File created: 2-Oct-91 11:38:50 **{PELE:MV:ENVOS}<LISPCORE>SOURCES>XCLC-GENCODE.;4**

changes to: (IL:FUNCTIONS GENCODE-CALL)

previous date: 4-Jun-90 13:14:30 {PELE:MV:ENVOS}<LISPCORE>SOURCES>XCLC-GENCODE.;3

Read Table: XCL

Package: COMPILER

Format: XCCS


```
(IL:RPAQQ IL:XCLC-GENCODECOMS
          (

;;; Code Generation

            (IL:VARIABLES *AVAILABLE-LEXICAL-NAMES* *BLIP-VAR* *CODE* *CURRENT-FRAME* *FRAME-NAME* *FREE-VENV*
                  *NON-LOCALS* *OTHERS* *PC-VAR* *SPECIAL-LOCALS-BOUND* *SPECIAL-VENV* *STACK-NUMBER*
                  *SUPPRESS-POPS* *TAG-NUMBER* *TAIL-RECURSION-THRESHOLD* *VAR-NUMBER* *LOCAL-FUNCTIONS*)
            (IL:FUNCTIONS START-LAP EMIT-LAP EMIT-LAP-LIST END-LAP)
            (IL:FUNCTIONS COLLECT-CODE FIND-SEGMENT MAKE-LAP-VAR MAKE-LAP-VAR-REFERENCE)
            (IL:FUNCTIONS SET-UP-RETURN-TO TAKE-DOWN-RETURN-TO)
            (IL:FUNCTIONS FRAME INTERCEPT-NON-LOCALS)
            (IL:COMS (IL:STRUCTURES UNBIND-FOR-TAIL-RECURSION)
                  (IL:FUNCTIONS STOP-UNBINDS-AT-FRAME-BOUNDARY))
            (IL:FUNCTIONS GENERATE-CODE GENCODE)
                                                        ; Yet to be written: gencode-progv
            (IL:FUNCTIONS GENCODE-BLOCK GENCODE-CALL GENCODE-CATCH GENCODE-GO GENCODE-IF GENCODE-LABELS
                  GENCODE-LAMBDA GENCODE-LET GENCODE-LITERAL GENCODE-MV-CALL GENCODE-MV-PROG1 GENCODE-OPCODES
                  GENCODE-PROGN GENCODE-PROGV GENCODE-RETURN GENCODE-SEGMENT GENCODE-SETQ GENCODE-TAGBODY
                  GENCODE-TAGBODY-INLINE GENCODE-THROW GENCODE-UNWIND-PROTECT GENCODE-VAR-REF)
            ;; Policy variables.
            (IL:VARIABLES *POP-SUPPRESSION-POLICY* *TAIL-RECURSION-POLICY*)
            ;; Testing Code Generation
            (IL:FUNCTIONS TEST-GENCODE TEST-GENCODE1)
            ;; Arrange to use the correct compiler.
            (IL:PROP IL:FILETYPE IL:XCLC-GENCODE)
            ;; Arrange to use the proper makefile environment
            (IL:PROP IL:MAKEFILE-ENVIRONMENT IL:XCLC-GENCODE)))
```

;;; Code Generation

(DEFVAR ***AVAILABLE-LEXICAL-NAMES*** NIL

;;; A list of the previously-allocated-but-now-free names for lexical variables. Newly-bound variables will take names from this list unless it's empty, at ;;; which point new names will be generated. Each new frame rebinds this list to NIL.

)

(DEFVAR ***BLIP-VAR*** NIL
"If non-NIL, this is the LAP variable to be used for naming the blip variable in the current frame.")

(DEFVAR ***CODE*** NIL
"The current collection of LAP instructions, in reverse order. Added to during code generation.")

(DEFVAR ***CURRENT-FRAME*** NIL

;;; Set to the block, tagbody, catch, unwind-protect, or lambda that was the cause of the current frame during code generation. Used by GENCODE-GO ;;; and GENCODE-RETURN to determine if a jump is possible. Also used by GENCODE-CALL in order to get the right code generated for calls to ;;; IL:ARG. (Yecch.)

)

(DEFVAR ***FRAME-NAME*** NIL
"The name of the frame currently under construction. Used in the code generator.")

(DEFVAR ***FREE-VENV*** NIL
"An AList mapping the symbols naming freely referenced special variables into the LAP variables representing them. See also *special-venv*.")

(DEFVAR ***NON-LOCALS*** NIL
"A list of the lexical variables (in the form of VARIABLE structures) used freely below this point. Added to and reset at various points during code generation.")

```
(DEFVAR *OTHERS* NIL
   "A list of all auxillary variables used below the current point.  It eventually includes all non-parameter
   variables created within a given lambda.  Used during code generation.")

(DEFVAR *PC-VAR* NIL

;;; Bound to the LAP-var representing the special variable SI::*CATCH-RETURN-PC* in the current frame.  Used by blippers for unwinding.

   )

(DEFVAR *SPECIAL-LOCALS-BOUND* NIL
   "Bound to T in contexts in which local (i.e., non-argument) specials have been bound, in order to diable the
   tail-recursion optimization.")

(DEFVAR *SPECIAL-VENV* NIL
   "An AList mapping the symbols naming currently-bound special variables into the LAP variables representing
   them.  See also *free-venv*.")

(DEFVAR *STACK-NUMBER* NIL
   "Counter for generation of unique LAP stack-level names.")

(DEFVAR *SUPPRESS-POPS* NIL
   "If non-NIL code in effect context will suppress any extra pop's that might normally be generated.  This
   variable is rebound throughout the code generator.  To turn off this optimization, set the variable
   *pop-suppression-policy* to NIL.")

(DEFVAR *TAG-NUMBER* 0
   "Counter for the generation of unique LAP statement labels.")

(DEFPARAMETER *TAIL-RECURSION-THRESHOLD* 6
   "The maximum number of required arguments a function can have and still enable the tail-recursion
   optimization.")

(DEFVAR *VAR-NUMBER* 0
   "Counter for the generation of unique LAP variables.")

(DEFVAR *LOCAL-FUNCTIONS*)

(DEFMACRO START-LAP ()
   'NIL)

(DEFMACRO EMIT-LAP (INST)
   `(PUSH ,INST *CODE*))

(DEFMACRO EMIT-LAP-LIST (L)
   `(SETQ *CODE* (REVAPPEND ,L *CODE*)))

(DEFMACRO END-LAP ()
   `(NREVERSE *CODE*))

(DEFUN COLLECT-CODE (NODE CONTEXT)
   (LET ((*CODE* (START-LAP)))
        (GENCODE NODE CONTEXT)
        (END-LAP)))

(DEFUN FIND-SEGMENT (TAGBODY TAG)
   "Return the segment in the given tagbody that contains the given tag."
   (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS TAGBODY) IL:WHEN (MEMBER TAG (SEGMENT-TAGS SEGMENT)
                                                                     :TEST
                                                                     'EQ)
      IL:DO (RETURN SEGMENT)))

(DEFUN MAKE-LAP-VAR (VAR &OPTIONAL ARG-P)

;;; Create a new LAP variable for the given VARIABLE structure and make the appropriate kind of note about the variable created.

   (IF (NOT (VARIABLE-P VAR))
       VAR
       (ECASE (VARIABLE-SCOPE VAR)
```

```
          ((:SPECIAL) (LET ((LV `(:S ,(VARIABLE-NAME VAR)
                                    ,(INCF *VAR-NUMBER*))))
                        (PUSH (CONS (VARIABLE-NAME VAR)
                                    LV)
                              *SPECIAL-VENV*)
                        (WHEN (NOT ARG-P)
                              (PUSH LV *OTHERS*))
                        LV))
          ((:LEXICAL) (IF (OR (NULL *AVAILABLE-LEXICAL-NAMES*)
                              (VARIABLE-CLOSED-OVER VAR)
                              (EQ :FUNCTION (VARIABLE-KIND VAR)))
                          ;; Can't re-use a variable, so we'll make a new one.
                          (LET ((LV (LIST (IF (EQ :FUNCTION (VARIABLE-KIND VAR))
                                              :FN
                                              :L)
                                          (VARIABLE-NAME VAR)
                                          (INCF *VAR-NUMBER*))))
                               (SETF (VARIABLE-LAP-VAR VAR)
                                     LV)
                               (WHEN (AND (NOT ARG-P)
                                          (NOT (EQ (VARIABLE-KIND VAR)
                                                   :FUNCTION)))
                                     (PUSH LV *OTHERS*))
                               LV)
                          ;; There are old variables available for use.  Re-use one.
                          (SETF (VARIABLE-LAP-VAR VAR)
                                (LIST :L (VARIABLE-NAME VAR)
                                      (POP *AVAILABLE-LEXICAL-NAMES*)))))
          ((:GLOBAL) `(:G ,(VARIABLE-NAME VAR))))))

(DEFUN MAKE-LAP-VAR-REFERENCE (VAR)
   (ECASE (VARIABLE-SCOPE VAR)
       ((:LEXICAL)
          (PUSHNEW VAR *NON-LOCALS*)
          (LET ((LAP-VAR (VARIABLE-LAP-VAR VAR)))
               (ASSERT (NOT (NULL LAP-VAR))
                       NIL "BUG: ~S should should have a LAP var by now." VAR)
               LAP-VAR))
       ((:SPECIAL) (LET ((LOOKUP (OR (ASSOC (VARIABLE-NAME VAR)
                                            *SPECIAL-VENV*)
                                     (ASSOC (VARIABLE-NAME VAR)
                                            *FREE-VENV*))))
                     (IF (NOT (NULL LOOKUP))
                         (CDR LOOKUP)
                         (LET ((LV `(:F ,(VARIABLE-NAME VAR)
                                        ,(INCF *VAR-NUMBER*))))
                              (PUSH (CONS (VARIABLE-NAME VAR)
                                          LV)
                                    *FREE-VENV*)
                              (PUSH LV *OTHERS*)
                              LV))))
       ((:GLOBAL) `(:G ,(VARIABLE-NAME VAR)))))

(DEFUN SET-UP-RETURN-TO ()
   ;; Perform those operations necessary to set up a return-to context for code-generation.
   (WHEN (NULL *BLIP-VAR*)
       (SETQ *BLIP-VAR* `(:S SI::*CATCH-RETURN-TO* ,(INCF *VAR-NUMBER*)))
       (PUSH *BLIP-VAR* *OTHERS*))
   (WHEN (NULL *PC-VAR*)
       (SETQ *PC-VAR* `(:S SI::*CATCH-RETURN-PC* ,(INCF *VAR-NUMBER*)))
       (PUSH *PC-VAR* *OTHERS*)))

(DEFUN TAKE-DOWN-RETURN-TO ()
   (EMIT-LAP-LIST `((:CONST NIL)
                    (:VAR_ ,*BLIP-VAR*)
                    (:POP))))

(DEFMACRO FRAME ((&KEY CURRENT-FRAME NAME BLIPS-ALLOWED)
                 &BODY BODY)
   `(LET (,@(AND CURRENT-FRAME `((*CURRENT-FRAME* ,CURRENT-FRAME)))
          ,@(AND NAME `((*FRAME-NAME* ,NAME)))
          (*BLIP-VAR* NIL)
          (*PC-VAR* NIL)
          (*CODE* (START-LAP))
          *OTHERS* *SPECIAL-LOCALS-BOUND* *SPECIAL-VENV* *FREE-VENV* *AVAILABLE-LEXICAL-NAMES* *LOCAL-FUNCTIONS*
          )
         (HANDLER-BIND ((UNBIND-FOR-TAIL-RECURSION #'STOP-UNBINDS-AT-FRAME-BOUNDARY))
                ,@BODY)))
```

```
(DEFMACRO INTERCEPT-NON-LOCALS (PASS-ON &BODY BODY)
   `(LET (OUTER-NON-LOCALS)
         (LET (*NON-LOCALS*)
              ,@BODY
              (SETQ OUTER-NON-LOCALS ,PASS-ON))
         (SETQ *NON-LOCALS* (UNION OUTER-NON-LOCALS *NON-LOCALS*))))


(DEFINE-CONDITION UNBIND-FOR-TAIL-RECURSION (CONDITION)
   NIL)


(DEFUN STOP-UNBINDS-AT-FRAME-BOUNDARY (CONDITION)
   ;; This routine stops propagation of UNBIND-FOR-TAIL-RECURSION.
   (ASSERT (TYPEP CONDITION 'UNBIND-FOR-TAIL-RECURSION)
          NIL "BUG: Unbind stopper called with bad condition.")
   (LET ((RESTART (FIND-RESTART 'CONTINUE-TAIL-CALL-TRANSFORMATION)))
        (WHEN (NULL RESTART)
              (CERROR "Muddle on anyway" "BUG: Can't find restart for tail call transformation"))
        (INVOKE-RESTART RESTART)))


(DEFUN GENERATE-CODE (TREE)

;;; GenCode functions take a subtree as an argument and return a list of LAP instructions that implement that subtree.  Here, at the top of the generator,
;;; we know that TREE is a LAMBDA and that only one instruction will be returned.  We return that instruction.

   (ASSERT (LAMBDA-P TREE)
          NIL "Root tree for code generation is not a LAMBDA")
   (LET* ((*VAR-NUMBER* 0)
          (*TAG-NUMBER* 0)
          (*STACK-NUMBER* 0)
          (*CODE* (START-LAP)))
         (GENCODE-LAMBDA TREE :ARGUMENT)
         (ASSERT (NULL (CDR (SETQ *CODE* (END-LAP))))
                 NIL "Code generation returned more than one instruction!")
         (CAR *CODE*)))


(DEFUN GENCODE (NODE CONTEXT)
   "Dispatching function for code generation."
   (NODE-DISPATCH GENCODE NODE CONTEXT))

;; Yet to be written: gencode-progv


(DEFUN GENCODE-BLOCK (NODE CONTEXT)
   (COND
      ((BLOCK-NEW-FRAME-P NODE)                                    ; Construct a new lambda for the block.
       (LET (NEW-LAMBDA)
            (FRAME (:NAME (FORMAT NIL "block ~A in ~A" (BLOCK-NAME NODE)
                                  *FRAME-NAME*)
                          :CURRENT-FRAME NODE)
                   (LET ((EFFECTIVE-CONTEXT (ECASE CONTEXT
                                                   (:MV :MV)
                                                   (:RETURN :RETURN)
                                                   ((:EFFECT :ARGUMENT) :ARGUMENT)))
                         (END-TAG (INCF *TAG-NUMBER*))
                         BLIP-RETURN-VAR OUR-NON-LOCALS)
                        (SETF (BLOCK-FRAME NODE)
                              *CURRENT-FRAME*)
                        (SETF (BLOCK-CONTEXT NODE)
                              EFFECTIVE-CONTEXT)
                        (SETF (BLOCK-END-TAG NODE)
                              END-TAG)
                        (COND
                           ((BLOCK-CLOSED-OVER-P NODE)
                            (SETQ *BLIP-VAR* `(:S SI::*CATCH-RETURN-FROM* ,(INCF *VAR-NUMBER*)))
                            (SETQ BLIP-RETURN-VAR (MAKE-LAP-VAR (BLOCK-BLIP-VAR NODE)))
                            (EMIT-LAP-LIST `((:CONST ,*FRAME-NAME*)
                                             (:CONST NIL)
                                             (:CALL CONS 2)
                                             (:VAR_ ,*BLIP-VAR*)
                                             (:VAR_ ,BLIP-RETURN-VAR)
                                             (:POP)))
                            (SETQ *OTHERS* (LIST BLIP-RETURN-VAR *BLIP-VAR*)))
                           (T (SETQ *BLIP-VAR* `(:S SI::*CATCH-RETURN-TO* ,(INCF *VAR-NUMBER*)))
                              (SETQ *OTHERS* (LIST *BLIP-VAR*))))
                        (INTERCEPT-NON-LOCALS (SETQ OUR-NON-LOCALS (DELETE (BLOCK-BLIP-VAR NODE)
                                                                           *NON-LOCALS*))
                              (GENCODE (BLOCK-STMT NODE)
                                       EFFECTIVE-CONTEXT))
                        (EMIT-LAP-LIST `((:TAG ,END-TAG)
                                         (:RETURN)))
                        (SETQ NEW-LAMBDA
                              `(:LAMBDA (NIL ,@(AND *OTHERS* `(:OTHERS ,*OTHERS*))
```

```
                                        :NAME
                                        ,*FRAME-NAME* :BLIP ,*BLIP-VAR*
                                        ,@(AND OUR-NON-LOCALS `(:NON-LOCAL ,(MAPCAR #'VARIABLE-LAP-VAR
                                                                                   OUR-NON-LOCALS)))
                                        ,@(AND (BLOCK-CLOSED-OVER-VARS NODE)
                                               `(:CLOSED-OVER ,(MAPCAR #'VARIABLE-LAP-VAR (
                                                                                BLOCK-CLOSED-OVER-VARS
                                                                                     NODE))))
                                        ,@(AND *LOCAL-FUNCTIONS* `(:LOCAL-FUNCTIONS ,*LOCAL-FUNCTIONS*)))
                                    ,@(END-LAP)))))
      (EMIT-LAP `(:CALL ,NEW-LAMBDA 0))                    ; Generate a call to the new lambda.
      (WHEN (AND (EQ CONTEXT :EFFECT)
                 (NOT *SUPPRESS-POPS*))
          (EMIT-LAP '(:POP))))
  (T ;; No new frame is needed, so compile the block inline, setting up and taking down the blip stuff if it's closed over.
     (LET ((END-TAG (INCF *TAG-NUMBER*))
           (STK-NUM (INCF *STACK-NUMBER*)))
          (SETF (BLOCK-FRAME NODE)
                *CURRENT-FRAME*)
          (SETF (BLOCK-END-TAG NODE)
                END-TAG)
          (SETF (BLOCK-STK-NUM NODE)
                STK-NUM)
          (SETF (BLOCK-CONTEXT NODE)
                CONTEXT)
          ;; If the block is closed over, we need to set up and take down the blip stuff around the execution of the body.  If it isn't closed over,
          ;; then almost nothing extra is needed.
          (COND
             ((BLOCK-CLOSED-OVER-P NODE)
              (LET ((BLIP-VAR (MAKE-LAP-VAR (BLOCK-BLIP-VAR NODE)))
                    (REMOTE-RETURN-TAG (INCF *TAG-NUMBER*)))
                   (FLET ((GENCODE-CLOSED-OVER-BLOCK NIL (SET-UP-RETURN-TO)
                                (EMIT-LAP-LIST `((:CONST ,(BLOCK-NAME NODE))
                                                 (:CONST ,*FRAME-NAME*)
                                                 (:CALL CONS 2)
                                                 (:VAR_ ,*BLIP-VAR*)
                                                 (:VAR_ ,BLIP-VAR)
                                                 (:POP)
                                                 (:PUSH-TAG ,REMOTE-RETURN-TAG)
                                                 (:VAR_ ,*PC-VAR*)
                                                 (:POP)
                                                 (:NOTE-STACK ,STK-NUM)))
                                (INTERCEPT-NON-LOCALS (DELETE (BLOCK-BLIP-VAR NODE)
                                                              *NON-LOCALS*)
                                      (GENCODE (BLOCK-STMT NODE)
                                               CONTEXT))
                                (ECASE CONTEXT
                                    ((:EFFECT)
                                       (EMIT-LAP-LIST `((:JUMP ,END-TAG)
                                                        (:TAG ,REMOTE-RETURN-TAG)
                                                        (:DSET-STACK ,STK-NUM)
                                                        (:TAG ,END-TAG)))
                                       (TAKE-DOWN-RETURN-TO))
                                    ((:MV :ARGUMENT)
                                       (EMIT-LAP-LIST `((:JUMP ,END-TAG)
                                                        (:TAG ,REMOTE-RETURN-TAG)
                                                        (:SET-STACK ,STK-NUM)
                                                        (:TAG ,END-TAG)))
                                       (TAKE-DOWN-RETURN-TO))
                                    ((:RETURN) (EMIT-LAP-LIST `((:TAG ,REMOTE-RETURN-TAG)
                                                                (:TAG ,END-TAG)))))))
                          (IF (NULL (BLOCK-CLOSED-OVER-VARS NODE))
                              (GENCODE-CLOSED-OVER-BLOCK)
                              (LET ((CODE (LET ((*CODE* (START-LAP)))
                                               (GENCODE-CLOSED-OVER-BLOCK)
                                               (END-LAP))))
                                   (EMIT-LAP `(:CLOSE ,(MAPCAR #'VARIABLE-LAP-VAR (BLOCK-CLOSED-OVER-VARS
                                                                                      NODE))
                                                      ,@CODE)))))))
             (T ;; Simplest case: the block is neither closed over nor needs a new frame.
                (EMIT-LAP `(:NOTE-STACK ,STK-NUM))
                (GENCODE (BLOCK-STMT NODE)
                         CONTEXT)
                (EMIT-LAP `(:TAG ,END-TAG))))))))

(DEFUN GENCODE-CALL (NODE CONTEXT)

;;; If the function is a global function symbol, evaluate the arguments onto the stack and emit a FN.  If it's a lambda with only required parameters, then
;;; evaluate only the non-nil arguments on the stack, bind the parameters, execute the body, and unbind.  Otherwise, evaluate the args and function and
;;; call FUNCALL.

   (LET* ((FN (CALL-FN NODE))
          (ARGS (CALL-ARGS NODE))
```

```
          (NUM-ARGS (LENGTH ARGS))
          (IL-LAMBDA NIL))
         (COND
```

;; Can we perform tail recursion elimination?

```
          ((AND (EQ CONTEXT :RETURN)
                (NOT (NULL *TAIL-RECURSION-POLICY*))
                (NOT *SPECIAL-LOCALS-BOUND*)
                (NOT (CALLER-NOT-INLINE NODE))
                (VAR-REF-P FN)
                (LET ((VAR (VAR-REF-VARIABLE FN)))
                     (AND (EQ :FUNCTION (VARIABLE-KIND VAR))
                          (EQUAL *FRAME-NAME* (VARIABLE-NAME VAR)))
                                                          ; EQUAL here because of FLET.

                     )
                (<= (LENGTH (LAMBDA-REQUIRED *CURRENT-FRAME*))
                    *TAIL-RECURSION-THRESHOLD*)
                (OR (AND (NULL (LAMBDA-OPTIONAL *CURRENT-FRAME*))
                         (NULL (LAMBDA-REST *CURRENT-FRAME*)))
```

;; This for Interlisp-D LAMBDA form

```
                    (PROG1 (= (LENGTH (LAMBDA-OPTIONAL *CURRENT-FRAME*))
                              NUM-ARGS)
                           (SETQ IL-LAMBDA T)))
                (NULL (LAMBDA-KEYWORD *CURRENT-FRAME*)))
           (IL:FOR ARG IL:IN ARGS IL:DO (GENCODE ARG :ARGUMENT))
           (IF IL-LAMBDA
               (IL:FOR PARAM IL:IN (REVERSE (LAMBDA-OPTIONAL *CURRENT-FRAME*))
                  IL:DO (EMIT-LAP-LIST `((:VAR_ ,(MAKE-LAP-VAR-REFERENCE (CAR PARAM)))
                                         (:POP))))
               (IL:FOR PARAM IL:IN (REVERSE (LAMBDA-REQUIRED *CURRENT-FRAME*))
                  IL:DO (EMIT-LAP-LIST `((:VAR_ ,(MAKE-LAP-VAR-REFERENCE PARAM))
                                         (:POP)))))
           (RESTART-CASE (SIGNAL 'UNBIND-FOR-TAIL-RECURSION)
                 (CONTINUE-TAIL-CALL-TRANSFORMATION NIL))
           (EMIT-LAP `(:JUMP ,(OR (LAMBDA-TAIL-CALL-TAG *CURRENT-FRAME*)
                                  (SETF (LAMBDA-TAIL-CALL-TAG *CURRENT-FRAME*)
                                        (INCF *TAG-NUMBER*))))))
```

;; Maybe it's a global function

```
          ((GLOBAL-FUNCTION-P FN)
           (LET ((FN-VAR (VAR-REF-VARIABLE FN)))
                (COND
                   ((EQ (VARIABLE-NAME FN-VAR)
                        'IL:\\CALLME)
```

;; Hook for the IL:\\CALLME special form. This simply changes the name of the current frame to the given argument and
;; otherwise generates no code.

```
                    (ASSERT (EQ CONTEXT :EFFECT)
                            NIL "BUG: The ~S special form not in effect context in code generation."
                            'IL:\\CALLME)
                    (ASSERT (LITERAL-P (FIRST ARGS))
                            NIL "BUG: The ~S special form has an unquoted argument in code generation."
                            'IL:\\CALLME)
                    (SETQ *FRAME-NAME* (LITERAL-VALUE (FIRST ARGS)))
                    (RETURN-FROM GENCODE-CALL))
                   ((AND (MEMBER (VARIABLE-NAME FN-VAR)
                                 '(IL:\\ARG IL:\\SETARG))
                         (LITERAL-P (FIRST ARGS))
                         (NOT (CALLER-NOT-INLINE NODE)))
```

;; Here it is, the nasty hook for compiling Interlisp's LAMBDA no-spread's. If we're compiling a call to the function IL:ARG,
;; we check to see if it's referring to the current frame. If so, we compile it as a call to IL:\\ARG0 which will later be
;; assembled into an opcode. If the IL:ARG call doesn't refer to the current frame, then we compile it closed, using the
;; LAMBDA-version of that function, IL:\\ARG.

;; The same mechanism is here for IL:SETARG.

```
                    (LET* ((PARAMETER-NAME (LITERAL-VALUE (FIRST ARGS)))
                           (CLOSED-FN-NAME (VARIABLE-NAME FN-VAR))
                           (OPEN-FN-NAME (CASE CLOSED-FN-NAME
                                               (IL:\\ARG 'IL:\\ARG0)
                                               (IL:\\SETARG 'IL:\\SETARG0))))
                          (UNLESS (SYMBOLP PARAMETER-NAME)
                              (CERROR "Use the symbol %LOSE% instead" "Illegal argument to the ~S function:
                                      ~S" (VARIABLE-NAME FN-VAR)
                                      PARAMETER-NAME))
                          (COND
                             ((AND (LAMBDA-P *CURRENT-FRAME*)
                                   (EQ (LAMBDA-NO-SPREAD-NAME *CURRENT-FRAME*)
                                       PARAMETER-NAME))       ; It's a reference to the local frame.
                              (GENCODE (SECOND ARGS)
                                       :ARGUMENT)
                              (WHEN (THIRD ARGS)
                                  (GENCODE (THIRD ARGS)
                                           :ARGUMENT))
                              (EMIT-LAP `(:CALL ,OPEN-FN-NAME ,(1- (LENGTH ARGS)))))
                             (T                               ; It's a remote reference.
```

```
                              (EMIT-LAP `(:CONST ,PARAMETER-NAME))
                              (GENCODE (SECOND ARGS)
                                     :ARGUMENT)
                              (WHEN (THIRD ARGS)
                                  (GENCODE (THIRD ARGS)
                                         :ARGUMENT))
                              (EMIT-LAP `(:CALL ,CLOSED-FN-NAME ,(LENGTH ARGS)))))))
                 (T (IL:FOR ARG IL:IN ARGS IL:DO (GENCODE ARG :ARGUMENT))
                    (EMIT-LAP `(:CALL ,(VARIABLE-NAME FN-VAR)
                                      ,NUM-ARGS
                                      ,@(AND (CALLER-NOT-INLINE NODE)
                                             `(:NOT-INLINE T))))))
           (CASE CONTEXT
               (:EFFECT (WHEN (NOT *SUPPRESS-POPS*)
                              (EMIT-LAP '(:POP))))
               (:MV (EMIT-LAP '(:CALL IL:\\MVLIST 1))))))
```

;; Is it a desugared LET?

```
((AND (LAMBDA-P FN)
      (NOT (LAMBDA-NEW-FRAME-P FN)))
```

;; NOTE: This code makes the assumption that *code* is maintained by pushing bytes onto a list and should be re-examined if that is ;; ever changed (e.g., to the TCONC method).

;; NOTE: This is a LET* so as to guarantee that the code below has been generated before we try to extract LAP vars from the ;; variables, since those LAP vars might not exist yet.

```
 (LET* ((INNER-CODE (LET (*CODE*)
                         (GENCODE-LET FN ARGS CONTEXT)
                         *CODE*))
        (CLOSED-OVER (MAPCAR #'VARIABLE-LAP-VAR (LAMBDA-CLOSED-OVER-VARS FN))))
       (COND
          ((NULL CLOSED-OVER)
           (SETQ *CODE* (NCONC INNER-CODE *CODE*)))
          (T (EMIT-LAP `(:CLOSE ,CLOSED-OVER ,@(NREVERSE INNER-CODE)))))))
```

;; Perhaps it's a low-level OPCODES function.

```
((OPCODES-P FN)
 (IL:FOR ARG IL:IN ARGS IL:DO (GENCODE ARG :ARGUMENT))
 (EMIT-LAP `(:CALL (:OPCODES ,@(OPCODES-BYTES FN))
                   ,NUM-ARGS))
 (CASE CONTEXT
     (:EFFECT (WHEN (NOT *SUPPRESS-POPS*)
                    (EMIT-LAP '(:POP))))
     (:MV (EMIT-LAP '(:CALL IL:\\MVLIST 1)))))
```

;; Well, it wasn't any of those, so compile the general case.

```
(T (COND
       ((IL:FOR ARG IL:IN ARGS IL:ALWAYS (PASSABLE FN ARG))
                                                  ; The function can pass all of the arguments, so we can use
                                                  ; APPLYFN without an auxillary variable.
        (IL:FOR ARG IL:IN ARGS IL:DO (GENCODE ARG :ARGUMENT))
        (COND
           ((LAMBDA-P FN)
            (LET ((CODE-FOR-FN (COLLECT-CODE FN :ARGUMENT)))
                 (EMIT-LAP `(:CALL ,(POP CODE-FOR-FN)
                                   ,NUM-ARGS))
                 (ASSERT (NULL CODE-FOR-FN)
                        NIL "BUG: a lambda generated more than one LAP op.")))
           ((AND (VAR-REF-P FN)
                 (NOT (EQ (VARIABLE-SCOPE (VAR-REF-VARIABLE FN))
                          :GLOBAL)))              ; Must be a local or a special - external functions have already
                                                  ; been handled.
            (LET ((VAR (VAR-REF-VARIABLE FN)))
                 (ASSERT (NOT (EQ (VARIABLE-SCOPE VAR)
                                  :GLOBAL))
                        '(FN)
                        "BUG: external function call got into the general case.")
                 (EMIT-LAP `(:CALL ,(MAKE-LAP-VAR-REFERENCE VAR)
                                   ,NUM-ARGS))))
           (T                                     ; Random expression - have to punt to a :STKCALL
              (EMIT-LAP `(:CONST ,NUM-ARGS))
              (GENCODE FN :ARGUMENT)
              (EMIT-LAP `(:STKCALL ,NUM-ARGS))))
        (COND
           ((LAMBDA-P FN)
            (GENCODE-LAMBDA FN :ARGUMENT CONTEXT))))
       (T                                         ; Rats!  We have to allocate a new local variable to store the
                                                  ; function in during the evaluation of the arguments.
          (LET ((FN-VAR (COND
                           ((NULL *AVAILABLE-LEXICAL-NAMES*)
                            (LET ((LV `(:L "APPLYFN Variable" ,(INCF *VAR-NUMBER*))))
                                 (PUSH LV *OTHERS*)
                                 LV))
                           (T `(:L "APPLYFN Variable" ,(POP *AVAILABLE-LEXICAL-NAMES*))))))
              (IF (LAMBDA-P FN)
                  (GENCODE-LAMBDA FN :ARGUMENT CONTEXT)
                  (GENCODE FN :ARGUMENT))
              (EMIT-LAP-LIST `((:VAR_ ,FN-VAR)
```

```
                                    (:POP)))
                          (IL:FOR ARG IL:IN ARGS IL:DO (GENCODE ARG :ARGUMENT))
                          (EMIT-LAP `(:CALL ,FN-VAR ,NUM-ARGS))
                          (PUSH (THIRD FN-VAR)
                                *AVAILABLE-LEXICAL-NAMES*))))
                (CASE CONTEXT
                    (:EFFECT (WHEN (NOT *SUPPRESS-POPS*)
                                   (EMIT-LAP '(:POP))))
                    (:MV (UNLESS (LAMBDA-P FN)                    ; If the function is a LAMBDA, then we've already made it return
                                                                  ; a list of values.
                                 (EMIT-LAP '(:CALL IL:\\MVLIST 1)))))))))
```

```
(DEFUN GENCODE-CATCH (NODE CONTEXT)
   (COND
      ((BLIPPER-NEW-FRAME-P NODE)
       ;; This CATCH has to be a new frame.  Compile the body as a new function with one argument, the value of the tag expression.
       (LET (NEW-LAMBDA)
            (FRAME (:CURRENT-FRAME NODE :NAME (FORMAT NIL "catch in ~A" *FRAME-NAME*))
                   (LET* ((BLIP-SLOT-NAME (IF (EQ CONTEXT :MV)
                                              'SI::*CATCH-RETURN-TO*
                                              'SI::*CATCH-RETURN-FROM*))
                          (BLIP-SLOT-VAR `(:S ,BLIP-SLOT-NAME ,(INCF *VAR-NUMBER*)))
                          (TAG-VAR `(:L "%TAG" ,(INCF *VAR-NUMBER*)))
                          OUR-NON-LOCALS
                          (THROW-PC-VAR `(:S SI::*CATCH-RETURN-PC* ,(INCF *VAR-NUMBER*)))
                          (THROW-DESTINATION-TAG (INCF *TAG-NUMBER*)))
                         (INTERCEPT-NON-LOCALS (SETQ OUR-NON-LOCALS *NON-LOCALS*)
                                (GENCODE (CATCH-STMT NODE)
                                         CONTEXT))
                         (SETQ NEW-LAMBDA `(:LAMBDA ((,TAG-VAR)
                                                     :BLIP
                                                     ,BLIP-SLOT-VAR :NAME ,*FRAME-NAME* :OTHERS
                                                     (,BLIP-SLOT-VAR ,@(AND (EQ CONTEXT :MV)
                                                                            `(,THROW-PC-VAR))
                                                            ,@*OTHERS*)
                                                     ,@(AND OUR-NON-LOCALS
                                                            `(:NON-LOCAL ,(MAPCAR #'VARIABLE-LAP-VAR
                                                                                  OUR-NON-LOCALS)))
                                                     ,@(AND (CATCH-CLOSED-OVER-VARS NODE)
                                                            `(:CLOSED-OVER ,(MAPCAR #'VARIABLE-LAP-VAR
                                                                                    (CATCH-CLOSED-OVER-VARS NODE)
                                                                                    )))
                                                     ,@(AND *LOCAL-FUNCTIONS* `(:LOCAL-FUNCTIONS
                                                                                ,*LOCAL-FUNCTIONS*)))
                                                ;; Set up the blip and, when in MV context, the THROW PC.
                                                (:VAR ,TAG-VAR)
                                                (:VAR_ ,BLIP-SLOT-VAR)
                                                (:POP)
                                                ,@(AND (EQ CONTEXT :MV)
                                                       `((:PUSH-TAG ,THROW-DESTINATION-TAG)
                                                         (:VAR_ ,THROW-PC-VAR)
                                                         (:POP)))
                                                ,@(END-LAP)
                                                (:RETURN)
                                                ,@(AND (EQ CONTEXT :MV)
                                                       `((:TAG ,THROW-DESTINATION-TAG)
                                                         (:CALL IL:\\MVLIST 1)
                                                         (:RETURN)))))))
                   (GENCODE (CATCH-TAG NODE)
                            :ARGUMENT)
                   (EMIT-LAP `(:CALL ,NEW-LAMBDA 1))
                   (WHEN (EQ CONTEXT :EFFECT)
                         (EMIT-LAP '(:POP)))))
      (T ;; This CATCH should not be a new frame.  We compile it inline, setting up and taking down blip stuff around the computation of the body.
         (LET ((END-TAG (INCF *TAG-NUMBER*))
               (THROW-TAG (INCF *TAG-NUMBER*))
               (STK-NUM (INCF *STACK-NUMBER*)))
              (SET-UP-RETURN-TO)
              (GENCODE (CATCH-TAG NODE)
                       :ARGUMENT)
              (EMIT-LAP-LIST `((:VAR_ ,*BLIP-VAR*)
                               (:POP)
                               (:PUSH-TAG ,THROW-TAG)
                               (:VAR_ ,*PC-VAR*)
                               (:POP)
                               (:NOTE-STACK ,STK-NUM)))
              (GENCODE (CATCH-STMT NODE)
                       CONTEXT)
              (ECASE CONTEXT
                  ((:EFFECT)
                     (EMIT-LAP-LIST `((:JUMP ,END-TAG)
                                      (:TAG ,THROW-TAG)
                                      (:DSET-STACK ,STK-NUM)
```

```
                                    (:TAG ,END-TAG)))
                        (TAKE-DOWN-RETURN-TO))
                    ((:MV)
                        (EMIT-LAP-LIST `((:JUMP ,END-TAG)
                                        (:TAG ,THROW-TAG)
                                        (:CALL IL:\\MVLIST 1)
                                        (:SET-STACK ,STK-NUM)
                                        (:TAG ,END-TAG)))
                        (TAKE-DOWN-RETURN-TO))
                    ((:ARGUMENT)
                        (EMIT-LAP-LIST `((:JUMP ,END-TAG)
                                        (:TAG ,THROW-TAG)
                                        (:SET-STACK ,STK-NUM)
                                        (:TAG ,END-TAG)))
                        (TAKE-DOWN-RETURN-TO))
                    ((:RETURN) (EMIT-LAP `(:TAG ,THROW-TAG)))))))

(DEFUN GENCODE-GO (NODE CONTEXT)
   (DECLARE (IGNORE CONTEXT))
   (LET* ((TAGBODY (GO-TAGBODY NODE))
          (SEGMENT (FIND-SEGMENT TAGBODY (GO-TAG NODE))))
         (COND
            ((EQ *CURRENT-FRAME* (TAGBODY-FRAME TAGBODY))        ; The tagbody is local; a simple stack adjustment and JUMP
                                                                 ; suffice.
             (EMIT-LAP-LIST `((:DSET-STACK ,(TAGBODY-STK-NUM TAGBODY))
                              (:JUMP ,(SEGMENT-LOCAL-TAG SEGMENT)))))
            (T (EMIT-LAP-LIST `((:VAR ,(VARIABLE-LAP-VAR (TAGBODY-BLIP-VAR TAGBODY)))
                                (:PUSH-TAG ,(SEGMENT-REMOTE-TAG SEGMENT))
                                (:CALL SI::NON-LOCAL-GO 2)))
               (PUSH (TAGBODY-BLIP-VAR TAGBODY)
                     *NON-LOCALS*)))))

(DEFUN GENCODE-IF (NODE CONTEXT)
   (LET ((ELSE-TAG (INCF *TAG-NUMBER*))
         (AFTER-IF-TAG (INCF *TAG-NUMBER*)))
        (GENCODE (IF-PRED NODE)
                 :ARGUMENT)
        (EMIT-LAP `(:FJUMP ,ELSE-TAG))
        (GENCODE (IF-THEN NODE)
                 CONTEXT)
        (IF (EQ CONTEXT :RETURN)
            (EMIT-LAP '(:RETURN))
            (EMIT-LAP `(:JUMP ,AFTER-IF-TAG)))
        (EMIT-LAP `(:TAG ,ELSE-TAG))
        (GENCODE (IF-ELSE NODE)
                 CONTEXT)
        (UNLESS (EQ CONTEXT :RETURN)
            (EMIT-LAP `(:TAG ,AFTER-IF-TAG)))))

(DEFUN GENCODE-LABELS (NODE CONTEXT)
   ;; Make LAP vars first to take care of "forward" references.
   (DOLIST (FN-PAIR (LABELS-FUNS NODE))
       (MAKE-LAP-VAR (CAR FN-PAIR)))
   ;; Generate the local functions.
   (LET (INNER-CODE CLOSED-OVER)
        (INTERCEPT-NON-LOCALS (DELETE-IF #'(LAMBDA (VAR)
                                             (EQ (VARIABLE-BINDER VAR)
                                                 NODE))
                                         *NON-LOCALS*)
             (DOLIST (FN-PAIR (LABELS-FUNS NODE))
                 (PUSH (CONS (MAKE-LAP-VAR-REFERENCE (CAR FN-PAIR))
                             (COLLECT-CODE (CDR FN-PAIR)
                                           :ARGUMENT))
                       *LOCAL-FUNCTIONS*))
             (SETQ INNER-CODE (COLLECT-CODE (LABELS-BODY NODE)
                                            CONTEXT))
             (SETQ CLOSED-OVER (MAPCAR #'VARIABLE-LAP-VAR (LABELS-CLOSED-OVER-VARS NODE))))
        (COND
           ((NULL CLOSED-OVER)
            (SETQ *CODE* (REVAPPEND INNER-CODE *CODE*)))
           (T (EMIT-LAP `(:CLOSE ,CLOSED-OVER ,@INNER-CODE))))))

(DEFUN GENCODE-LAMBDA (NODE CONTEXT)
   (WHEN (EQ CONTEXT :EFFECT)                                    ; Lambda expressions cannot have any side-effects.
       (RETURN-FROM GENCODE-LAMBDA))
   (LET (NEW-LAMBDA OUR-NON-LOCALS CLOSED-OVER REQUIRED OPTIONAL REST KEY)
        (FRAME (:NAME (OR (LAMBDA-NAME NODE)
                          (FORMAT NIL "lambda in ~A" *FRAME-NAME*))
                :CURRENT-FRAME NODE)
             (INTERCEPT-NON-LOCALS (SETQ OUR-NON-LOCALS (DELETE-IF #'(LAMBDA (VAR)
                                                                       (EQ (VARIABLE-BINDER VAR)
```

```
                                                                    NODE))
                                                          *NON-LOCALS*))

        ;; Convert the parameters into LAP-code notation.
        (SETQ REQUIRED (MAPCAR #'(LAMBDA (VAR)
                                         (MAKE-LAP-VAR VAR T))
                               (LAMBDA-REQUIRED NODE)))
        (SETQ OPTIONAL (MAPCAR #'(LAMBDA (OPT-VAR)
                                         (LET ((INIT-CODE (COLLECT-CODE (SECOND OPT-VAR)
                                                                        :ARGUMENT)))
                                           ;; Generating code for  the init form has to come before we create the varibles so that free
                                           ;; references in the init form don't capture them.
                                                (LIST (MAKE-LAP-VAR (FIRST OPT-VAR)
                                                                    T)
                                                      INIT-CODE
                                                      (MAKE-LAP-VAR (THIRD OPT-VAR)
                                                                    T))))
                               (LAMBDA-OPTIONAL NODE)))
        (SETQ REST (LET ((REST-VAR (LAMBDA-REST NODE)))
                     (COND
                        ((NULL REST-VAR)
                         NIL)
                        ((AND (EQ :LEXICAL (VARIABLE-SCOPE REST-VAR))
                              (NULL (VARIABLE-READ-REFS REST-VAR))
                              (NULL (VARIABLE-WRITE-REFS REST-VAR)))
                         :IGNORED)
                        (T (MAKE-LAP-VAR REST-VAR T)))))
        (SETQ KEY (MAPCAR #'(LAMBDA (KEY-VAR)
                                    (LET ((INIT-CODE (COLLECT-CODE (THIRD KEY-VAR)
                                                                   :ARGUMENT)))
                                      ;; Generating code for  the init form has to come before we create the varibles so that free
                                      ;; references in the init form don't capture them.
                                           (LIST (FIRST KEY-VAR)
                                                 (MAKE-LAP-VAR (SECOND KEY-VAR)
                                                               T)
                                                 INIT-CODE
                                                 (MAKE-LAP-VAR (FOURTH KEY-VAR)
                                                               T))))
                          (LAMBDA-KEYWORD NODE)))
        ;; Generate code for the body of the lambda.
        (GENCODE (LAMBDA-BODY NODE)
                 :RETURN))
    ;; Convert the closed-over variable list into LAP vars.
    (SETQ CLOSED-OVER (MAPCAR #'VARIABLE-LAP-VAR (LAMBDA-CLOSED-OVER-VARS NODE)))
    ;; Finally, construct the lambda-structure for the LAP-code.
    (SETQ NEW-LAMBDA `(:LAMBDA (,REQUIRED ,@(AND OPTIONAL `(:OPTIONAL ,OPTIONAL))
                                          ,@(AND REST `(:REST ,REST))
                                          ,@(AND KEY `(:KEY ,KEY))
                                          ,@(AND (LAMBDA-ALLOW-OTHER-KEYS NODE)
                                                 '(:ALLOW-OTHER-KEYS T))
                                          ,@(AND *OTHERS* `(:OTHERS ,*OTHERS*))
                                          ,@(AND *BLIP-VAR* `(:BLIP ,*BLIP-VAR*))
                                          :NAME
                                          ,*FRAME-NAME*
                                          ,@(AND (LAMBDA-ARG-TYPE NODE)
                                                 `(:ARG-TYPE ,(LAMBDA-ARG-TYPE NODE)))
                                          ,@(AND CLOSED-OVER `(:CLOSED-OVER ,CLOSED-OVER))
                                          ,@(AND OUR-NON-LOCALS `(:NON-LOCAL ,(MAPCAR #'VARIABLE-LAP-VAR
                                                                                      OUR-NON-LOCALS)))
                                          ,@(AND *LOCAL-FUNCTIONS* `(:LOCAL-FUNCTIONS ,*LOCAL-FUNCTIONS*)
                                                 ))
                                    ,@(AND (LAMBDA-TAIL-CALL-TAG NODE)
                                           `((:TAG ,(LAMBDA-TAIL-CALL-TAG NODE))))
                                    ,@(END-LAP)
                                    (:RETURN))))
;; Now that we're outside of the bindings of the specials above, we can pass on our results to the outside world.
(EMIT-LAP NEW-LAMBDA)
(WHEN (EQ CONTEXT :MV)                                  ; In MV context, we need to make this a list.
    (EMIT-LAP-LIST '((:CONST NIL)
                     (:CALL CONS 2))))))

(DEFUN GENCODE-LET (FN ARGS CONTEXT)

;;; Compile the given function and arguments as a LET.

    ;; Separate the arguments and matching parameters into two sets, according to whether or not the argument is NIL.  Also, since we're not allowed to
    ;; BIND any closed-over variables, we have to store those arguments as we compute them to get them out of the way.
    (LET ((*SPECIAL-VENV* *SPECIAL-VENV*)
          (SPECIAL-LOCALS-HERE NIL)
          (STK-NUM (INCF *STACK-NUMBER*))
```

```
         NULL-PARAMS NON-NULL-PARAMS)
        (IL:FOR ARG IL:IN ARGS IL:AS PARAM IL:IN (LAMBDA-REQUIRED FN)
           IL:DO (WHEN (EQ :SPECIAL (VARIABLE-SCOPE PARAM))
                       (SETQ SPECIAL-LOCALS-HERE T))
                 (COND
                    ;; This parameter is set to NIL.
                    ((AND (LITERAL-P ARG)
                          (EQ NIL (LITERAL-VALUE ARG)))
                     (IF (EQ :LEXICAL (VARIABLE-SCOPE PARAM))
                         (EMIT-LAP-LIST `((:CONST NIL)
                                          (:VAR_ ,(MAKE-LAP-VAR PARAM))
                                          (:POP)))
                         (PUSH PARAM NULL-PARAMS)))
                    ;; This parameter is bound to the result of a non-null expression
                    (T (GENCODE ARG :ARGUMENT)
                       (IF (EQ :LEXICAL (VARIABLE-SCOPE PARAM))
                           (EMIT-LAP-LIST `((:VAR_ ,(MAKE-LAP-VAR PARAM))
                                            (:POP)))
                           (PUSH PARAM NON-NULL-PARAMS)))))
        (SETQ NULL-PARAMS (MAPCAR #'MAKE-LAP-VAR NULL-PARAMS))
        (SETQ NON-NULL-PARAMS (NREVERSE (MAPCAR #'MAKE-LAP-VAR NON-NULL-PARAMS)))
        ;; Bind the variables and evaluate the body
        (WHEN (OR NULL-PARAMS NON-NULL-PARAMS)
            (EMIT-LAP `(:BIND ,NON-NULL-PARAMS ,NULL-PARAMS ,STK-NUM)))
        (INTERCEPT-NON-LOCALS (DELETE-IF #'(LAMBDA (VAR)
                                             (EQ (VARIABLE-BINDER VAR)
                                                 FN))
                                         *NON-LOCALS*)
              ;; !!HACK!! Fix this silliness once the compiler can use lexical closures.
              (LET* ((*SPECIAL-LOCALS-BOUND* (OR *SPECIAL-LOCALS-BOUND* SPECIAL-LOCALS-HERE)))
                    ;; If we've done a BIND, then we need to make it possible for the tail-recursion optimization to generate a DUNBIND.
                    ;; Otherwise, things are simpler.
                    (IF (OR NULL-PARAMS NON-NULL-PARAMS)
                        (LET* ((UNBIND-INST `(:DUNBIND ,NON-NULL-PARAMS ,NULL-PARAMS ,STK-NUM))
                               (UNBIND-FN `(LAMBDA (CONDITION)
                                                   (DECLARE (IGNORE CONDITION))
                                                   (EMIT-LAP ',UNBIND-INST))))
                              (CONDITION-BIND ((UNBIND-FOR-TAIL-RECURSION UNBIND-FN))
                                    (GENCODE (LAMBDA-BODY FN)
                                             CONTEXT)))
                        (GENCODE (LAMBDA-BODY FN)
                                 CONTEXT))))
        ;; Again, we only need to UNBIND if we generated a BIND earlier.
        (WHEN (OR NULL-PARAMS NON-NULL-PARAMS)
            (ECASE CONTEXT
                ((:EFFECT) (WHEN (NOT *SUPPRESS-POPS*)
                                 (EMIT-LAP `(:DUNBIND ,NON-NULL-PARAMS ,NULL-PARAMS ,STK-NUM))))
                ((:ARGUMENT :MV) (EMIT-LAP `(:UNBIND ,NON-NULL-PARAMS ,NULL-PARAMS ,STK-NUM)))
                ((:RETURN) )))
        ;; Finally, we know that these variables aren't in use any more, so we can put their names on *AVAILABLE-LEXICAL-NAMES*.
        (IL:FOR PARAM IL:IN (LAMBDA-REQUIRED FN) IL:WHEN (AND (EQ :LEXICAL (VARIABLE-SCOPE PARAM))
                                                              (NOT (VARIABLE-CLOSED-OVER PARAM)))
           IL:DO (PUSH (THIRD (VARIABLE-LAP-VAR PARAM))
                       *AVAILABLE-LEXICAL-NAMES*))))


(DEFUN GENCODE-LITERAL (NODE CONTEXT)
   (ECASE CONTEXT
       (:EFFECT                                                    ; Do nothing.
             )
       ((:ARGUMENT :RETURN) (EMIT-LAP `(:CONST ,(LITERAL-VALUE NODE))))
       (:MV                                                        ; In MV context, we need to make this a list.  Because MV-CALL
                                                                   ; uses NCONC to put the lists together, we have to CONS a
                                                                   ; fresh cell here.
          (EMIT-LAP-LIST `((:CONST ,(LITERAL-VALUE NODE))
                           (:CONST NIL)
                           (:CALL CONS 2))))))


(DEFUN GENCODE-MV-CALL (NODE CONTEXT)
   (LET ((FN (MV-CALL-FN NODE))
         (ARGS (MV-CALL-ARG-EXPRS NODE)))
        (FLET ((GENERATE-VALUES NIL
                    ;; Generate the code for putting the list of values on the top-of-stack.
                    (GENCODE (FIRST ARGS)
                             :MV)
                    (IL:|for| ARG IL:|in| (REST ARGS) IL:|do| (GENCODE ARG :MV)
                                                              (EMIT-LAP '(:CALL IL:\\NCONC2 2)))))
              (COND
```

```
                ((AND (GLOBAL-FUNCTION-P FN)
                      (EQ (VARIABLE-NAME (VAR-REF-VARIABLE FN))
                          'LIST))                                   ; This is a use of MULTIPLE-VALUE-LIST, the only use of
                                                                    ; multiple-values that XCL can do reasonably well.
                 ;; we can do better here - if we're in effect context, there's no reason to do all this consing and we should just treat this as a
                 ;; PROGN.
                 (GENERATE-VALUES))
                (T (IF (GLOBAL-FUNCTION-P FN)
                       (EMIT-LAP `(:CONST ,(VARIABLE-NAME (VAR-REF-VARIABLE FN))))
                       (GENCODE FN :ARGUMENT))
                   (GENERATE-VALUES)
                   (EMIT-LAP '(:CALL APPLY 2))))
            (CASE CONTEXT
                (:EFFECT (WHEN (NOT *SUPPRESS-POPS*)
                               (EMIT-LAP '(:POP))))
                (:MV (EMIT-LAP '(:CALL IL:\\MVLIST 1)))))))


(DEFUN GENCODE-MV-PROG1 (NODE CONTEXT)
   (ECASE CONTEXT
       ((:MV :RETURN)                                               ; All other contexts should have been meta-eval'ed away.
          (DESTRUCTURING-BIND (VALUES-FORM . EFFECT-FORMS)
                 (MV-PROG1-STMTS NODE)
                 ;; Save the values from the first statement on the stack while we evaluate the rest of the values.
                 (GENCODE VALUES-FORM :MV)
                 (IL:FOR FORM IL:IN EFFECT-FORMS IL:DO (GENCODE FORM :EFFECT))
                 (WHEN (EQ CONTEXT :RETURN)
                     (EMIT-LAP '(:CALL VALUES-LIST 1)))))))


(DEFUN GENCODE-OPCODES (NODE CONTEXT)
   (DECLARE (IGNORE NODE CONTEXT))
   (ASSERT NIL NIL "BUG: GENCODE-OPCODES was called!"))


(DEFUN GENCODE-PROGN (NODE CONTEXT)
   (LET ((*SUPPRESS-POPS* (AND *POP-SUPPRESSION-POLICY* (EQ CONTEXT :RETURN))))
        (IL:FOR TAIL IL:ON (PROGN-STMTS NODE) IL:DO (GENCODE (CAR TAIL)
                                                             (IF (NULL (CDR TAIL))
                                                                 CONTEXT
                                                                 :EFFECT)))))


(DEFUN GENCODE-PROGV (&REST IGNORE)
   (ASSERT NIL NIL "BUG: GENCODE-PROGV was called."))


(DEFUN GENCODE-RETURN (NODE CONTEXT)
   (LET ((BLOCK (RETURN-BLOCK NODE)))
        (COND
           ((EQ *CURRENT-FRAME* (BLOCK-FRAME BLOCK))
            ;; The block is local; a simple stack adjustment and jump will suffice.
            (GENCODE (RETURN-VALUE NODE)
                     (BLOCK-CONTEXT BLOCK))
            (COND
               ((BLOCK-NEW-FRAME-P BLOCK)
                ;; This RETURN is returning from the frame itself, rather than from a block internal to a frame.  Don't need the stack adjustment.
                ;; JDS 1/26/89 I think this is correct.
                (ECASE (BLOCK-CONTEXT BLOCK)
                    ((:EFFECT) (EMIT-LAP-LIST `((:JUMP ,(BLOCK-END-TAG BLOCK)))))
                    ((:ARGUMENT :MV) (EMIT-LAP-LIST `((:JUMP ,(BLOCK-END-TAG BLOCK)))))
                    ((:RETURN) (EMIT-LAP '(:RETURN)))))
               (T (ECASE (BLOCK-CONTEXT BLOCK)
                      ((:EFFECT) (EMIT-LAP-LIST `((:DSET-STACK ,(BLOCK-STK-NUM BLOCK))
                                                  (:JUMP ,(BLOCK-END-TAG BLOCK)))))
                      ((:ARGUMENT :MV) (EMIT-LAP-LIST `((:SET-STACK ,(BLOCK-STK-NUM BLOCK))
                                                        (:JUMP ,(BLOCK-END-TAG BLOCK)))))
                      ((:RETURN) (EMIT-LAP '(:RETURN))))))))
           (T ;; The block is remote; call on the unwinder.
              (EMIT-LAP `(:VAR ,(VARIABLE-LAP-VAR (BLOCK-BLIP-VAR BLOCK))))
              (ECASE (BLOCK-CONTEXT BLOCK)
                  (:EFFECT
                     (GENCODE (RETURN-VALUE NODE)
                              :EFFECT)
                     (EMIT-LAP '(:CALL SI::NON-LOCAL-RETURN 1)))
                  ((:MV :ARGUMENT)
                     (GENCODE (RETURN-VALUE NODE)
                              (BLOCK-CONTEXT BLOCK))
                     (EMIT-LAP '(:CALL SI::NON-LOCAL-RETURN 2)))
                  (:RETURN
                     (GENCODE (RETURN-VALUE NODE)
                              :MV)
```

```
                    (EMIT-LAP '(:CALL SI::NON-LOCAL-RETURN-VALUES 2))))
               (EMIT-LAP `(:RETURN))                          ; This :RETURN will never be reached, but it makes stack
                                                              ; analysis happier.
               (PUSH (BLOCK-BLIP-VAR BLOCK)
                     *NON-LOCALS*)))))

(DEFUN GENCODE-SEGMENT (SEGMENT)
   (LET ((*SUPPRESS-POPS* NIL))
        (EMIT-LAP `(:TAG ,(SEGMENT-LOCAL-TAG SEGMENT)))
        (DOLIST (STMT (SEGMENT-STMTS SEGMENT))
            (GENCODE STMT :EFFECT))))

(DEFUN GENCODE-SETQ (NODE CONTEXT)
   (ASSERT (NOT (EQ (VARIABLE-KIND (SETQ-VAR NODE))
                    :FUNCTION))
           '(NODE)
           "BUG: Attempt to set a function variable.")
   (GENCODE (SETQ-VALUE NODE)
            :ARGUMENT)
   (EMIT-LAP `(:VAR_ ,(MAKE-LAP-VAR-REFERENCE (SETQ-VAR NODE))))
   (CASE CONTEXT
       (:EFFECT (WHEN (NOT *SUPPRESS-POPS*)
                      (EMIT-LAP '(:POP))))
       (:MV                                                   ; In MV context, we have to return a list of values.
          (EMIT-LAP-LIST '((:CONST NIL)
                           (:CALL CONS 2))))))

(DEFUN GENCODE-TAGBODY (NODE CONTEXT)

;;; Very much like the BLOCK case.  Sometimes we need to make a function call but usually we can avoid it to some degree.

   (COND
      ((TAGBODY-NEW-FRAME-P NODE)                             ; Construct a new lambda for the tagbody
       (LET (NEW-LAMBDA)
            (FRAME (:CURRENT-FRAME NODE :NAME (FORMAT NIL "tagbody in ~A" *FRAME-NAME*))
                   (LET ((STK-NUM (INCF *STACK-NUMBER*))
                         BLIP-GO-VAR OUR-NON-LOCALS)
                        (SETF (TAGBODY-FRAME NODE)
                              *CURRENT-FRAME*)
                        (SETF (TAGBODY-STK-NUM NODE)
                              STK-NUM)
                        (COND
                           ((TAGBODY-CLOSED-OVER-P NODE)
                            (SETQ *BLIP-VAR* `(:S SI::*CATCH-RETURN-FROM* ,(INCF *VAR-NUMBER*)))
                            (SETQ BLIP-GO-VAR (MAKE-LAP-VAR (TAGBODY-BLIP-VAR NODE)))
                            (EMIT-LAP-LIST `((:CONST ,*FRAME-NAME*)
                                             (:CONST NIL)
                                             (:CALL CONS 2)
                                             (:VAR_ ,*BLIP-VAR*)
                                             (:VAR_ ,BLIP-GO-VAR)
                                             (:POP)))
                            (SETQ *OTHERS* (LIST BLIP-GO-VAR *BLIP-VAR*)))
                           (T (SETQ *BLIP-VAR* `(:S SI::*CATCH-RETURN-TO* ,(INCF *VAR-NUMBER*)))
                              (SETQ *OTHERS* (LIST *BLIP-VAR*))))
                        (EMIT-LAP `(:NOTE-STACK ,STK-NUM))
                        (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:DO (SETF (SEGMENT-LOCAL-TAG SEGMENT)
                                                                                  (INCF *TAG-NUMBER*))
                                                                            (WHEN (SEGMENT-CLOSED-OVER-P SEGMENT)
                                                                                (SETF (SEGMENT-REMOTE-TAG SEGMENT)
                                                                                      (INCF *TAG-NUMBER*))))
                        (INTERCEPT-NON-LOCALS (SETQ OUR-NON-LOCALS (DELETE (TAGBODY-BLIP-VAR NODE)
                                                                           *NON-LOCALS*))
                              (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:DO (GENCODE-SEGMENT SEGMENT)))
                        (IF (EQ CONTEXT :MV)                  ; In MV context, we have to return a list of values.  We have to
                                                              ; CONS it freshly since MV-CALL uses NCONC to put the lists
                                                              ; together.
                            (EMIT-LAP-LIST '((:CONST NIL)
                                             (:CONST NIL)
                                             (:CALL CONS 2)
                                             (:RETURN)))
                            (EMIT-LAP-LIST '((:CONST NIL)
                                             (:RETURN))))
                        (WHEN (TAGBODY-CLOSED-OVER-P NODE)
                            (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:WHEN (SEGMENT-CLOSED-OVER-P SEGMENT)
                               IL:DO (EMIT-LAP-LIST `((:TAG ,(SEGMENT-REMOTE-TAG SEGMENT))
                                                      (:DSET-STACK ,STK-NUM)
                                                      (:JUMP ,(SEGMENT-LOCAL-TAG SEGMENT))))))
                        (SETQ NEW-LAMBDA
                              `(:LAMBDA (NIL :NAME ,*FRAME-NAME* ,@(AND *OTHERS* `(:OTHERS ,*OTHERS*))
                                             :BLIP
                                             ,*BLIP-VAR*
                                             ,@(AND OUR-NON-LOCALS `(:NON-LOCAL ,(MAPCAR #'VARIABLE-LAP-VAR
                                                                                         OUR-NON-LOCALS)))
                                             ,@(AND (TAGBODY-CLOSED-OVER-VARS NODE)
```

```
                                                    `(:CLOSED-OVER ,(MAPCAR #'VARIABLE-LAP-VAR (
                                                                                TAGBODY-CLOSED-OVER-VARS
                                                                                      NODE))))
                                                ,@(AND *LOCAL-FUNCTIONS* `(:LOCAL-FUNCTIONS ,*LOCAL-FUNCTIONS*)))
                                            ,@(END-LAP)))))
          ;; Generate a call to the new lambda.
          (EMIT-LAP `(:CALL ,NEW-LAMBDA 0))
          (WHEN (AND (EQ CONTEXT :EFFECT)
                     (NOT *SUPPRESS-POPS*))
              (EMIT-LAP '(:POP)))))
    (T (IF (NULL (TAGBODY-CLOSED-OVER-VARS NODE))
           (GENCODE-TAGBODY-INLINE NODE CONTEXT)
           (LET ((CODE (LET ((*CODE* (START-LAP)))
                         (GENCODE-TAGBODY-INLINE NODE CONTEXT)
                         (END-LAP))))
             (EMIT-LAP `(:CLOSE ,(MAPCAR #'VARIABLE-LAP-VAR (TAGBODY-CLOSED-OVER-VARS NODE))
                               ,@CODE)))))))
```

```
(DEFUN GENCODE-TAGBODY-INLINE (NODE CONTEXT)
  ;; We don't need a separate frame, so generate the code inline, setting up and taking down the blip stuff if necessary.
  (LET ((STK-NUM (INCF *STACK-NUMBER*)))
       (SETF (TAGBODY-FRAME NODE)
             *CURRENT-FRAME*)
       (SETF (TAGBODY-STK-NUM NODE)
             STK-NUM)
       (COND
          ((TAGBODY-CLOSED-OVER-P NODE)
           (LET ((BLIP-VAR (MAKE-LAP-VAR (TAGBODY-BLIP-VAR NODE)))
                 (END-TAG (INCF *TAG-NUMBER*)))
                (SET-UP-RETURN-TO)
                (EMIT-LAP-LIST `((:CONST TAGBODY)
                                 (:CONST ,*FRAME-NAME*)
                                 (:CALL CONS 2)
                                 (:VAR_ ,*BLIP-VAR*)
                                 (:VAR_ ,BLIP-VAR)
                                 (:POP)
                                 (:NOTE-STACK ,STK-NUM)))
                (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:DO (SETF (SEGMENT-LOCAL-TAG SEGMENT)
                                                                          (INCF *TAG-NUMBER*))
                                                                  (WHEN (SEGMENT-CLOSED-OVER-P SEGMENT)
                                                                      (SETF (SEGMENT-REMOTE-TAG SEGMENT)
                                                                            (INCF *TAG-NUMBER*))))
                (INTERCEPT-NON-LOCALS (DELETE (TAGBODY-BLIP-VAR NODE)
                                              *NON-LOCALS*)
                       (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:DO (GENCODE-SEGMENT SEGMENT)))
                (TAKE-DOWN-RETURN-TO)
                (EMIT-LAP `(:JUMP ,END-TAG))
                (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:WHEN (SEGMENT-CLOSED-OVER-P SEGMENT)
                   IL:DO (EMIT-LAP-LIST `((:TAG ,(SEGMENT-REMOTE-TAG SEGMENT))
                                          (:DSET-STACK ,STK-NUM)
                                          (:JUMP ,(SEGMENT-LOCAL-TAG SEGMENT)))))
                (EMIT-LAP `(:TAG ,END-TAG))))
          (T                                                    ; Simplest case: the tagbody isn't even closed over.
             (EMIT-LAP `(:NOTE-STACK ,STK-NUM))
             (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:DO (SETF (SEGMENT-LOCAL-TAG SEGMENT)
                                                                       (INCF *TAG-NUMBER*)))
             (IL:FOR SEGMENT IL:IN (TAGBODY-SEGMENTS NODE) IL:DO (GENCODE-SEGMENT SEGMENT))))
       ;; Arrange to return NIL from the TAGBODY.
       (ECASE CONTEXT
           (:EFFECT                                             ; Do nothing
                   )
           ((:ARGUMENT :RETURN) (EMIT-LAP '(:CONST NIL)))
           (:MV                                                 ; In MV context, we have to return a freshly-CONSed list of
                                                                ; values.
              (EMIT-LAP-LIST '((:CONST NIL)
                               (:CONST NIL)
                               (:CALL CONS 2)))))))
```

```
(DEFUN GENCODE-THROW (NODE CONTEXT)
  (GENCODE (THROW-TAG NODE)
           :ARGUMENT)
  (CASE CONTEXT
      ((:ARGUMENT :RETURN)
         (GENCODE (THROW-VALUE NODE)
                  CONTEXT)
         (EMIT-LAP-LIST '((:CALL SI::INTERNAL-THROW 2)
                          (:RETURN))))
      (OTHERWISE
         (GENCODE (THROW-VALUE NODE)
                  :MV)
         (EMIT-LAP-LIST '((:CALL SI::INTERNAL-THROW-VALUES 2)
                          (:RETURN)))                           ; The :RETURN will never be reached, but it makes
                                                                ; stack-analysis happier.
```

```
          )))

(DEFUN GENCODE-UNWIND-PROTECT (NODE CONTEXT)

;;; Funcall the body on the argument of the cleanup forms as a closure.

   (GENCODE (UNWIND-PROTECT-CLEANUP NODE)
         :ARGUMENT)
   (LET ((STMT-CODE (COLLECT-CODE (UNWIND-PROTECT-STMT NODE)
                                  :ARGUMENT)))
        (EMIT-LAP `(:CALL ,(POP STMT-CODE)
                          1))
        (ASSERT (NULL STMT-CODE)
                NIL "BUG: unwind-protect body code generated more than one LAP instruction"))
   (CASE CONTEXT
       (:EFFECT (WHEN (NOT *SUPPRESS-POPS*)
                      (EMIT-LAP '(:POP))))
       (:MV (EMIT-LAP '(:CALL IL:\\MVLIST 1)))))


(DEFUN GENCODE-VAR-REF (NODE CONTEXT)
   (LET ((VAR (VAR-REF-VARIABLE NODE)))
        (UNLESS (EQ CONTEXT :EFFECT)
            (IF (AND (EQ :GLOBAL (VARIABLE-SCOPE VAR))
                     (EQ :FUNCTION (VARIABLE-KIND VAR)))
                (EMIT-LAP-LIST `((:CONST ,(VARIABLE-NAME VAR))
                                 (:CALL SYMBOL-FUNCTION 1)))
                (EMIT-LAP `(:VAR ,(MAKE-LAP-VAR-REFERENCE VAR))))
            (WHEN (EQ CONTEXT :MV)                          ; In MV context, we have to return a list of the values.
                (EMIT-LAP-LIST '((:CONST NIL)
                                 (:CALL CONS 2)))))))

;; Policy variables.


(DEFPARAMETER *POP-SUPPRESSION-POLICY* NIL
   "If this is non-NIL, the code generator will suppress unnecessary pops.  This can increase stack usage.")


(DEFVAR *TAIL-RECURSION-POLICY* T
   "Set this to NIL to disable the tail-recursion optimization.")

;; Testing Code Generation


(DEFUN TEST-GENCODE (FN)
   (PPRINT (TEST-GENCODE1 FN))
   (FRESH-LINE)
   (VALUES))


(DEFUN TEST-GENCODE1 (FN)
   (DESTRUCTURING-BIND (IGNORE NAME ARG-LIST &BODY BODY)
          (IL:GETDEF FN 'IL:FUNCTIONS)
          (MULTIPLE-VALUE-BIND (FORMS DECLS)
              (PARSE-BODY BODY NIL T)
            (LET ((*ENVIRONMENT* (MAKE-CHILD-ENV T))
                  (*PROCESSED-FUNCTIONS* NIL)
                  (*UNKNOWN-FUNCTIONS* NIL)
                  (*CONSTANTS-HASH-TABLE* (MAKE-HASH-TABLE)))
                 (COMPILE-ONE-LAMBDA FN `(LAMBDA ,ARG-LIST ,@DECLS (BLOCK ,NAME ,@FORMS)))))))

;; Arrange to use the correct compiler.

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

;; Arrange to use the proper makefile environment

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

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

FUNCTION INDEX

VARIABLE INDEX

MACRO INDEX

PROPERTY INDEX

STRUCTURE INDEX