFECT (IL:MKLIST (NODE-EFFECTS NODE)))
            (WHEN (NONLOCAL-VARIABLE-EFFECT-P EFFECT)
                (PUSHNEW EFFECT VARS :TEST 'EQ)))
        (DOLIST (EFFECT (IL:MKLIST (NODE-AFFECTED NODE)))
            (WHEN (NONLOCAL-VARIABLE-EFFECT-P EFFECT)
                (PUSHNEW EFFECT VARS :TEST 'EQ)))
        VARS))

(DEFVAR *SUBST-VAR* NIL

;;; The variable for occurrences of which we are substituting *SUBST-EXPR*.

   )

(DEFVAR *SUBST-EXPR* NIL

;;; The expression being substituted for all occurrences of *SUBST-VAR*.

   )

(DEFVAR *SUBST-OCCURRED*

;;; Bound by substitution methods that need to know whether or not anything actually happened and set by META-SUBST-VAR-REF when something
;;; does.

   )

(DEFUN META-SUBST-BLOCK (NODE)                                  ;
   (MSUBST (BLOCK-STMT NODE))
   (SETF (NODE-META-P NODE)
         (NODE-META-P (BLOCK-STMT NODE)))
   NODE)

(DEFUN META-SUBST-CALL (NODE)
   (WHEN (AND (NOT (LAMBDA-P (CALL-FN NODE)))
              (NOT (CALLER-NOT-INLINE NODE)))                   ; The body of the lambda won't be eval'd until after the
                                                                ; arguments.
       (MSUBST (CALL-FN NODE)))
   (META-SUBST-ANY-CALL NODE (CALL-FN NODE)
          (CALL-ARGS NODE)))

(DEFUN META-SUBST-CATCH (NODE)                                  ;
   (MSUBST (CATCH-TAG NODE))
   (WHEN (PASSABLE *SUBST-EXPR* (CATCH-TAG NODE))
       (MSUBST (CATCH-STMT NODE)))
   (WHEN (OR (NULL (NODE-META-P (CATCH-TAG NODE)))
             (NULL (NODE-META-P (CATCH-STMT NODE))))
       (SETF (NODE-META-P NODE)
             NIL))
   NODE)

(DEFUN META-SUBST-GO (NODE)
   NODE)

(DEFUN META-SUBST-IF (NODE)                                     ;
   (MSUBST (IF-PRED NODE))
   (WHEN (AND (EFFECTLESS-EXCEPT-CONS (NODE-EFFECTS *SUBST-EXPR*))
              (PASSABLE *SUBST-EXPR* (IF-PRED NODE)))
       (MSUBST (IF-THEN NODE))
       (MSUBST (IF-ELSE NODE)))
   (WHEN (OR (NULL (NODE-META-P (IF-PRED NODE)))
             (NULL (NODE-META-P (IF-THEN NODE)))
             (NULL (NODE-META-P (IF-ELSE NODE))))
       (SETF (NODE-META-P NODE)
             NIL))
   NODE)

(DEFUN META-SUBST-LABELS (NODE)
   (DOLIST (FUN (LABELS-FUNS NODE))
       (MSUBST (CDR FUN))
       (WHEN (NULL (NODE-META-P (CDR FUN)))
           (SETF (NODE-META-P NODE)
                 NIL)))
   (MSUBST (LABELS-BODY NODE))
   (WHEN (NULL (NODE-META-P (LABELS-BODY NODE)))
       (SETF (NODE-META-P NODE)
```

```
            NIL))
   NODE)


(DEFUN META-SUBST-LAMBDA (NODE &OPTIONAL (IN-FUNCTIONAL-POSITION NIL))
   (WHEN (OR IN-FUNCTIONAL-POSITION (AND (EFFECTLESS (NODE-EFFECTS *SUBST-EXPR*))
                                         (EFFECTLESS (NODE-AFFECTED *SUBST-EXPR*))))
       (LET ((NONLOCAL-VAR-EFFECTS (COLLECT-NONLOCAL-VAR-EFFECTS *SUBST-EXPR*)))
            (FLET ((SPECIAL-CLASHES-WITH-EFFECTS (VAR)
                       ;; This is to check for the case of substituting an expression which depends on a special variable into a scope that
                       ;; rebinds that special.
                       (AND (EQ (VARIABLE-SCOPE VAR)
                                :SPECIAL)
                            (IL:FMEMB (VARIABLE-NAME VAR)
                                      NONLOCAL-VAR-EFFECTS))))
                  (BLOCK SUBSTITUTION
                      (DOLIST (REQ-VAR (LAMBDA-REQUIRED NODE))
                          (WHEN (SPECIAL-CLASHES-WITH-EFFECTS REQ-VAR)
                                (RETURN-FROM SUBSTITUTION)))
                      (DOLIST (OPT-VAR (LAMBDA-OPTIONAL NODE))
                          (MSUBST (SECOND OPT-VAR))
                          (WHEN (NULL (NODE-META-P (SECOND OPT-VAR)))
                              (SETF (NODE-META-P NODE)
                                    NIL))
                          (WHEN (OR (SPECIAL-CLASHES-WITH-EFFECTS (FIRST OPT-VAR))
                                    (NOT (PASSABLE *SUBST-EXPR* (SECOND OPT-VAR))))
                                (RETURN-FROM SUBSTITUTION)))
                      (WHEN (AND (LAMBDA-REST NODE)
                                 (SPECIAL-CLASHES-WITH-EFFECTS (LAMBDA-REST NODE)))
                        (RETURN-FROM SUBSTITUTION))
                      ;; JDS 1/6/89:  This appears to loop thru the keywords, checking each one.  There uesd to be a
                      ;; (SPECIAL-CLASHES-WITH-EFFECTS (THIRD KEY-VAR)) in the (when (or...)) clause, but that list seems to be
                      ;; ( variable-symbol variable-structure var's-value-structure)
                      ;; and that's not checkable with SPECIAL-CLASHES-WITH-EFFECTS....
                      (DOLIST (KEY-VAR (LAMBDA-KEYWORD NODE))
                          (MSUBST (THIRD KEY-VAR))
                          (WHEN (NULL (NODE-META-P (THIRD KEY-VAR)))
                              (SETF (NODE-META-P NODE)
                                    NIL))
                          (WHEN (OR (SPECIAL-CLASHES-WITH-EFFECTS (SECOND KEY-VAR))
                                    (NOT (PASSABLE *SUBST-EXPR* (THIRD KEY-VAR))))
                                (RETURN-FROM SUBSTITUTION)))
                      (MSUBST (LAMBDA-BODY NODE))
                      (WHEN (NULL (NODE-META-P (LAMBDA-BODY NODE)))
                          (SETF (NODE-META-P NODE)
                                NIL))))))
   NODE)


(DEFUN META-SUBST-LITERAL (NODE)
   NODE)


(DEFUN META-SUBST-MV-CALL (NODE)
   (WHEN (AND (NOT (LAMBDA-P (MV-CALL-FN NODE)))
              (NOT (CALLER-NOT-INLINE NODE)))                    ; The body of the lambda won't be eval'd until after the
                                                                 ; arguments.
       (MSUBST (MV-CALL-FN NODE)))
   (META-SUBST-ANY-CALL NODE (MV-CALL-FN NODE)
          (MV-CALL-ARG-EXPRS NODE)))


(DEFUN META-SUBST-MV-PROG1 (NODE)
   (DESTRUCTURING-BIND (VALUES-STMT . EFFECT-STMTS)
          (MV-PROG1-STMTS NODE)
          (MSUBST VALUES-STMT)
          (WHEN (NULL (NODE-META-P VALUES-STMT))
              (SETF (NODE-META-P NODE)
                    NIL))
          (WHEN (PASSABLE *SUBST-EXPR* VALUES-STMT)
              (SETQ EFFECT-STMTS (META-SUBST-STMTS NODE EFFECT-STMTS NIL)))
          (SETF (MV-PROG1-STMTS NODE)
                (CONS VALUES-STMT EFFECT-STMTS)))
   NODE)


(DEFUN META-SUBST-OPCODES (NODE)
   NODE)


(DEFUN META-SUBST-PROGN (NODE)
   (SETF (PROGN-STMTS NODE)
         (META-SUBST-STMTS NODE (PROGN-STMTS NODE)
                T))
```

```
   NODE)

(DEFUN META-SUBST-PROGV (NODE)                                    ;
   (MSUBST (PROGV-SYMS-EXPR NODE))
   (WHEN (PASSABLE (PROGV-SYMS-EXPR NODE)
                *SUBST-EXPR*)
       (MSUBST (PROGV-VALS-EXPR NODE))
       (WHEN (PASSABLE (PROGV-VALS-EXPR NODE)
                    *SUBST-EXPR*)
           (MSUBST (PROGV-STMT NODE))))
   (WHEN (OR (NULL (NODE-META-P (PROGV-SYMS-EXPR NODE)))
             (NULL (NODE-META-P (PROGV-VALS-EXPR NODE)))
             (NULL (NODE-META-P (PROGV-STMT NODE))))
       (SETF (NODE-META-P NODE)
             NIL))
   NODE)

(DEFUN META-SUBST-RETURN (NODE)
   (MSUBST (RETURN-VALUE NODE))
   (WHEN (NULL (NODE-META-P (RETURN-VALUE NODE)))
       (SETF (NODE-META-P NODE)
             NIL))
   NODE)

(DEFUN META-SUBST-SETQ (NODE)

;;; Someday, the SETQ removal code will make this method more substantial.

   (MSUBST (SETQ-VALUE NODE))
   (WHEN (NULL (NODE-META-P (SETQ-VALUE NODE)))
       (SETF (NODE-META-P NODE)
             NIL))
   NODE)

(DEFUN META-SUBST-TAGBODY (NODE)

;;; Because we don't do enough flow-analysis (or any, really), we can only safely substitute literals into loops.  Even variables are unsafe since they
;;; could be SETQ'd later in the loop.  We decide that we may be in a loop when we encounter a segment with a non-empty list of tags.

   (DOLIST (SEGMENT (TAGBODY-SEGMENTS NODE))
       (UNLESS (OR (LITERAL-P *SUBST-EXPR*)
                   (NULL (SEGMENT-TAGS SEGMENT)))
           (RETURN))
       (MULTIPLE-VALUE-BIND (STMTS PASSABLE?)
           (META-SUBST-STMTS NODE (SEGMENT-STMTS SEGMENT)
                  NIL)
         (SETF (SEGMENT-STMTS SEGMENT)
               STMTS)
         (UNLESS PASSABLE? (RETURN))))
   NODE)

(DEFUN META-SUBST-THROW (NODE)                                    ;
   (MSUBST (THROW-TAG NODE))
   (WHEN (PASSABLE *SUBST-EXPR* (THROW-TAG NODE))
       (MSUBST (THROW-VALUE NODE)))
   (WHEN (OR (NULL (NODE-META-P (THROW-TAG NODE)))
             (NULL (NODE-META-P (THROW-VALUE NODE))))
       (SETF (NODE-META-P NODE)
             NIL))
   NODE)

(DEFUN META-SUBST-UNWIND-PROTECT (NODE)

;;; This is fucked up because of the fact that the components of UNWIND-PROTECT's are stored as LAMBDA's prematurely.

   (MSUBST (UNWIND-PROTECT-STMT NODE))
   (WHEN (PASSABLE *SUBST-EXPR* (UNWIND-PROTECT-STMT NODE))
       (MSUBST (UNWIND-PROTECT-CLEANUP NODE)))
   (WHEN (OR (NULL (NODE-META-P (UNWIND-PROTECT-STMT NODE)))
             (NULL (NODE-META-P (UNWIND-PROTECT-CLEANUP NODE))))
       (SETF (NODE-META-P NODE)
             NIL))
   NODE)

(DEFUN META-SUBST-VAR-REF (NODE)
   (IF (AND (EQ *SUBST-VAR* (VAR-REF-VARIABLE NODE))
            (NOT (MEMBER (NODE-META-P NODE)
                         '(:MV :RETURN))))
       (LET ((NEW-CODE (COPY-CODE *SUBST-EXPR*)))
            (RELEASE-TREE NODE)
```

```
            (SETF (NODE-SUBST-P *SUBST-EXPR*)
                  T)
            (SETQ *SUBST-OCCURRED* T)
           NEW-CODE)
      NODE))
```

(DEFUN **META-SUBST-ANY-CALL** (NODE FN ARGS)

;;; Common code between CALL and MV-CALL

```
   (WHEN (PASSABLE *SUBST-EXPR* FN)
```

;; This can avoid being the case when the function is computed, for example by function call. Remember, CALL nodes are used for uses of ;; FUNCALL in the source, as well as in other cases.

```
      (DO ((TAIL ARGS (CDR TAIL)))
          ((NULL TAIL)
           (WHEN (LAMBDA-P FN)
```

;; Finally it's time to substitute inside the lambda.

```
                 (META-SUBST-LAMBDA FN T)))
```

;; Substitute into each argument and see if we can go further.

```
           (MSUBST (CAR TAIL))
           (WHEN (NOT (PASSABLE *SUBST-EXPR* (CAR TAIL)))          ; Can't go beyond this argument.
               (RETURN))))
```

;; Keep the META-P information up to date.

```
   (WHEN (OR (NULL (NODE-META-P FN))
             (NOTEVERY #'NODE-META-P ARGS))
         (SETF (NODE-META-P NODE)
               NIL))
   NODE)
```

(DEFUN **META-SUBST-STMTS** (NODE STMTS-LIST FINAL-STMT-IMMOVABLE)

;;; Common code between all of the too-many PROGN-like structures (PROGN, MV-PROG1, TAGBODY).

;;; The idea here is that we can reorder the nodes in the list as long as the side-effects analysis permits. We will partition the already-processed nodes ;;; into two sets, named after their counterparts in the S-1 compiler: WINNERS and BARRIERS. BARRIERS are nodes that are not PASSABLE with the ;;; *SUBST-EXPR*. For each node we process, we try to bring it past all of the BARRIERS. If we succeed, then we can substitute into it. Afterwards, if ;;; the node is PASSABLE with *SUBST-EXPR*, it goes on the WINNERS list; otherwise, it gets put onto BARRIERS at the end closer to the front of the ;;; stmts list.

;;; Before processing node A: || WINNERS || BARRIERS || A | OTHERS ||

;;; If can't pass all BARRIERS: || WINNERS || BARRIERS | A || OTHERS ||

;;; If passes all BARRIERS and is itself passable: || WINNERS | A || BARRIERS || OTHERS ||

;;; If passes all BARRIERS but not itself passable: || WINNERS || A | BARRIERS || OTHERS ||

;;; The lists WINNERS and BARRIERS are kept in reverse order, with the CAR being closer to the end of the stmts list.

;;; We return the new, possibly-permuted stmts list. Just for META-SUBST-TAGBODY, if there were no barriers (i.e., the whole stmts list is passable) we ;;; return a second value of T. The given NODE has its META-P field set to NIL if anything changes under here.

```
   (LET (NEW-STMTS-LIST)
        (DO ((TAIL STMTS-LIST (CDR TAIL))
             (WINNERS NIL)
             (BARRIERS NIL))
```

;; In some cases (notably PROGN), the final stmt is not movable.

```
            ((NULL (IF FINAL-STMT-IMMOVABLE
                       (CDR TAIL)
                       TAIL))
             (WHEN (AND TAIL (NULL BARRIERS))
                   (MSUBST (CAR TAIL))
                   (WHEN (NULL (NODE-META-P (CAR TAIL)))
                         (SETF (NODE-META-P NODE)
                               NIL)))
             (RETURN (VALUES (NCONC (REVERSE WINNERS)
                                    (REVERSE BARRIERS)
                                    TAIL)
                             (NULL BARRIERS))))
```

;; For each stmt, see if we can get it past all of the barriers.

```
            (COND
               ((EVERY #'(LAMBDA (BARRIER)
                                 (PASSABLE (CAR TAIL)
                                           BARRIER))
                       BARRIERS)
```

;; We got past the barriers.

```
                (LET (OUTER-S-O)
                     (LET ((*SUBST-OCCURRED* NIL))
```

```
                        (MSUBST (CAR TAIL))
                        (SETQ OUTER-S-O *SUBST-OCCURRED*)
                        ;; We made a change here if either we've permuted some stmts or a substitution was made inside the node.
                        (COND
                           ((PASSABLE (CAR TAIL)
                                   *SUBST-EXPR*)
                            (PUSH (CAR TAIL)
                                  WINNERS)
                            (WHEN (OR BARRIERS *SUBST-OCCURRED*)
                                (SETF (NODE-META-P NODE)
                                      NIL)))
                           (*SUBST-OCCURRED* (SETQ BARRIERS (NCONC BARRIERS (LIST (CAR TAIL))))
                                  (SETF (NODE-META-P NODE)
                                        NIL))
                           (T ;; Nothing happened and this one's not a winner, so don't make any gratuitous changes.
                              (PUSH (CAR TAIL)
                                    BARRIERS))))
                     ;; Pass on the fact that a substitution happened below here.
                     (WHEN OUTER-S-O (SETQ *SUBST-OCCURRED* OUTER-S-O))))
               (T ;; We couldn't get past the barriers.
                  (PUSH (CAR TAIL)
                        BARRIERS))))))

;; Testing meta-evaluation

(DEFUN TEST-META-EVAL (FN)
   (LET ((TREE (TEST-ALPHA-2 FN)))
        (UNWIND-PROTECT
            (PRINT-TREE (META-EVALUATE TREE))
            (RELEASE-TREE TREE))))

;; Arrange to use the correct compiler

(IL:PUTPROPS IL:XCLC-META-EVAL IL:FILETYPE COMPILE-FILE)

;; Arrange for the proper makefile-environment

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

(IL:PUTPROPS IL:XCLC-META-EVAL IL:COPYRIGHT ("Venue & Xerox Corporation" 1986 1987 1988 1989 1990 1991))
```

FUNCTION INDEX

VARIABLE INDEX

MACRO INDEX

PROPERTY INDEX