

File created: 1-Dec-92 02:18:56 {Pele:mv:envos}<LispCore>Sources>D-ASSEM.;12

changes to: (IL:FUNCTIONS EMIT-BYTE FIXUP-PTR FIXUP-PTR-NO-REF FIXUP-SYMBOL FIXUP-NTENTRY INTERN-DCODE)

previous date: 17-Nov-92 02:55:57 {Pele:mv:envos}<LispCore>Sources>D-ASSEM.;11

Read Table: XCL

Package: D-ASSEM

Format: XCCS

; Copyright (c) 1986, 1987, 1988, 1990, 1991, 1992 by Xerox Corporation. All rights reserved.

(IL:RPAQQ IL:D-ASSEMBLERS

(

;;; D-machine Assembler.

```
(IL:FILES IL:D-ASSEM-PACKAGE)
(IL:COMS
  ;; Data structures and utilities
  (IL:STRUCTURES DCODE DJUMP DLAMBDA DTAG DVAR)
  (IL:FUNCTIONS RELEASE-DCODE)
  (IL:FUNCTIONS CREATE-HUNK TYPE-NAME-FROM-SIZE)
  (IL:FUNCTIONS COPY-LAP-FN COPY-LAP-CODE)
  (IL:FUNCTIONS MAXF))
(IL:COMS
  ;; Handy constants
  (IL:VARIABLES +IVAR-CODE+ +PVAR-CODE+ +FVAR-CODE+)
  (IL:VARIABLES +LAMBDA-SPREAD+ +NLAMBDA-SPREAD+ +LAMBDA-NO-SPREAD+ +NLAMBDA-NO-SPREAD+)
  (IL:VARIABLES +CONSTANT-OPCODES+))
(IL:COMS
  ;; Opcode generation
  (IL:VARIABLES *BYTES* *BYTE-COUNT*)
  (IL:FUNCTIONS START-BYTES EMIT-BYTE EMIT-BYTE-LIST END-BYTES)
  (IL:FUNCTIONS CHOOSE-OP FETCH-HUNK REF-VAR STORE-VAR MAX-ARG PUSH-INTEGER))
(IL:COMS
  ;; Main driving
  (IL:VARIABLES *DTAG-ENV* *DVAR-ENV* *STACK-ENV*)
  (IL:VARIABLES *HUNK-MAP* *DCODE* *LEVEL*)
  (IL:FUNCTIONS ASSEMBLE-FUNCTION DLAMBDA-FROM-LAMBDA DCODE-FROM-DLAMBDA))
(IL:COMS
  ;; Digesting the function
  (IL:VARIABLES *HUNK-SIZE* *PVAR-COUNT* *FREE-VARS* *BOUND-SPECIALS*)
  (IL:VARIABLES +MAX-ALLOWABLE-PVAR-COUNT+ +MAX-ALLOWABLE-SPECIAL-COUNT+ +SLOW-FVAR-SLOT+)
  (IL:FUNCTIONS DIGEST-FUNCTION DETERMINE-LOCAL-FN-LEXICAL-LEVEL DIGEST-CODE STORE-DIGEST-INFO)
  (IL:FUNCTIONS DVAR-FROM-LAP-VAR LAP-VAR-ID INSTALL-LOCAL INSTALL-VAR INTERN-VAR INTERN-TAG))
(IL:COMS
  ;; Function entry code
  (IL:FUNCTIONS EASY-ENTRY-P GENERATE-EASY-ENTRY)
  (IL:FUNCTIONS GENERATE-HARD-ENTRY GENERATE-ARG-CHECK GENERATE-KEY GENERATE-OPT-AND-REST))
(IL:COMS
  ;; Stack analysis
  (IL:VARIABLES *ENDING-DEPTH*)
  (IL:FUNCTIONS GATHER-TAGS GATHER-ROOTS REACH-TAGS STACK-ANALYZE STACK-ANALYZE-CODE))
(IL:COMS
  ;; The guts of assembly
  (IL:FUNCTIONS ASSEMBLE ASSEMBLE-CODE))
(IL:COMS
  ;; Jump resolution
  (IL:VARIABLES *JUMP-LIST*)
  (IL:VARIABLES +JUMP-CHOICES+ +JUMP-RANGE-SIZE-MAP+ +JUMP-SIZES+)
  (IL:FUNCTIONS RESOLVE-JUMPS REDUCE-UNCERTAINTY SPLICE-IN-JUMPS COMPUTE-JUMP-SIZE)
  (IL:COMS
    (IL:FUNCTIONS PRETTY-JUMPS)))
(IL:COMS
  ;; Conversion to binary
  (IL:VARIABLES *LOCAL-FN-FIXUPS*)
  (IL:FUNCTIONS CONVERT-TO-BINARY))
(IL:COMS
  ;; Setting up the debugging information
  (IL:FUNCTIONS COMPUTE-DEBUGGING-INFO))
(IL:COMS
  ;; Fixup resolution and DCODE interning
  (IL:FUNCTIONS START-PC-FROM-NT-COUNT START-PC-FROM-NT-COUNT-LOCAL ALLOCATE-CODE-BLOCK
```

```

FIXUP-PTR FIXUP-PTR-NO-REF FIXUP-SYMBOL FIXUP-NTENTRY FIXUP-WORD INTERN-DCODE
PERFORM-LOCAL-FN-FIXUPS))

```

```

;; Arrange for the correct compiler to be used

```

```

(IL:PROP IL:FILETYPE IL:D-ASSEM)

```

```

;; Arrange for the proper makefile environment

```

```

(IL:PROP IL:MAKEFILE-ENVIRONMENT IL:D-ASSEM)

```

```

(IL:DECLARE\ IL:EVAL@COMPILE IL:DONTCOPY (IL:FILES (IL:LOADCOMP)
IL:LLBASIC IL:LLCODE IL:LLGC IL:MODARITH))))

```

```

;;; D-machine Assembler.

```

```

(IL:FILESLOAD IL:D-ASSEM-PACKAGE)

```

```

;; Data structures and utilities

```

```

(DEFSTRUCT DCODE

```

```

  FRAME-NAME

```

```

  NLOCALS

```

```

  NFREEVARS

```

```

  ARG-TYPE

```

```

  NUM-ARGS

```

```

  (NAME-TABLE NIL)

```

```

  DEBUGGING-INFO CODE-ARRAY (FN-FIXUPS NIL)

```

```

  (SYM-FIXUPS NIL)

```

```

  (LIT-FIXUPS NIL)

```

```

  (TYPE-FIXUPS NIL)

```

```

  CLOSURE-P

```

```

  (LOCAL-FN-FIXUPS NIL)

```

```

  (INTERN-RESULT NIL))

```

```

(DEFSTRUCT DJUMP

```

```

  KIND

```

```

  TAG

```

```

  PTR

```

```

  MIN-PC

```

```

  MIN-SIZE

```

```

  FORWARD-P

```

```

  SIZE-UNCERTAINTY)

```

```

(DEFSTRUCT (DLAMBDA (:CONSTRUCTOR MAKE-DLAMBDA

```

```

      (REQUIRED OPTIONAL REST KEY ALLOW-OTHER-KEYS OTHERS NAME ARG-TYPE BLIP
      CLOSED-OVER NON-LOCAL BODY LOCAL-FUNCTIONS)))

```

```

  REQUIRED

```

```

  OPTIONAL

```

```

  REST

```

```

  KEY

```

```

  ALLOW-OTHER-KEYS

```

```

  OTHERS

```

```

  NAME

```

```

  ARG-TYPE

```

```

  BLIP

```

```

  CLOSED-OVER

```

```

  NON-LOCAL

```

```

  BODY

```

```

  LOCAL-FUNCTIONS)

```

```

(DEFSTRUCT DTAG

```

```

;;; LEVEL is the lexical level of this tag, for use by the stack analyzer.

```

```

;;; STACK-DEPTH is a pair <binding-depth . depth> representing the state of the stack analyzer last time it was here.

```

```

;;; PTR is the tail of the code list starting with this tag, used by the stack analyzer.

```

```

;;; PC is the final location of this tag, after jump resolution.

```

```

;;; MIN-PC is the least location at which this tag could end up, used during jump resolution.

```

```

;;; PC-UNCERTAINTY is the amount of slack there is in the final location of this tag, used during jump resolution.

```

```

;;; REACHABLE? is T if we have discovered a way to reach this tag. Used during a pre-pass of stack analysis.

```

```

  LEVEL

```

```

  STACK-DEPTH

```

```

  PTR

```

```

  PC

```

```

  MIN-PC

```

```

  PC-UNCERTAINTY

```

```

  (REACHABLE? NIL))

```

```
(DEFSTRUCT DVAR
```

```
  KIND
  LEVEL
  SLOT
  NAME)
```

```
(DEFUN RELEASE-DCODE (DCODE)
  (LET ((LOCAL-FN-FIXUPS (PROG1 (DCODE-LOCAL-FN-FIXUPS DCODE)
                                (SETF (DCODE-LOCAL-FN-FIXUPS DCODE)
                                      NIL))))
    (DOLIST (FIXUP LOCAL-FN-FIXUPS)
      ((RELEASE-DCODE (FIRST FIXUP))
       (RELEASE-DCODE (THIRD FIXUP))))))
```

```
(DEFUN CREATE-HUNK (HUNK-SIZE MY-SLOT PREV-SLOT POP-P)
```

;; Emit code to create a hunk of the given size and store it into PVAR my-slot. If prev-slot is non-NIL, also emit code to link the new hunk to the one in
 ;; that slot. If pop-p is non-NIL then don't leave the hunk on the stack.

```
  (EMIT-BYTE-LIST '(IL:SICX (:TYPE , (TYPE-NAME-FROM-SIZE HUNK-SIZE))
                        IL:CREATECELL
                        ,@ (AND PREV-SLOT ' ( ,@ (CHOOSE-OP ' (IL:PVAR . IL:PVARX)
                                                            PREV-SLOT)
                                     IL:RPLPTR.N 0))
                        ,@ (STORE-VAR MY-SLOT POP-P))))
```

```
(DEFUN TYPE-NAME-FROM-SIZE (LEN)
  (IL:PACK* '\PTRHUNK (IL:for| HUNK-SIZE IL:|in| IL:\HUNK.PTRSIZE IL:|when| (<= LEN HUNK-SIZE)
                IL:|do| (RETURN HUNK-SIZE) IL:|finally| (ERROR "Can't make a hunk that big: ~S" LEN))))
```

```
(DEFUN COPY-LAP-FN (FN)
  (CAR (COPY-LAP-CODE (LIST FN))))
```

```
(DEFUN COPY-LAP-CODE (CODE)
  (IL:FOR INST IL:IN CODE
    IL:COLLECT
      (CASE (FIRST INST)
        ((:CONST) ; Don't copy the constant itself; it might be shared.
         (COPY-LIST INST))
        ((:LAMBDA)
         (FLET
            ((COPY-LAMBDA-ARGS
              (ARGS)
              (UNLESS (NULL ARGS)
                (WITH-COLLECTION
                  (COLLECT (COPY-TREE (POP ARGS))) ; required.
                  (LOOP
                     (WHEN (NULL ARGS)
                       (RETURN))
                     (LET* ((KEY (POP ARGS))
                           (VAL (POP ARGS)))
                       (COLLECT (COPY-TREE KEY))
                       (COLLECT (CASE KEY
                                ((:LOCAL-FUNCTIONS)
                                 (MAPCAR #'(LAMBDA (FN-PAIR)
                                              (CONS (COPY-TREE (CAR FN-PAIR))
                                                    (COPY-LAP-CODE (CDR FN-PAIR))))
                                VAL))
                                (OTHERWISE (COPY-TREE VAL))))))))
                     (LIST* :LAMBDA (COPY-LAMBDA-ARGS (SECOND INST))
                           (COPY-LAP-CODE (CDDR INST))))
                     ((:CLOSE) (LIST* :CLOSE (COPY-TREE (SECOND INST))
                                       (COPY-LAP-CODE (CDDR INST))))
                     ((:CALL) (IF (AND (CONSP (SECOND INST))
                                         (EQ :LAMBDA (FIRST (SECOND INST))))
                                   (LIST* :CALL (LIST* :LAMBDA (COPY-TREE (SECOND (SECOND INST)))
                                                         (COPY-LAP-CODE (CDDR (SECOND INST))))
                                       (COPY-TREE (CDDR INST)))
                                   (COPY-TREE INST)))
                     (OTHERWISE (COPY-TREE INST))))
            (OTHERWISE (COPY-TREE INST))))))
```

```
(DEFINE-MODIFY-MACRO MAXF (&REST COMPILER::NEW-VALUES) MAX)
```

;; Handy constants

```
(DEFCONSTANT +IVAR-CODE+ 0
  "Code in name-table for IVARs")
```

```

{MEDLEY}<sources>D-ASSEM.;1

(DEFCONSTANT +PVAR-CODE+ 2
  "Code in name-table for PVARs")

(DEFCONSTANT +FVAR-CODE+ 3
  "Code in name-table for FVARs")

(DEFCONSTANT +LAMBDA-SPREAD+ 0
  "ARGTYPE value for lambda spread functions")

(DEFCONSTANT +NLAMBDA-SPREAD+ 1
  "ARGTYPE value for nlambda spread functions")

(DEFCONSTANT +LAMBDA-NO-SPREAD+ 2
  "ARGTYPE value for lambda no-spread functions")

(DEFCONSTANT +NLAMBDA-NO-SPREAD+ 3
  "ARGTYPE value for nlambda no-spread functions")

(DEFCONSTANT +CONSTANT-OPCODES+ '((0 . IL:\'0)
                                   (1 . IL:\'1)
                                   (NIL . IL:\'NIL)
                                   (T . IL:\'T))
  "An AList of all constants with dedicated opcodes.")

;; Opcode generation

(DEFVAR *BYTES* NIL
  "The data-structure holding the bytes of the current function. Use (start-bytes) to create an empty one,
  (emit-byte) or (emit-op) to add more bytes on the end, and (end-bytes) to close it off and get an array of
  the bytes.")

(DEFVAR *BYTE-COUNT* 0
  "The number of bytes put on *bytes* so far.")

(DEFUN START-BYTES ()
  NIL)

(DEFUN EMIT-BYTE (BYTE)
  ;; Given the symbolic representation of a byte/opcode to be emitted as part of the assembled code for a function, emit it. Actually, do some fix-ups at
  ;; the same time, and record some information for other parts of the assembler.
  ;; *BYTES* = the list of emitted bytes, in reverse order (we push onto it)
  ;; *BYTE-COUNT* = running count of bytes emitted so far.
  ;; *JUMP-LIST* = list of jumps and jump-target tags, for later jump resolution.
  (COND
    ((CONSP BYTE)
     (CASE (FIRST BYTE)
       ((:TAG)
        (SETF (DTAG-MIN-PC (SECOND BYTE))
              *BYTE-COUNT*)
        (PUSH (SECOND BYTE)
              *JUMP-LIST*))
        ((:JUMP :FJUMP :TJUMP :NFJUMP :NTJUMP)
         (PUSH BYTE *BYTES*)
         (PUSH (MAKE-DJUMP :KIND (FIRST BYTE)
                          :TAG
                          (SECOND BYTE)
                          :PTR *BYTES* :MIN-PC *BYTE-COUNT*)
               *JUMP-LIST*))
         (INCF *BYTE-COUNT* (SECOND (ASSOC (FIRST BYTE)
                                           +JUMP-SIZES+))))
        ((:PUSH-TAG)
         (PUSH 'IL:SICX *BYTES*)
         (PUSH BYTE *BYTES*)
         (PUSH 0 *BYTES*)
         (INCF *BYTE-COUNT* 3))
        ((:SYM :FN)
         ;; Symbol (e.g., for GVAR) or function name inline in code.
         (PUSH BYTE *BYTES*)
         (COND
           ((IL:FMEMB :4-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
            (INCF *BYTE-COUNT*)
            (INCF *BYTE-COUNT*))
           (T))))
    (T)))

```

```

      (PUSH 0 *BYTES*)
      (PUSH 0 *BYTES*)
      ((IL:FMEMB :3-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
        (INCF *BYTE-COUNT*))
      (PUSH 0 *BYTES*))
    (PUSH 0 *BYTES*)
    (INCF *BYTE-COUNT* 2))
  ((:TYPE)
    ;; Type numbers are 11 bits, and fit in 2 bytes of a SICX.
    (PUSH BYTE *BYTES*)
    (PUSH 0 *BYTES*)
    (INCF *BYTE-COUNT* 2))
  ((:LAMBDA :LIT :LOCAL-FUNCTION)
    (PUSH BYTE *BYTES*)
    (COND
      ((IL:FMEMB :4-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
        (PUSH 0 *BYTES*)
        (PUSH 0 *BYTES*)
        (PUSH 0 *BYTES*)
        (INCF *BYTE-COUNT* 4))
      (T (PUSH 0 *BYTES*)
        (PUSH 0 *BYTES*)
        (INCF *BYTE-COUNT* 3))))
  ((IL:ATOM)
    ;; ByteCompiler-style fixup, here because of a DOPVAL. The ByteCompiler put its fixup bytes AFTER the padding bytes, so we have
    ;; to rearrange things in the byte list. The padding bytes have been emitted already, so pop them off. Then emit the ATOM byte, and
    ;; put out new padding bytes. The net increase in bytes is 1, so incf *BYTE-COUNT*.
    (POP *BYTES*)
    (COND
      ((IL:FMEMB :4-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
        (POP *BYTES*)
        (POP *BYTES*))
      ((IL:FMEMB :3-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
        (POP *BYTES*))
      (PUSH (LIST ' :SYM (CDR BYTE))
        *BYTES*))
    (COND
      ((IL:FMEMB :4-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
        (PUSH 0 *BYTES*)
        (PUSH 0 *BYTES*))
      ((IL:FMEMB :3-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
        (PUSH 0 *BYTES*))
      (PUSH 0 *BYTES*)
      (INCF *BYTE-COUNT*))
    ((IL:PTR)
      ;; ByteCompiler-style fixup, here because of a DOPVAL. The ByteCompiler put its fixup bytes AFTER the padding bytes, so we have
      ;; to rearrange things in the byte list.
      (COND
        ((IL:FMEMB :4-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
          (POP *BYTES*)
          (POP *BYTES*)
          (POP *BYTES*)
          (PUSH (LIST ' :LIT (CDR BYTE))
            *BYTES*)
          (PUSH 0 *BYTES*)
          (PUSH 0 *BYTES*)
          (PUSH 0 *BYTES*)
          (INCF *BYTE-COUNT*))
        ((IL:FMEMB :3-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE COMPILER::*ENVIRONMENT*))
          (POP *BYTES*)
          (POP *BYTES*)
          (PUSH (LIST ' :LIT (CDR BYTE))
            *BYTES*)
          (PUSH 0 *BYTES*)
          (PUSH 0 *BYTES*)
          (INCF *BYTE-COUNT*)))
        (OTHERWISE
          (PUSH BYTE *BYTES*)
          (INCF *BYTE-COUNT*)))
      (T (PUSH BYTE *BYTES*)
        (INCF *BYTE-COUNT*)))

(DEFUN EMIT-BYTE-LIST (L)
  (IL:|for| BYTE IL:|in| L IL:|do| (EMIT-BYTE BYTE)))

(DEFUN END-BYTES ()
  (NREVERSE *BYTES*))

(DEFUN CHOOSE-OP (CHOICES ARG)
  (IF (<= ARG (MAX-ARG (CAR CHOICES)))
    `((, (CAR CHOICES)

```

```

    , ARG))
  ` (, (CDR CHOICES)
    , (IL:LLSH ARG 1)))

```

```

(DEFUN FETCH-HUNK (LEVEL)
  (IF (ZEROP LEVEL) ; No environment pointer in the base lexical level.
    ' (IL:\'NIL)
    (LET* ((MAP-ENTRY (IL:|find| ENTRY IL:|in| (REVERSE *HUNK-MAP*) IL:|suchthat| (<= LEVEL (CAR ENTRY))))
      (HUNK-LEVEL (CAR MAP-ENTRY))
      (HUNK-SLOT (CDR MAP-ENTRY))
      ` (, @ (CHOOSE-OP ' (IL:PVAR . IL:PVARX)
        HUNK-SLOT)
        , @ (IL:|for| I IL:|from| 1 IL:|to| (- HUNK-LEVEL LEVEL) IL:|join| (LIST 'IL:GETBASEPTR.N 0))))))

```

```

(DEFUN REF-VAR (VAR)

```

;;; Return a list of instructions to push the given variable's value onto the stack.

```

  (IF (DVAR-P VAR)
    (ECASE (DVAR-KIND VAR)
      ((:LOCAL) (CHOOSE-OP ' (IL:PVAR . IL:PVARX)
        (DVAR-SLOT VAR)))
      ((:ARGUMENT) (CHOOSE-OP ' (IL:IVAR . IL:IVARX)
        (DVAR-SLOT VAR)))
      ((:FREE) (IF (EQL (DVAR-SLOT VAR)
        +SLOW-FVAR-SLOT+)
        ;; This one is icky. It couldn't fit in the frame, so we use a call to SYMBOL-VALUE to find the value. Ugh.
        ` (IL:ACONST (:SYM , (DVAR-NAME VAR))
          IL:FNI
          (:FN SYMBOL-VALUE))
        (CHOOSE-OP ' (IL:FVAR . IL:FVARX)
          (DVAR-SLOT VAR)))
      ((:CLOSED) ` (, @ (FETCH-HUNK (DVAR-LEVEL VAR))
        IL:GETBASEPTR.N
        , (IL:LLSH (DVAR-SLOT VAR)
          1)))
      ((:FUNCTION)
        (ASSERT (NOT (NULL (DVAR-LEVEL VAR)))
          ' (VAR)
          "BUG: The local function ~A should have a lexical level by now."
          (DVAR-NAME VAR))
        (COND
          ((AND NIL (ZEROP (DVAR-LEVEL VAR))) ; We used to do something different for empty environments.
            ` (IL:GCONST (:LOCAL-FUNCTION , VAR)))
          (T ` (IL:SICX (:TYPE IL:COMPILED-CLOSURE)
            IL:CREATECELL IL:GCONST (:LOCAL-FUNCTION , VAR)
            IL:RPLPTR.N 0 , @ (AND (NOT (ZEROP (DVAR-LEVEL VAR)))
              ` (, @ (FETCH-HUNK (DVAR-LEVEL VAR))
                IL:RPLPTR.N 2)))))))
    (IF (AND (CONSP VAR)
      (EQ (FIRST VAR)
        ' :G))
      ` (IL:GVAR (:SYM , (SECOND VAR)))
      (ERROR "BUG: This variable is neither global nor a DVAR: ~S" VAR)))

```

```

(DEFUN STORE-VAR (VAR-OR-SLOT POP-P)

```

;;; Return a list of instructions to store the value on the top of stack into the given variable. If a slot number is given instead, it is assumed to refer to a
 ;;; PVAR. If POP-P is non-NIL, don't leave the value on the stack.

```

  (LET (KIND SLOT)
    (ETYPESCASE VAR-OR-SLOT
      (FIXNUM
        (SETQ SLOT VAR-OR-SLOT)
        (SETQ KIND :LOCAL))
      (CONS
        (UNLESS (EQ (FIRST VAR-OR-SLOT)
          ' :G)
          (ERROR "BUG: This variable is neither a global nor a DVAR: ~S" VAR-OR-SLOT))
        (SETQ KIND :GLOBAL)
        (SETQ SLOT (SECOND VAR-OR-SLOT)))
      (DVAR
        (SETQ SLOT (DVAR-SLOT VAR-OR-SLOT))
        (SETQ KIND (DVAR-KIND VAR-OR-SLOT)))
    (ECASE KIND
      ((:LOCAL) (IF POP-P
        (IF (<= SLOT (MAX-ARG 'IL:PVAR_^))
          ` ((IL:PVAR_^
            , SLOT))
          ` (, @ (CHOOSE-OP ' (IL:PVAR_ . IL:PVARX_)
            SLOT)
            IL:POP))
        (CHOOSE-OP ' (IL:PVAR_ . IL:PVARX_)

```

```

        SLOT)))
((:ARGUMENT) `(IL:IVARX_ , (IL:LLSH SLOT 1)
               ,@ (AND POP-P ' (IL:POP))))
((:FREE) (IF (EQL SLOT +SLOW-FVAR-SLOT+)
             ;; This one is icky. It couldn't fit in the frame, so we use a call to SET to store the value. Ugh.
             `(IL:ACONST (:SYM , (DVAR-NAME VAR-OR-SLOT))
                       IL:SWAP IL:FN2 (:FN SET)
                       ,@ (AND POP-P ' (IL:POP))))
             `(IL:FVARX_ , (IL:LLSH SLOT 1)
                       ,@ (AND POP-P ' (IL:POP))))))
((:CLOSED) ` (, @ (AND (NOT POP-P)
                       , (IL:COPY))
              , @ (FETCH-HUNK (DVAR-LEVEL VAR-OR-SLOT))
              IL:SWAP IL:RPLPTR.N , (IL:LLSH SLOT 1)
              IL:POP))
((:GLOBAL) `(IL:GVAR_ (:SYM , SLOT)
                    ,@ (AND POP-P ' (IL:POP))))))

```

```

(DEFUN MAX-ARG (OPCODE)
  (LET ((RANGE (IL:|fetch| IL:OP# IL:|of| (IL:\\FINDOP OPCODE))))
    (- (SECOND RANGE)
       (FIRST RANGE))))

```

```

(DEFUN PUSH-INTEGER (N)
  (COND
    ((ZEROP N)
     ' (IL:\\'0))
    ((= N 1)
     ' (IL:\\'1))
    ((<= -256 N -1)
     `(IL:SNIC , (+ N 256)))
    ((<= 0 N 255)
     `(IL:SIC ,N))
    ((<= 255 N 65535)
     `(IL:SICX , (IL:LRSH N 8)
               , (LOGAND N 255)))
    (T `(IL:GCONST (:LIT ,N)))))

```

;; Main driving

```

(DEFVAR *DTAG-ENV* NIL
  "A hash-table mapping the EQL-unique ID of a LAP tag into the DTAG structure used by D-ASSEM.")

```

```

(DEFVAR *DVAR-ENV* NIL
  "A hash-table mapping the EQL-unique ID of a LAP variable into the DVAR structure used by D-ASSEM.")

```

```

(DEFVAR *STACK-ENV* NIL

```

;; Hash-table mapping stack-level identifiers to a pair (depth . binding-depths).

)

```

(DEFVAR *HUNK-MAP* NIL
  "An Alist mapping lexical level numbers into the PVAR number of a slot in the current frame holding the hunk
  for that level.")

```

```

(DEFVAR *DCODE* 0
  "The currently-under-construction DCODE structure.")

```

```

(DEFVAR *LEVEL* 0
  "The current lexical nesting level.")

```

```

(DEFUN ASSEMBLE-FUNCTION (LAP-FN)

```

;; LAP-FN is a LAP-format function description (a LAMBDA). Return the DCODE structure that results from assembling it into D-machine bytecodes.
 ;; The code is copied so as to avoid problems with shared structure when we diddle with it.

```

  (LET ((*DVAR-ENV* (MAKE-HASH-TABLE :TEST 'EQL))
        (*DTAG-ENV* (MAKE-HASH-TABLE :TEST 'EQL))
        (*STACK-ENV* (MAKE-HASH-TABLE :TEST 'EQL))
        (*LOCAL-FN-FIXUPS* NIL))
    (UNWIND-PROTECT
      (LET ((DCODE (DCODE-FROM-DLAMBDA (DLAMBDA-FROM-LAMBDA (COPY-LAP-FN LAP-FN)
                                                             0)))
            (SETF (DCODE-CLOSURE-P DCODE)
                  :FUNCTION)
            DCODE)
        ; The top-level guy is always a :FUNCTION.

```

;; Break all of the circularities created during this assembly.

```
(MAPHASH #'(LAMBDA (ID TAG)
              (DECLARE (IGNORE ID))
              (SETF (DTAG-PTR TAG)
                    NIL))
  *DTAG-ENV*))
```

```
(DEFUN DLAMBDA-FROM-LAMBDA (LAP-FN)
  "Break out the fields of a LAP lambda and return them in the form of a DLAMBDA structure."
  (DESTRUCTURING-BIND ((REQUIRED &KEY OPTIONAL REST KEY ALLOW-OTHER-KEYS OTHERS NAME ARG-TYPE BLIP CLOSED-OVER
                          NON-LOCAL LOCAL-FUNCTIONS)
    &REST BODY)
    (CDR LAP-FN)
    (MAKE-DLAMBDA REQUIRED OPTIONAL REST KEY ALLOW-OTHER-KEYS OTHERS NAME ARG-TYPE BLIP CLOSED-OVER
      (MAPCAR #'LAP-VAR-ID NON-LOCAL) ; non-local
      BODY
      (MAPCAR #'(LAMBDA (PAIR)
                      (CONS (FIRST PAIR)
                            (DLAMBDA-FROM-LAMBDA (SECOND PAIR))))
        LOCAL-FUNCTIONS) ; local-functions
      )))
```

```
(DEFUN DCODE-FROM-DLAMBDA (DLAMBDA LEVEL)
```

;;; LEVEL is the length of the chain of hunks that will be passed to the result of assembling DLAMBDA at runtime.

```
(LET ((*DCODE* (MAKE-DCODE :FRAME-NAME (DLAMBDA-NAME DLAMBDA)
                           :CLOSURE-P :CLOSURE)
      ; By default, no DCODE gets wrapped up into a closure object.
      ; The values for this field are somewhat misleading.
```

```
    )
    (*HUNK-MAP* (AND (> LEVEL 0)
                     `((,LEVEL . 0))))
    (*HUNK-SIZE* (IF (> LEVEL 0)
                     1
                     0))
    (*BYTES* (START-BYTES))
    (*BYTE-COUNT* 0)
    (*JUMP-LIST* NIL)
    (*PVAR-COUNT* (IF (> LEVEL 0)
                      1
                      0))
    (*LEVEL* LEVEL)
    (EASY-ENTRY (EASY-ENTRY-P DLAMBDA)))
```

;; Pass 0: Intern all of the variables and tags

```
(DIGEST-FUNCTION DLAMBDA EASY-ENTRY)
(LET ((FIXUP-DESC (FIND DLAMBDA *LOCAL-FN-FIXUPS* :TEST 'EQ :KEY 'CADR)))
  (WHEN FIXUP-DESC
    (SETF (CADR FIXUP-DESC)
          *DCODE*)))
```

;; Pass 1: Translate LAP code into opcodes and bytes.

```
(IF EASY-ENTRY
  (GENERATE-EASY-ENTRY DLAMBDA)
  (GENERATE-HARD-ENTRY DLAMBDA))
(STACK-ANALYZE (DLAMBDA-BODY DLAMBDA))
(ASSEMBLE (DLAMBDA-BODY DLAMBDA))
(EMIT-BYTE 'IL:-X-)
(SETQ *BYTES* (END-BYTES))
```

;; Pass 1-1/2: Resolve the uncertainty in jump sizes and distances.

```
(UNLESS (NULL *JUMP-LIST*)
  (RESOLVE-JUMPS (REVERSE *JUMP-LIST*)))
```

;; Pass 2: Convert the byte-list into its final binary form and create the fixup lists. This pass also performs the recursion necessary to compile nested lambdas.

```
(CONVERT-TO-BINARY *BYTES*)
```

;; Fill in the last few bits of the DCODE and quit.

```
(COMPUTE-DEBUGGING-INFO DLAMBDA)
*DCODE*)
```

;; Digesting the function

```
(DEFVAR *HUNK-SIZE* 0
  "The number of hunk slots allocated so far.")
```

```
(DEFVAR *PVAR-COUNT* 0
  "The number of PVAR allocated so far.")
```



```
(DEFVAR *FREE-VARS* NIL
  "An Alist mapping DVARS for the free variables in a function into the number of times they're referenced in
  the function.")
```

```
(DEFVAR *BOUND-SPECIALS* NIL
  "A list of the special variables bound in this frame.")
```

```
(DEFCONSTANT +MAX-ALLOWABLE-PVAR-COUNT+ 128
```

```
;;; On the D-machine, there is a hard limit to the number of bound variables we can allow. This is that limit.
```

```
)
```

```
(DEFCONSTANT +MAX-ALLOWABLE-SPECIAL-COUNT+ 119
```

```
;;; Because of page-boundary constraints, there is a limit to how many entries there can be in the name table of a frame. This is that limit. Note that it
;;; refers both to bound and free specials.
```

```
)
```

```
(DEFCONSTANT +SLOW-FVAR-SLOT+ -1
```

```
;;; This is the slot number for free variables that have to be accessed the slow way, using SYMBOL-VALUE and SET.
```

```
)
```

```
(DEFUN DIGEST-FUNCTION (DLAMBDA EASY-ENTRY)
  (LET ((CLOSED-OVER (DLAMBDA-CLOSED-OVER DLAMBDA))
        (IVAR-COUNT 0)
        (*FREE-VARS* NIL)
        (*BOUND-SPECIALS* NIL))
    ;; Allocate slots for the top-level hunk and the blip slot variable, if they're needed.
    (WHEN (NOT (NULL (DLAMBDA-BLIP DLAMBDA))) ; This can lead to a wasted PVAR0, but I'm not losing sleep over
                                                ; it.
      (INSTALL-VAR (DLAMBDA-BLIP DLAMBDA)
                    NIL :LOCAL 1)
      (SETQ *PVAR-COUNT* 2))
    (WHEN (NOT (NULL CLOSED-OVER))
      (INCF *LEVEL*) ; The lexical level just changed.
      (PUSH (CONS *LEVEL* *PVAR-COUNT*)
            *HUNK-MAP*)
      (INCF *PVAR-COUNT*))
    ;; Intern the local functions
    (IL:|for| FN-PAIR IL:|in| (DLAMBDA-LOCAL-FUNCTIONS DLAMBDA)
      IL:|do| (LET ((DVAR (INSTALL-VAR (CAR FN-PAIR)
                                       CLOSED-OVER :FUNCTION NIL)))
                (PUSH (LIST DVAR (CDR FN-PAIR)
                            *DCODE*)
                      *LOCAL-FN-FIXUPS*)
                (SETF (CAR FN-PAIR)
                      DVAR)))
    ;; Intern the required parameters.
    (IL:|for| TAIL IL:|on| (DLAMBDA-REQUIRED DLAMBDA) IL:|do| (SETF (CAR TAIL)
                                                                    (INSTALL-VAR (CAR TAIL)
                                                                    CLOSED-OVER :ARGUMENT IVAR-COUNT))
                        (INCF IVAR-COUNT))
    ;; And then the optional parameters and their supplied-p colleagues.
    (IL:|for| OPT-VAR IL:|in| (DLAMBDA-OPTIONAL DLAMBDA) IL:|do| (COND
                                                                (EASY-ENTRY (SETF (FIRST OPT-VAR)
                                                                    (INSTALL-VAR (FIRST OPT-VAR)
                                                                    CLOSED-OVER :ARGUMENT
                                                                    IVAR-COUNT))
                                                                (INCF IVAR-COUNT))
                                                                (T (SETF (FIRST OPT-VAR)
                                                                    (INSTALL-LOCAL (FIRST OPT-VAR)
                                                                    CLOSED-OVER))
                                                                (DIGEST-CODE (SECOND OPT-VAR))
                                                                (SETF (THIRD OPT-VAR)
                                                                    (INSTALL-LOCAL (THIRD OPT-VAR)
                                                                    CLOSED-OVER))))))
    ;; Next comes the rest and keyword parameters.
    (WHEN (NOT EASY-ENTRY)
      (WHEN (AND (NOT (NULL (DLAMBDA-REST DLAMBDA)))
                (NOT (EQ :IGNORED (DLAMBDA-REST DLAMBDA))))
        (SETF (DLAMBDA-REST DLAMBDA)
              (INSTALL-LOCAL (DLAMBDA-REST DLAMBDA)
```

```

      CLOSED-OVER)))
  (IL:|for| KEY-VAR IL:|in| (DLAMBDA-KEY DLAMBDA) IL:|do| (SETF (SECOND KEY-VAR)
    (INSTALL-LOCAL (SECOND KEY-VAR)
      CLOSED-OVER)))
    (DIGEST-CODE (THIRD KEY-VAR))
    (SETF (FOURTH KEY-VAR)
      (INSTALL-LOCAL (FOURTH KEY-VAR)
        CLOSED-OVER))))

```

;; Then intern any stragglers on the closed-over list.

```

  (IL:|for| VAR IL:|in| CLOSED-OVER IL:|do| (WHEN (AND (CONSP VAR)
    (NULL (GETHASH (THIRD VAR)
      *DVAR-ENV*)))
    (SETF (GETHASH (THIRD VAR)
      *DVAR-ENV*)
      (MAKE-DVAR :KIND :CLOSED :SLOT *HUNK-SIZE* :LEVEL *LEVEL*
        :NAME (SECOND VAR)))
    (INCF *HUNK-SIZE*)))

```

;; And, at long last, digest the body of the function.

```

  (DIGEST-CODE (DLAMBDA-BODY DLAMBDA))

```

;; Now that we have lexical levels for all of the variables in this lambda, we can figure out the proper lexical levels for all of its subfunctions.

```

  (IL:|for| FN-PAIR IL:|in| (DLAMBDA-LOCAL-FUNCTIONS DLAMBDA) IL:|do| (SETF (DVAR-LEVEL (CAR FN-PAIR))
    (DETERMINE-LOCAL-FN-LEXICAL-LEVEL
      (CAR FN-PAIR)
      NIL)))

```

;; Record the results of this digestion (yecch...).

```

  (STORE-DIGEST-INFO)))

```

```

(DEFUN DETERMINE-LOCAL-FN-LEXICAL-LEVEL (FN PENDING)

```

;; Determines the minimum lexical level of the given local functions. The level is the maximum of the levels of its non-local variable references and
 ;; the levels of any functions that it references. PENDING is a list of function variables for which the level is currently being determined. If you
 ;; reference any of these, ignore them for now.

```

  (ASSERT (EQ (DVAR-KIND FN)
    :FUNCTION)
    ' (FN)
    "BUG: Trying to determine minimum level of a non-function.")
  (LET ((LEVEL 0)
    (DLAMBDA (SECOND (ASSOC FN *LOCAL-FN-FIXUPS*)))
    (IL:FOR ID IL:IN (DLAMBDA-NON-LOCAL DLAMBDA)
      IL:DO (LET ((DVAR (GETHASH ID *DVAR-ENV*)))
        (MAXF LEVEL (OR (DVAR-LEVEL DVAR)
          (IF (OR (EQ DVAR FN)
            (MEMBER DVAR PENDING :TEST 'EQ))
            0)
          :ignore it.
          (DETERMINE-LOCAL-FN-LEXICAL-LEVEL DVAR (CONS FN PENDING))))))
    LEVEL))

```

```

(DEFUN DIGEST-CODE (LAP-CODE)

```

```

  (IL:FOR TAIL IL:ON LAP-CODE
    IL:DO (LET ((INST (CAR TAIL)))
      (CASE (FIRST INST)
        ((:VAR :VAR_) (SETF (SECOND INST)
          (INTERN-VAR (SECOND INST))))
        ((:CALL) (WHEN (CONSP (SECOND INST))
          (CASE (FIRST (SECOND INST))
            ((:LAMBDA) (DIGEST-CODE (LIST (SECOND INST))))
            ((:L :F :S :FN) (SETF (SECOND INST)
              (INTERN-VAR (SECOND INST))))))
          ((:LAMBDA) (LET ((DLAMBDA (DLAMBDA-FROM-LAMBDA INST))
            (LEVEL 0))
            (IL:|for| ID IL:|in| (DLAMBDA-NON-LOCAL DLAMBDA)
              IL:|do| (LET ((DVAR (GETHASH ID *DVAR-ENV*)))
                (MAXF LEVEL (OR (DVAR-LEVEL DVAR)
                  (DETERMINE-LOCAL-FN-LEXICAL-LEVEL DVAR NIL)))
              )))
            (SETF (CDR INST)
              (LIST DLAMBDA LEVEL))))
        ((:BIND)
          ;; (if (and (consp var) (null (gethash (third var) *dvar-env*))) (install-local var nil) (error "Variable in :BIND appeared
          ;; earlier: ~S" var))
          (FLET ((INSTALL-NEW (VAR)
            (IF (AND (CONSP VAR)
              (NULL (GETHASH (THIRD VAR)
                *DVAR-ENV*)))
              (INSTALL-LOCAL VAR NIL)
              (GETHASH (THIRD VAR)
                *DVAR-ENV*)))
            (SETF (SECOND INST)
              (MAPCAR #'INSTALL-NEW (SECOND INST)))
            (SETF (THIRD INST)

```

```

      (MAPCAR #'INSTALL-NEW (THIRD INST))))))
    ((:UNBIND :DUNBIND) (SETF (SECOND INST)
      (LENGTH (SECOND INST))
      (THIRD INST)
      (LENGTH (THIRD INST)))))
    ((:TAG)
      (SETF (SECOND INST)
        (INTERN-TAG (SECOND INST)))
      (SETF (DTAG-PTR (SECOND INST))
        TAIL)
      (SETF (DTAG-LEVEL (SECOND INST))
        *LEVEL*))
    ((:JUMP :FJUMP :TJUMP :NFJUMP :NTJUMP :PUSH-TAG) (SETF (SECOND INST)
      (INTERN-TAG (SECOND INST)))))
    ((:CLOSE) (LET ((*LEVEL* (1+ *LEVEL*))
      (*HUNK-SIZE* (IF (> *LEVEL* 0)
        1
        0))))
      ;; In setting up the new lexical environment, don't forget to leave a slot for linking the hunks together, if
      ;; necessary.
      (LET ((DVAR (WITH-COLLECTION (DOLIST (VAR (SECOND INST))
        (COLLECT (SETF (GETHASH (THIRD VAR)
          *DVAR-ENV*)
          (MAKE-DVAR :KIND :CLOSED
            :SLOT *HUNK-SIZE*
            :LEVEL *LEVEL* :NAME
            (SECOND VAR))))))
        (INCF *HUNK-SIZE*))))
        (SETF (REST INST)
          (LIST* DVAR *PVAR-COUNT* (CDDR INST))))
        (INCF *PVAR-COUNT*) ; Allocate a slot for the new hunk.
        (DIGEST-CODE (CDDDR INST))))))

```

(DEFUN STORE-DIGEST-INFO ())

```

  (LET* ((FREE-VAR-ALIST (SORT *FREE-VARS* #'< :KEY #'CDR))
    (FREE-VAR-COUNT (LIST-LENGTH FREE-VAR-ALIST))
    (BOUND-SPECIAL-COUNT (LIST-LENGTH *BOUND-SPECIALS*)))

```

;; First we check for there being too many variables. If we can get it down below the limit by eliminating FVAR's, we'll do that. The
 ;; technique is to replace references with calls to SYMBOL-VALUE and SETQs by calls to SET. This is gross, but there's no easier way out.
 ;; If there would still be too many after eliminating all of the FVARs, we have to punt.

```

  (WHEN (> *PVAR-COUNT* +MAX-ALLOWABLE-PVAR-COUNT+)
    (COMPILER:ASSEMBLER-ERROR "Too many bound variables: ~D. Limit is ~D." *PVAR-COUNT*
      +MAX-ALLOWABLE-PVAR-COUNT+))
  (WHEN (> BOUND-SPECIAL-COUNT +MAX-ALLOWABLE-SPECIAL-COUNT+)
    (COMPILER:ASSEMBLER-ERROR "Too many bound special variables: ~D. Limit is ~D."
      BOUND-SPECIAL-COUNT +MAX-ALLOWABLE-SPECIAL-COUNT+))
  (DOTIMES (I (MAX (- (+ BOUND-SPECIAL-COUNT FREE-VAR-COUNT)
    +MAX-ALLOWABLE-SPECIAL-COUNT+)
    (- (+ *PVAR-COUNT* FREE-VAR-COUNT)
    +MAX-ALLOWABLE-PVAR-COUNT+))
    0))
    (SETF (DVAR-SLOT (CAR (POP FREE-VAR-ALIST)))
      +SLOW-FVAR-SLOT+)
    (DECF FREE-VAR-COUNT))

```

;; This first part gets the entries on the name-table in the right order by building the table backwards. The final order is PVARs, IVARs,
 ;; FVARs with the PVARs and IVARs in reverse order. This lets the free variable lookup find the correct variable first.

```

  (IL:FOR PAIR IL:IN FREE-VAR-ALIST IL:AS SLOT IL:FROM (1- (+ FREE-VAR-COUNT *PVAR-COUNT*)) IL:BY -1
    IL:DO (PUSH (LIST +FVAR-CODE+ SLOT (DVAR-NAME (CAR PAIR)))
      (DCODE-NAME-TABLE *DCODE*))
    (SETF (DVAR-SLOT (CAR PAIR))
      SLOT)

```

; While we're at this, assign slots to the free variables.

```

  )
  (IL:|for| DVAR IL:|in| (NREVERSE *BOUND-SPECIALS*) IL:|do| (PUSH (LIST (ECASE (DVAR-KIND DVAR)
    ((:LOCAL) +PVAR-CODE+)
    ((:ARGUMENT) +IVAR-CODE+))
    (DVAR-SLOT DVAR)
    (DVAR-NAME DVAR))
    (DCODE-NAME-TABLE *DCODE*)))

```

;; Now to fill in some of the more mundane fields of the DCODE.

```

  (SETF (DCODE-NLOCALS *DCODE*)
    *PVAR-COUNT*)
  (SETF (DCODE-NFREEVARS *DCODE*)
    FREE-VAR-COUNT))

```

(DEFUN DVAR-FROM-LAP-VAR (LAP-VAR)

```

  (OR (GETHASH (LAP-VAR-ID LAP-VAR)
    *DVAR-ENV*)
    (ERROR "The LAP var ~S should have been interned by now." LAP-VAR)))

```

```

(DEFININE LAP-VAR-ID (VAR)
  (IF (CONSP VAR)
    (THIRD VAR)
    VAR))

(DEFUN INSTALL-LOCAL (VAR CLOSED-OVER)
  (AND VAR (LET ((DVAR (INSTALL-VAR VAR CLOSED-OVER :LOCAL *PVAR-COUNT*)))
    (WHEN (EQ :LOCAL (DVAR-KIND DVAR))
      (INCF *PVAR-COUNT*))
    DVAR)))

(DEFUN INSTALL-VAR (VAR CLOSED-OVER KIND SLOT)
  (AND VAR (DESTRUCTURING-BIND (SCOPE NAME ID)
    VAR
    (COND
      ((MEMBER ID CLOSED-OVER :KEY #'LAP-VAR-ID)
        (PROG1 (SETF (GETHASH ID *DVAR-ENV*)
          (MAKE-DVAR :KIND :CLOSED :SLOT *HUNK-SIZE* :LEVEL *LEVEL* :NAME NAME))
          (INCF *HUNK-SIZE*)))
        (T (LET ((DVAR (SETF (GETHASH ID *DVAR-ENV*)
          (MAKE-DVAR :KIND (IF (EQ SCOPE ' :F)
            :FREE
            KIND)
            :SLOT SLOT :NAME NAME))))
          (CASE SCOPE
            ((:S) (PUSH DVAR *BOUND-SPECIALS*))
            ((:F) (PUSH (CONS DVAR 1)
              *FREE-VARS*)))
          DVAR))))))

(DEFUN INTERN-VAR (VAR)
  (IF (CONSP VAR)
    (IF (EQ (FIRST VAR)
      ' :G) ; Global vars don't get interned.
      VAR
      (LET ((DVAR (GETHASH (THIRD VAR)
        *DVAR-ENV*)))
        (COND
          ((NOT (NULL DVAR))
            (WHEN (EQ :FREE (DVAR-KIND DVAR))
              (INCF (CDR (ASSOC DVAR *FREE-VARS*))))
            DVAR)
          (T (INSTALL-LOCAL VAR NIL))))))
    (OR (GETHASH VAR *DVAR-ENV*)
      (ERROR "Unknown LAP variable ID: ~S" VAR)))

(DEFUN INTERN-TAG (ID)
  (OR (GETHASH ID *DTAG-ENV*)
    (SETF (GETHASH ID *DTAG-ENV*)
      (MAKE-DTAG))))

;; Function entry code

(DEFUN EASY-ENTRY-P (DLAMBDA)
  (AND (OR (NULL (DLAMBDA-REST DLAMBDA))
    (EQ :IGNORED (DLAMBDA-REST DLAMBDA)))
    (NULL (DLAMBDA-KEY DLAMBDA))
    (IL:|for| OPT-VAR IL:|in| (DLAMBDA-OPTIONAL DLAMBDA) IL:|always| (AND (EQUAL ' (:CONST NIL))
      (SECOND OPT-VAR))
      (NULL (THIRD OPT-VAR))))))

(DEFUN GENERATE-EASY-ENTRY (DLAMBDA)
  ;; Emit code to create the hunk for this level and leave it on top of stack. We'll use it in the processing of the arguments.
  (WHEN (NOT (NULL (DLAMBDA-CLOSED-OVER DLAMBDA)))
    (CREATE-HUNK *HUNK-SIZE* (CDAR *HUNK-MAP*)
      (AND (> *LEVEL* 1)
        0)
      NIL))
  ;; The required and optional parameters are treated alike in the easy entry. If any of them are closed over, copy them into the hunk.
  (IL:|for| DVAR IL:|in| (APPEND (DLAMBDA-REQUIRED DLAMBDA)
    (MAPCAR #'FIRST (DLAMBDA-OPTIONAL DLAMBDA))))
  IL:|as| IVAR-COUNT IL:|from| 0 IL:|do| (WHEN (EQ :CLOSED (DVAR-KIND DVAR))
    (EMIT-BYTE-LIST ` (,@ (CHOOSE-OP ' (IL:IVAR . IL:IVARX)
      IVAR-COUNT)
      IL:RPLPTR.N
      , (IL:LLSH (DVAR-SLOT DVAR)
        1))))))
  (WHEN (NOT (NULL (DLAMBDA-CLOSED-OVER DLAMBDA)))
    (EMIT-BYTE ' IL:POP)) ; Get rid of the hunk on the top of stack.

```

;; Set up the ARG-TYPE and NUM-ARGS information. All Interlisp functions pass thru the easy-entry code, so this will catch them.

```
(LET ((ARG-TYPE (OR (DLAMBDA-ARG-TYPE DLAMBDA)
                    +LAMBDA-SPREAD+)))
      (SETF (DCODE-ARG-TYPE *DCODE*)
            ARG-TYPE)
      (SETF (DCODE-NUM-ARGS *DCODE*)
            (COND
              ((OR (= ARG-TYPE +LAMBDA-SPREAD+)
                   (= ARG-TYPE +NLAMBDA-SPREAD+))
               (+ (LENGTH (DLAMBDA-REQUIRED DLAMBDA))
                  (LENGTH (DLAMBDA-OPTIONAL DLAMBDA))))
              ((= ARG-TYPE +LAMBDA-NO-SPREAD+)
               -1)
              ((= ARG-TYPE +NLAMBDA-NO-SPREAD+)
               1))))))
```

```
(DEFUN GENERATE-HARD-ENTRY (DLAMBDA)
  (LET ((NUM-REQUIRED (LENGTH (DLAMBDA-REQUIRED DLAMBDA)))
        (NUM-OPTIONAL (LENGTH (DLAMBDA-OPTIONAL DLAMBDA))))
    ;; Emit code to create the hunk for this level and store it away.
    (WHEN (NOT (NULL (DLAMBDA-CLOSED-OVER DLAMBDA)))
      (CREATE-HUNK *HUNK-SIZE* (CDAR *HUNK-MAP*)
                   (AND (> *LEVEL* 1)
                        0)
                   T))
    ;; Generate a check for a bad number of arguments, unless there are no illegal values.
    (UNLESS (AND (ZEROP NUM-REQUIRED)
                  (OR (AND (DLAMBDA-REST DLAMBDA)
                           (NOT (EQ :IGNORED (DLAMBDA-REST DLAMBDA))))
                      (DLAMBDA-KEY DLAMBDA))))
      (GENERATE-ARG-CHECK DLAMBDA))
    ;; Copy the closed required args to the hunk.
    (IL:for| DVAR IL:|in| (DLAMBDA-REQUIRED DLAMBDA) IL:|as| IVAR-COUNT IL:|from| 0
      IL:|do| (WHEN (EQ :CLOSED (DVAR-KIND DVAR))
        (EMIT-BYTE-LIST `(@ (CHOOSE-OP ' (IL:PVAR . IL:PVARX)
                                         (CDAR *HUNK-MAP*))
                             ,@ (CHOOSE-OP ' (IL:IVAR . IL:IVARX)
                                             IVAR-COUNT)
                             IL:RPLPTR.N
                             , (IL:LLSH (DVAR-SLOT DVAR)
                                           1)
                             IL:POP))))))
    ;; Generate code for the optional and rest arguments.
    (GENERATE-OPT-AND-REST DLAMBDA NUM-REQUIRED NUM-OPTIONAL)
    ;; Generate code for the keyword arguments.
    (GENERATE-KEY DLAMBDA NUM-REQUIRED NUM-OPTIONAL)
    ;; Fill in some information in the DCODE structure.
    (SETF (DCODE-ARG-TYPE *DCODE*)
          +LAMBDA-NO-SPREAD+)
    (SETF (DCODE-NUM-ARGS *DCODE*)
          -1)))
```

```
(DEFUN GENERATE-ARG-CHECK (DLAMBDA)
```

;;; Generate code that signals an error if too few or too many arguments are given.

```
(LET* ((MIN-ARGS (LENGTH (DLAMBDA-REQUIRED DLAMBDA)))
       (MAX-ARGS (AND (NULL (DLAMBDA-REST DLAMBDA))
                       (NULL (DLAMBDA-KEY DLAMBDA))
                       (NULL (DLAMBDA-ALLOW-OTHER-KEYS DLAMBDA))
                       (+ MIN-ARGS (LENGTH (DLAMBDA-OPTIONAL DLAMBDA)))))
      (OK-TAG (MAKE-DTAG))
      (BAD-TAG (MAKE-DTAG)))
  (IF (NULL MAX-ARGS)
      (EMIT-BYTE-LIST ` (IL:MYARGCOUNT ,@ (PUSH-INTEGER (1- MIN-ARGS))
                    IL:GREATERP
                    (:TJUMP ,OK-TAG)
                    ,@ (PUSH-INTEGER MIN-ARGS)
                    IL:FN2 (:FN SI::ARGUMENT-ERROR)
                    IL:POP
                    (:TAG ,OK-TAG)))
      (EMIT-BYTE-LIST ` (IL:MYARGCOUNT ,@ (PUSH-INTEGER (1- MIN-ARGS))
                    IL:GREATERP
                    (:FJUMP ,BAD-TAG)
                    IL:MYARGCOUNT
                    ,@ (PUSH-INTEGER MAX-ARGS)
                    IL:GREATERP
                    (:FJUMP ,OK-TAG))
```

```
(DEFUN GENERATE-KEY (DLAMBDA NUM-REQUIRED NUM-OPTIONAL)
  "Generate code to check for and default the keyword arguments."
  (LET ((START (+ 1 NUM-REQUIRED NUM-OPTIONAL)))
    (IL:FOR TAIL IL:ON (DLAMBDA-KEY DLAMBDA)
      IL:DO (DESTRUCTURING-BIND (KEYWORD VAR CODE SVAR)
        (CAR TAIL)
        (LET ((FOUND-TAG (MAKE-DTAG))
              (NEXT-TAG (MAKE-DTAG)))
          (EMIT-BYTE-LIST `(IL:ACONST (:SYM ,KEYWORD)
                                   IL:FINDKEY
                                   ,START
                                   (:NTJUMP ,FOUND-TAG)))
          ;; Not there; compute the init-form.
          (STACK-ANALYZE CODE 1)
          (ASSEMBLE CODE)
          (EMIT-BYTE-LIST ` (, @ (STORE-VAR VAR T)
                               , @ (AND SVAR `(IL:\'NIL , @ (STORE-VAR SVAR T)))
                               (:JUMP ,NEXT-TAG)
                               (:TAG ,FOUND-TAG)
                               IL:ARG0
                               , @ (STORE-VAR VAR T)
                               , @ (AND SVAR `(IL:\'T , @ (STORE-VAR SVAR T)))
                               (:TAG ,NEXT-TAG)))))))

(DEFUN GENERATE-OPT-AND-REST (DLAMBDA NUM-REQUIRED NUM-OPTIONAL)
  (LET ((OPT-INIT-VALUES NIL)
        (AFTER-OPTS-TAG (MAKE-DTAG)))
    ;; OPT-INIT-VALUES will hold a list of lists (var svar tag . lap-code), one for each opt-var. These will be generated in order after we take care
    ;; of the rest argument.
    (UNLESS (ZEROP NUM-OPTIONAL)
      ;; Convert the arg-count into a count of remaining arguments.
      (EMIT-BYTE-LIST `(IL:MYARGCOUNT , @ (AND (NOT (ZEROP NUM-REQUIRED))
                                                ` (, @ (PUSH-INTEGER NUM-REQUIRED)
                                                  IL:IDIFFERENCE))))
      ;; Generate the code for testing for each optional variable. If it's there, copy it to its slot and set the svar, if one exists. Otherwise, jump
      ;; into the middle of the stream of init-form computations.
      (IL:|for| TAIL IL:|on| (DLAMBDA-OPTIONAL DLAMBDA) IL:|as| IVAR-COUNT IL:|from| (1+ NUM-REQUIRED)
        IL:|do| (LET ((TAG (MAKE-DTAG)))
          (DESTRUCTURING-BIND (VAR CODE SVAR)
            (CAR TAIL)
            (EMIT-BYTE-LIST `(, @ (AND (CDR TAIL)
                                       ' (IL:COPY))
                               IL:\'0 EQ (:TJUMP ,TAG)
                               , @ (PUSH-INTEGER IVAR-COUNT)
                               IL:ARG0
                               , @ (STORE-VAR VAR T)
                               , @ (AND SVAR `(IL:\'T , @ (STORE-VAR SVAR T)))
                               , @ (AND (CDR TAIL)
                                       ' (IL:\'1 IL:IDIFFERENCE))))
            (PUSH (LIST* VAR SVAR TAG CODE)
              OPT-INIT-VALUES))))))
    ;; All of the &optionals were provided. Handle the &rest argument.
    (WHEN (AND (DLAMBDA-REST DLAMBDA)
              (NOT (EQ :IGNORED (DLAMBDA-REST DLAMBDA))))
      (EMIT-BYTE-LIST `(IL:\'NIL IL:MYARGCOUNT IL:RESTARTLIST , (+ 1 NUM-REQUIRED NUM-OPTIONAL)
                        , @ (STORE-VAR (DLAMBDA-REST DLAMBDA)
                                      T))))
    ;; Compute the default values for the various optional parameters one after another. The testing code above jumps into the middle of this.
    (UNLESS (ZEROP NUM-OPTIONAL)
      (EMIT-BYTE ` (:JUMP ,AFTER-OPTS-TAG)) ; If we fall into this code, all of the optionals were provided, so
      ; jump around the default-value calculations.
      (IL:FOR VARS-TAG-CODE IL:IN (NREVERSE OPT-INIT-VALUES)
        IL:DO (EMIT-BYTE ` (:TAG , (CADDR VARS-TAG-CODE)))
        (STACK-ANALYZE (CDDDR VARS-TAG-CODE)
          1)
        (ASSEMBLE (CDDDR VARS-TAG-CODE))
        (EMIT-BYTE-LIST (STORE-VAR (CAR VARS-TAG-CODE)
          T))
        (WHEN (CADR VARS-TAG-CODE) ; There's an svar
          (EMIT-BYTE-LIST `(IL:\'NIL , @ (STORE-VAR (CADR VARS-TAG-CODE)
            T))))))
```

```

(WHEN (AND (DLAMBDA-REST DLAMBDA)
            (NOT (EQ :IGNORED (DLAMBDA-REST DLAMBDA)))))
; If not all of the optionals were there, then there can't be any
; &rest arguments.
(EMIT-BYTE-LIST `(IL:\'NIL ,@(STORE-VAR (DLAMBDA-REST DLAMBDA)
                                         T))))
(EMIT-BYTE `(:TAG ,AFTER-OPTS-TAG))))))

```

:: Stack analysis

```

(DEFVAR *ENDING-DEPTH* NIL
  "An Alist mapping lexical level to the stack depth at exit from that level.")

```

```

(DEFUN GATHER-TAGS (CODE)
  (LET ((TAGS-FOUND NIL))
    (DOLIST (INST CODE)
      (CASE (FIRST INST)
        ((:TAG) (PUSH (SECOND INST)
                      TAGS-FOUND))
        ((:CLOSE) (SETQ TAGS-FOUND (NCONC (GATHER-TAGS (CDDDR INST))
                                           TAGS-FOUND))))))
  TAGS-FOUND))

```

```

(DEFUN GATHER-ROOTS (CODE)

```

;;; Return the minimum set of tags in CODE such that starting stack analysis at the beginning of CODE and at each of these tags will result in all of
 ;;; CODE being reached.

;;; We work by using the same algorithm for reaching code as in STACK-ANALYZE, marking each reachable tag. We start the search first at the
 ;;; beginning of CODE and then at each so-far unmarked tag.

```

(LET ((TAG-LIST (GATHER-TAGS CODE)))
  (REACH-TAGS CODE)
  (DOLIST (TAG TAG-LIST)
    (WHEN (NOT (DTAG-REACHABLE? TAG))
      (REACH-TAGS (CDR (DTAG-PTR TAG)))))
  (REMOVE-IF 'DTAG-REACHABLE? TAG-LIST)))

```

```

(DEFUN REACH-TAGS (CODE)

```

;;; Mark all reachable tags as being so.

```

(DOLIST (INST CODE)
  (ECASE (CAR INST)
    ((:TAG) (COND
              ((DTAG-REACHABLE? (SECOND INST))
               (RETURN))
              (T (SETF (DTAG-REACHABLE? (SECOND INST))
                       T)))))
    ((:FJUMP :NFJUMP :TJUMP :NTJUMP) (REACH-TAGS (DTAG-PTR (SECOND INST)))))
    ((:JUMP)
     (REACH-TAGS (DTAG-PTR (SECOND INST)))
     (RETURN))
    ((:RETURN) (RETURN))
    ((:CLOSE) (REACH-TAGS (CDDDR INST)))
    ((:VAR :VAR_ :CONST :POP :LAMBDA :COPY :PUSH-TAG :NOTE-STACK :SET-STACK :DSET-STACK :SWAP :BIND
          :UNBIND :DUNBIND :CALL :STKCALL) NIL))))

```

```

(DEFUN STACK-ANALYZE (CODE &OPTIONAL (EXPECTED-ENDING-DEPTH 0))

```

;;; Walk the given code annotating the tags in it with information about the stack and binding depth of control at that point.

```

(LET ((*LEVEL* *LEVEL*)
      (*ENDING-DEPTH* (LIST (CONS *LEVEL* NIL)))
      (ROOT-LIST (GATHER-ROOTS CODE)))
  (STACK-ANALYZE-CODE CODE 0 NIL)
  (DOLIST (TAG ROOT-LIST)
    ;; JDS 18-NOV-91 Added *Edning-DEPTH* binding here, because not all tags will be at level 1! Fixes AR 11428.
    (WHEN (NULL (DTAG-STACK-DEPTH TAG))
      (LET* ((*LEVEL* (DTAG-LEVEL TAG))
             (*ENDING-DEPTH* (LIST (CONS *LEVEL* NIL))))
        (STACK-ANALYZE-CODE (DTAG-PTR TAG)
                             NIL NIL))))
    (LET ((ENDING-DEPTH (CDR (FIRST *ENDING-DEPTH*))))
      (ASSERT (OR (NULL ENDING-DEPTH)
                  (= ENDING-DEPTH EXPECTED-ENDING-DEPTH))
              NIL "Code doesn't leave stack empty!"))))

```

```

(DEFUN STACK-ANALYZE-CODE (CODE INIT-DEPTH INIT-BINDING-DEPTH)

```

;;; Annotate the tags in CODE with the stack and binding depth at those points in execution, assuming that the stack depth is INIT-DEPTH and the
 ;;; binding depth is as in INIT-BINDING-DEPTH on entry to the code..

```
(FLET ((STACK-AMBIGUOUS-ERROR (OPCODE)
  (ERROR "BUG: The LAP opcode ~S should not appear in stack-ambiguous territory." OPCODE)))
  (MACROLET ((CHECK-STACK NIL `(WHEN (NULL DEPTH)
    (STACK-AMBIGUOUS-ERROR (FIRST INST)))))
    (LET ((DEPTH INIT-DEPTH)
      (BINDING-DEPTH INIT-BINDING-DEPTH))
      (DOLIST (INST CODE)
        (ECASE (CAR INST)
          ((:TAG) (LET ((THE-TAG (SECOND INST)))
            (COND
              ((NULL (DTAG-STACK-DEPTH THE-TAG))
               (SETF (DTAG-STACK-DEPTH THE-TAG)
                     (CONS BINDING-DEPTH DEPTH)))
              (ASSERT (>= *LEVEL* (DTAG-LEVEL THE-TAG))
                NIL "COMPILER BUG: Jump INTO a lexical contour."))
              (T (ASSERT (AND (EQUAL BINDING-DEPTH (CAR (DTAG-STACK-DEPTH THE-TAG)))
                (EQUAL DEPTH (CDR (DTAG-STACK-DEPTH THE-TAG))))
                NIL "BUG: Inconsistent stack depths seen at the target of
                several branches."))
                (RETURN-FROM STACK-ANALYZE-CODE (STACK-ANALYZE-CODE (DTAG-PTR
                  (SECOND INST))
                    DEPTH BINDING-DEPTH))))
          ((:JUMP) (RETURN-FROM STACK-ANALYZE-CODE (STACK-ANALYZE-CODE (DTAG-PTR
            (SECOND INST))
              DEPTH BINDING-DEPTH)))
          ((:FJUMP :TJUMP)
            (CHECK-STACK)
            (DECF DEPTH)
            (STACK-ANALYZE-CODE (DTAG-PTR (SECOND INST))
              DEPTH BINDING-DEPTH))
          ((:NFJUMP :NTJUMP)
            (CHECK-STACK)
            (STACK-ANALYZE-CODE (DTAG-PTR (SECOND INST))
              DEPTH BINDING-DEPTH)
            (DECF DEPTH))
          ((:VAR :COPY :CONST :LAMBDA :PUSH-TAG)
            (CHECK-STACK)
            (INCF DEPTH))
          ((:VAR_ :SWAP)
            (CHECK-STACK)
            ; Net stack effect is zero.
            )
          ((:POP)
            (CHECK-STACK)
            (DECF DEPTH))
          ((:NOTE-STACK)
            (CHECK-STACK)
            (SETF (GETHASH (SECOND INST)
              *STACK-ENV*)
              (LIST DEPTH BINDING-DEPTH)))
          ((:SET-STACK :DSET-STACK) (LET ((LOOKUP (GETHASH (SECOND INST)
            *STACK-ENV*)))
            (ASSERT (NOT (NULL LOOKUP))
              NIL ":NOTE-STACK not seen before :SET-STACK")
            (ASSERT (OR (NULL DEPTH)
              ; We don't know where we are, or
              (AND (>= DEPTH (FIRST LOOKUP))
                (OR (NULL (SECOND LOOKUP))
                  (TAILP (SECOND LOOKUP)
                    BINDING-DEPTH))))
              ; We can, indeed, feasibly set the stack to the given place.
              )
            )
            NIL "Attempt to :SET-STACK to unreachable
            state.")
            (DESTRUCTURING-SETQ (DEPTH BINDING-DEPTH)
              LOOKUP)
            (IF (EQ (FIRST INST)
              :SET-STACK)
              (INCF DEPTH))))
          ((:BIND)
            (CHECK-STACK)
            ;; This takes into account the popping of some number of values into the variables and then the pushing of
            ;; the binding mark(s).
            (DECF DEPTH (LENGTH (SECOND INST)))
            (PUSH (CONS (FOURTH INST)
              DEPTH)
              BINDING-DEPTH)
            (INCF DEPTH (MAX 1 (FLOOR (+ (LENGTH (THIRD INST))
              14)
              15))))
          ((:UNBIND :DUNBIND)
            (CHECK-STACK)
            (UNLESS (EQL (FOURTH INST)
```



```

(CAR (FIRST BINDING-DEPTH)))
(ERROR "ASSEMBLER BUG: Mismatched :BIND and :UNBIND.")
(SETQ DEPTH (CDR (POP BINDING-DEPTH)))
(WHEN (EQ 'UNBIND (FIRST INST))
  (INCF DEPTH))
((:CALL) (IF (NULL DEPTH)
  (UNLESS (EQ (SECOND INST)
    'IL:\\MVLIST)
    (STACK-AMBIGUOUS-ERROR (FIRST INST)))
  (DECF DEPTH (1- (THIRD INST)))))
((:STKCALL)
  (CHECK-STACK)
  (DECF DEPTH (1+ (SECOND INST))))
((:RETURN) (RETURN-FROM STACK-ANALYZE-CODE))
((:CLOSE)
  (CHECK-STACK)
  (LET* ((*LEVEL* (1+ *LEVEL*))
    (*ENDING-DEPTH* (CONS (CONS *LEVEL* NIL)
      *ENDING-DEPTH*)))
    (STACK-ANALYZE-CODE (CDDDR INST)
      DEPTH BINDING-DEPTH)
    (SETQ DEPTH (CDR (FIRST *ENDING-DEPTH*)))
    (WHEN (NULL DEPTH)
      (RETURN-FROM STACK-ANALYZE-CODE))))))
(LET ((LOOKUP (ASSOC *LEVEL* *ENDING-DEPTH*)))
  ;; If this assertion fails, it means that we've twice analyzed a piece of code -- and run off at the end of it without
  ;; returning to the caller. Normally, this is used to compute the ending depth of a closed-over part of code.
  (ASSERT (NOT (CDR LOOKUP))
    NIL "Ran off end twice")
  (WHEN (NULL (CDR LOOKUP))
    (SETF (CDR LOOKUP)
      DEPTH))))))

```

;; The guts of assembly

```
(DEFUN ASSEMBLE (LAP-CODE)
```

;;; Translate LAP code into D-machine bytecodes.

```
(ASSEMBLE-CODE LAP-CODE 0 NIL)
```

```
(DEFUN ASSEMBLE-CODE (LAP-CODE DEPTH BINDING-DEPTH)
```

;;; Translate LAP code into D-machine bytecodes.

```

(DO ((TAIL LAP-CODE (CDR TAIL))
  INST)
  ((ENDP TAIL))
  (SETQ INST (FIRST TAIL))
  (MACROLET
    ((INCR (VAR &OPTIONAL (DELTA 1))
      `(AND ,VAR (SETQ ,VAR (+ ,VAR ,DELTA))))
    (DECR (VAR &OPTIONAL (DELTA 1))
      `(AND ,VAR (SETQ ,VAR (- ,VAR ,DELTA))))
    (ECASE (CAR INST)
      ((:VAR)
        (EMIT-BYTE-LIST (REF-VAR (SECOND INST)))
        (INCR DEPTH)
        (ASSERT (OR (NOT DEPTH)
          (>= DEPTH 0))
          NIL "Depth went negative in ~S." :VAR))
      ((:VAR_) (EMIT-BYTE-LIST (STORE-VAR (SECOND INST)
        (COND
          ((EQ 'POP (FIRST (SECOND TAIL)))
            (SETQ TAIL (CDR TAIL))
            (DECR DEPTH)
            (ASSERT (OR (NOT DEPTH)
              (>= DEPTH 0))
              NIL "Depth went negative in ~S." :VAR_)
            T)
          (T NIL))))))
      ((:COPY)
        (EMIT-BYTE 'IL:COPY)
        (INCR DEPTH)
        (ASSERT (OR (NOT DEPTH)
          (>= DEPTH 0))
          NIL "Depth went negative in ~S." :COPY))
      ((:SWAP) (EMIT-BYTE 'IL:SWAP))
      ((:CONST)
        (LET* ((VALUE (SECOND INST))
          (LOOKUP (ASSOC VALUE +CONSTANT-OPCODES+)))
          (COND
            ((NOT (NULL LOOKUP))
              (EMIT-BYTE (CDR LOOKUP)))

```

```

((SYMBOLP VALUE)
 (EMIT-BYTE-LIST `(IL:ACONST (:SYM ,VALUE))))
((INTEGERP VALUE)
 (EMIT-BYTE-LIST (PUSH-INTEGER VALUE)))
(T (EMIT-BYTE-LIST `(IL:GCONST (:LIT ,VALUE)))))
(INCR DEPTH)
(ASSERT (OR (NOT DEPTH)
              (>= DEPTH 0))
  NIL "Depth went negative in ~S." :CONST))
((:LAMBDA) (LET ((DLAMBDA (SECOND INST))
                  (LAMBDA-LEVEL (THIRD INST)))
              (COND
                ((AND NIL (ZEROP LAMBDA-LEVEL)) ; We used to do something different for lambdas with empty
                ; environments.
                 (EMIT-BYTE-LIST `(IL:GCONST (:LAMBDA 0 ,DLAMBDA)))))
                (T ;; This will need to be a closure. Find our best hunk for it and construct a closure object around it and the
                ;; lambda.
                 (EMIT-BYTE-LIST `(IL:SICK (:TYPE IL:COMPILED-CLOSURE)
                                           IL:CREATECELL IL:GCONST (:LAMBDA ,LAMBDA-LEVEL
                                                                    ,DLAMBDA)
                                           IL:RPLPTR.N 0 ,@(AND (NOT (ZEROP LAMBDA-LEVEL))
                                                                `(@ (FETCH-HUNK LAMBDA-LEVEL)
                                                                  IL:RPLPTR.N 2))))))
              (INCR DEPTH)
              (ASSERT (OR (NOT DEPTH)
                          (>= DEPTH 0))
                NIL "Depth went negative in ~S." :LAMBDA)))
((:POP)
 (EMIT-BYTE 'IL:POP)
 (DECR DEPTH)
 (ASSERT (OR (NOT DEPTH)
              (>= DEPTH 0))
  NIL "Depth went negative in ~S." :POP))
((:NOTE-STACK) ) ; Now a no-op; used during stack analysis.
((:SET-STACK :DSET-STACK)
 (FLET
  ((EMIT-UNWIND (DESIRED-DEPTH SAVE-TOS?)
    (EMIT-BYTE-LIST `(IL:UNWIND (:UNWIND ,DESIRED-DEPTH)
                                , (IF SAVE-TOS?
                                    1
                                    0))))))
  (LET* ((SAVE-TOS? (EQ (FIRST INST)
                        :SET-STACK))
         (LOOKUP (GETHASH (SECOND INST)
                          *STACK-ENV*))
         (DESIRED-DEPTH (FIRST LOOKUP))
         (DESIRED-BINDING-DEPTH (MAPCAR 'CDR (SECOND LOOKUP))))
    (COND
      ((NULL DEPTH)
       ;; We don't know where we are: use UNWIND.
       (EMIT-UNWIND DESIRED-DEPTH SAVE-TOS?)
       (SETQ DEPTH DESIRED-DEPTH BINDING-DEPTH DESIRED-BINDING-DEPTH))
      ((EQ (FIRST BINDING-DEPTH)
            (FIRST DESIRED-BINDING-DEPTH))
       ;; There are no intervening binds, so we can just pop.
       (WHEN SAVE-TOS?
        (DECF DEPTH)
        (ASSERT (OR (NOT DEPTH)
                    (>= DEPTH 0))
          NIL "Depth went negative in ~S." :SET-STACK))
        (LET ((ADJUSTMENT (- DEPTH DESIRED-DEPTH))
              (IF (MINUSP ADJUSTMENT)
                  (HELP "POP.N stack adjustment negative: " ADJUSTMENT))
              (CASE ADJUSTMENT
                ((0) )
                ((1)
                 (IF SAVE-TOS?
                  (EMIT-BYTE 'IL:SWAP))
                  (EMIT-BYTE 'IL:POP))
                (OTHERWISE (IF SAVE-TOS?
                              (IF (<= ADJUSTMENT 128)
                                ;; STORE.N can only be used for distances less than this limit.
                                (EMIT-BYTE-LIST `(IL:STORE.N ,(* 2 (1- ADJUSTMENT))
                                                         IL:POP.N
                                                         , (1- ADJUSTMENT)))
                              (EMIT-UNWIND DESIRED-DEPTH T))
                              (IF (<= ADJUSTMENT 256)
                                ;; POP.N can only be used for disatnces less than this limit.
                                (EMIT-BYTE-LIST `(IL:POP.N , (1- ADJUSTMENT)))
                                (EMIT-UNWIND DESIRED-DEPTH NIL))))))
          (SETQ DEPTH DESIRED-DEPTH)))

```

```

((AND (EQUAL (REST BINDING-DEPTH)
              DESIRED-BINDING-DEPTH)
      (EQL DESIRED-DEPTH (FIRST DESIRED-BINDING-DEPTH))))
;; There is only one bind mark in the way - use UNBIND
(EMIT-BYTE (IF SAVE-TOS?
              'IL:UNBIND
              'IL:DUNBIND))
(SETQ DEPTH (POP BINDING-DEPTH))
(ASSERT (OR (NOT DEPTH)
             (>= DEPTH 0)))
NIL "Depth went negative in ~S." :|pop-of-binding-stack|)
(T ;; Use UNWIND in all other cases.
  (EMIT-UNWIND DESIRED-DEPTH SAVE-TOS?)
  (SETQ DEPTH DESIRED-DEPTH BINDING-DEPTH DESIRED-BINDING-DEPTH)
  (ASSERT (OR (NOT DEPTH)
               (>= DEPTH 0)))
  NIL "Depth went negative in ~S." :SET-STACK-USING-UNWIND)))
(WHEN SAVE-TOS? (INCF DEPTH))))
((:BIND) (LABELS ((DO-BIND (NUM-VALUES NUM-NILS STARTING-SLOT)
                          (COND
                           ((> NUM-VALUES 15)
                            (COMPILER:ASSEMBLER-ERROR "Too many non-NIL values bound in a single
                                                         :BIND: ~S. Limit is 15." NUM-VALUES))
                           ((> NUM-NILS 15)
                            (DO-BIND NUM-VALUES 15 STARTING-SLOT)
                            (DO-BIND 0 (- NUM-NILS 15)
                                      (+ STARTING-SLOT NUM-VALUES 15)))
                           (T (EMIT-BYTE-LIST '(IL:BIND , (+ (IL:LLSH NUM-NILS 4)
                                                             NUM-VALUES)
                                                , (1- (+ STARTING-SLOT NUM-VALUES NUM-NILS))))
                                (INCR DEPTH))))))
  (LET* ((VALUES (SECOND INST))
         (NUM-VALUES (LENGTH VALUES))
         (NILS (THIRD INST))
         (NUM-NILS (LENGTH NILS)))
    (DECR DEPTH NUM-VALUES)
    (ASSERT (OR (NOT DEPTH)
                 (>= DEPTH 0)))
    NIL "Depth went negative in ~S." :BIND)
    (PUSH DEPTH BINDING-DEPTH)
    (DO-BIND NUM-VALUES NUM-NILS (COND
                                  (VALUES (DVAR-SLOT (CAR VALUES)))
                                  (NILS (DVAR-SLOT (CAR NILS)))
                                  (T 1))))))
  ((:UNBIND :DUNBIND)
   (LET ((BYTE (CASE (FIRST INST)
                     (:UNBIND 'IL:UNBIND)
                     (:DUNBIND 'IL:DUNBIND))))
     (DOTIMES (I (FLOOR (+ (SECOND INST)
                           (THIRD INST)
                           14)
                           15))
       (EMIT-BYTE BYTE)))
   (SETQ DEPTH (POP BINDING-DEPTH))
   (ASSERT (OR (NOT DEPTH)
                (>= DEPTH 0)))
   NIL "Depth went negative in ~S." :UNBIND)
   (IF (EQ (FIRST INST)
           'IL:UNBIND)
       (INCR DEPTH)))
  ((:TAG)
   (EMIT-BYTE '(:TAG , (SECOND INST)))
   (LET ((STACK-DEPTH (DTAG-STACK-DEPTH (SECOND INST))))
     (SETQ DEPTH (CDR STACK-DEPTH))
     (ASSERT (OR (NOT DEPTH)
                  (>= DEPTH 0)))
     NIL "Depth went negative in ~S." :TAG)
     (SETQ BINDING-DEPTH (MAPCAR 'CDR (CAR STACK-DEPTH))))))
  ((:PUSH-TAG)
   (EMIT-BYTE INST)
   (INCR DEPTH))
  ((:JUMP)
   (EMIT-BYTE INST))
  ((:TJUMP :FJUMP :NTJUMP :NFJUMP)
   (EMIT-BYTE INST)
   (DECR DEPTH)
   (ASSERT (OR (NOT DEPTH)
                (>= DEPTH 0)))
   NIL "Depth went negative in ~S." :JUMP))
  ((:CALL) (DESTRUCTURING-BIND
             (FN-TO-CALL NUM-ARGS &KEY ((:NOT-INLINE NOT-INLINE?))
                        ((:SPREAD-LAST SPREAD-LAST?))
             ; SPREAD-LAST? is the hook for APPLY and the interpreter
             ; hacks. Currently ignored. The idea is that you let the assembler
             ; JUMP opcode does NOT pop anything off the stack
             ; Other jump opcodes DO pop (the NT & NF, only if the jump isn't
             ; taken). Since we're looking at stack depth right after this
             ; instruction, this means we can assume the jump didn't
             ; happen....

```

```

; put in the magic loop that spreads the last argument, and takes
; case of allocating the temps for that loop.

)
(REST INST)
(TYPECASE FN-TO-CALL
  (SYMBOL
    (LET ((DOPVAL (GET FN-TO-CALL 'IL:DOPVAL)))
      (BLOCK :CALL-PROCESSING
        (UNLESS (OR NOT-INLINE? (NULL DOPVAL))
          (ASSERT (CONSP DOPVAL)
            ' (FN-TO-CALL DOPVAL)
            "DOPVAL for ~S is not a list: ~S" FN-TO-CALL DOPVAL)
          (IL:FOR ITEM IL:INSIDE (IF (ATOM (CAR DOPVAL))
            (LIST DOPVAL)
            DOPVAL)
            IL:DO (COND
              ((ATOM ITEM) ; The ITEM is OPT.COMPILERERROR. Compile the call closed.
                (RETURN))
              ((OR (NULL (CAR ITEM))
                (= (CAR ITEM)
                  NUM-ARGS))
                (COND
                  ((CONSP (CDR ITEM))
                    (MAPC 'EMIT-BYTE (CDR ITEM))
                    (RETURN-FROM :CALL-PROCESSING))
                  (T
                    ; The ITEM is something like (0 . OPT.COMPILERERROR).
                    ; Compile the call closed.
                    (RETURN))))))
          ;; Either no DOPVAL or the DOPVAL failed. Compile as a closed call.
          (COND
            ((<= NUM-ARGS 255)
              (EMIT-BYTE-LIST (CASE NUM-ARGS
                ((0) ' (IL:FN0))
                ((1) ' (IL:FN1))
                ((2) ' (IL:FN2))
                ((3) ' (IL:FN3))
                ((4) ' (IL:FN4))
                (OTHERWISE ' (IL:FNX ,NUM-ARGS))))
              (EMIT-BYTE ' (:FN ,FN-TO-CALL)))
            (T
              ;; Lots of arguments. Call using APPLYFN.
              (EMIT-BYTE-LIST (PUSH-INTEGER NUM-ARGS))
              (EMIT-BYTE-LIST ' (IL:ACONST (:FN ,FN-TO-CALL)
                IL:APPLYFN))))))
          ; Call a function that lives in a variable
          (DVAR (EMIT-BYTE-LIST (PUSH-INTEGER NUM-ARGS))
            (COND
              ((EQ (DVAR-KIND FN-TO-CALL)
                :FUNCTION)
                (ASSERT (NOT (NULL (DVAR-LEVEL FN-TO-CALL)))
                  ' (FN-TO-CALL)
                  "BUG: The local function ~A should have a lexical level by now."
                  (DVAR-NAME FN-TO-CALL))
                (EMIT-BYTE-LIST ' (IL:GCONST (:LOCAL-FUNCTION ,FN-TO-CALL)))
                (COND
                  ((AND NIL (ZEROP (DVAR-LEVEL FN-TO-CALL)))
                    ; We used to do something different for an empty environment.
                    ; No non-locals -- use applyfn.
                    (EMIT-BYTE ' IL:APPLYFN))
                  (T (EMIT-BYTE-LIST (FETCH-HUNK (DVAR-LEVEL FN-TO-CALL)))
                    (EMIT-BYTE ' IL:ENVCALL))))
              (T (EMIT-BYTE-LIST (REF-VAR FN-TO-CALL))
                (EMIT-BYTE ' IL:APPLYFN))))
          (CONS (ECASE (FIRST FN-TO-CALL)
            ((:OPCODES) (EMIT-BYTE-LIST (REST FN-TO-CALL)))
            ((:LAMBDA)
              (EMIT-BYTE-LIST (PUSH-INTEGER NUM-ARGS))
              (LET ((DLAMBDA (SECOND FN-TO-CALL))
                (LAMBDA-LEVEL (THIRD FN-TO-CALL)))
                (COND
                  ((AND NIL (ZEROP LAMBDA-LEVEL))
                    ; We used to do something different for an empty environment.
                    ;; No closed-over variables: use APPLYFN.
                    (EMIT-BYTE-LIST ' (IL:GCONST (:LAMBDA 0 ,DLAMBDA)
                      IL:APPLYFN)))
                  (T
                    ;; This will need to be a closure. Find our best hunk for it and call using ENVCALL.
                    (EMIT-BYTE-LIST ' (IL:GCONST (:LAMBDA ,LAMBDA-LEVEL ,DLAMBDA)
                      ,@ (FETCH-HUNK LAMBDA-LEVEL
                        IL:ENVCALL))))))
              (T (ERROR "BUG: Weird argument to :CALL ~S" FN-TO-CALL)))
            (DECR DEPTH (1- NUM-ARGS))
            (ASSERT (OR (NOT DEPTH)
              (>= DEPTH 0))

```

```

      NIL "Depth went negative in ~S." :CALL)))
  ((:STKCALL)
   (EMIT-BYTE 'IL:APPLYFN)
   (DECR DEPTH (1+ (SECOND INST)))
   (ASSERT (OR (NOT DEPTH)
                (>= DEPTH 0))
            NIL "Depth went negative in ~S." :STKCALL))
  ((:RETURN) (EMIT-BYTE 'RETURN))
  ((:CLOSE)

```

; After digestion, this looks like this:
; (:CLOSE dvars hunk-slot . code).

```

  (CREATE-HUNK (+ (LIST-LENGTH (SECOND INST))
                  (IF (NULL *HUNK-MAP*)

```

; If this hunk is not at level 0, we need an extra hunk slot in order
; to link it to the previous one.

```

                      0
                      1))
  (THIRD INST)
  (CDAR *HUNK-MAP*)
  T)
  (LET* ((*LEVEL* (1+ *LEVEL*))
         (*HUNK-MAP* (CONS (CONS *LEVEL* (THIRD INST))
                           *HUNK-MAP*)))
        (SETQ DEPTH (ASSEMBLE-CODE (CDDDR INST)
                                     DEPTH BINDING-DEPTH))
        (ASSERT (>= DEPTH 0)
                  (OR (NOT DEPTH)
                      NIL "Depth went negative in ~S." :CLOSE))))))
  DEPTH)

```

:: Jump resolution

```

(DEFVAR *JUMP-LIST* NIL
 "A list of DJUMP and DTAG structures for use by jump resolution.")

```

```

(DEFCONSTANT +JUMP-CHOICES+
 ' ((:JUMP IL:JUMP IL:JUMPX IL:JUMPXX)
   (:FJUMP IL:FJUMP IL:FJUMPX (IL:TJUMP 2))
   (:TJUMP IL:TJUMP IL:TJUMPX (IL:FJUMP 2))
   (:NFJUMP IL:NFJUMP IL:NFJUMPX)
   (:NTJUMP IL:NTJUMP IL:NTJUMPX))

```

::: Alist from kinds of jumps to lists of choices for implementation of that kind of jump. See SPLICE-IN-JUMPS for details.

)

```

(DEFCONSTANT +JUMP-RANGE-SIZE-MAP+
 ' ((:JUMP (-128 . 3)
          (1 . 2)
          (18 . 1)
          (128 . 2)
          (32768 . 3))
   (:FJUMP (-128 . 4)
          (2 . 2)
          (18 . 1)
          (128 . 2)
          (32768 . 4))
   (:TJUMP (-128 . 4)
          (2 . 2)
          (18 . 1)
          (128 . 2)
          (32768 . 4))
   (:NFJUMP (-128 . 6)
          (128 . 2)
          (32768 . 6))
   (:NTJUMP (-128 . 6)
          (128 . 2)
          (32768 . 6)))

```

::: An Alist mapping kinds of jumps into an range-to-size table. The table is a list of pairs, sorted on the CAR. The shortest jump for a given distance is
::: the CDR of the first pair whose CAR is strictly greater than the distance.

)

```

(DEFCONSTANT +JUMP-SIZES+
 ' ((:JUMP 1 3)
   (:FJUMP 1 4)
   (:TJUMP 1 4)
   (:NFJUMP 2 6)
   (:NTJUMP 2 6))
 "Alist mapping kinds of jumps into the range of sizes for that kind, in bytes.")

```

```

(DEFUN RESOLVE-JUMPS (JUMP-LIST)
  (LET ((CUMULATIVE-UNCERTAINTY 0))

```

```

(IL:|for| JUMP-OR-TAG IL:|in| JUMP-LIST IL:|do| (ETYPESCASE JUMP-OR-TAG
(DTAG (SETF (DTAG-PC-UNCERTAINTY JUMP-OR-TAG)
CUMULATIVE-UNCERTAINTY))
(DJUMP (LET ((RANGE (ASSOC (DJUMP-KIND JUMP-OR-TAG)
+JUMP-SIZES+)))
(SETF (DJUMP-FORWARD-P JUMP-OR-TAG)
(> (DTAG-MIN-PC (DJUMP-TAG JUMP-OR-TAG))
(DJUMP-MIN-PC JUMP-OR-TAG)))
(SETF (DJUMP-MIN-SIZE JUMP-OR-TAG)
(SECOND RANGE))
(INCF CUMULATIVE-UNCERTAINTY
(SETF (DJUMP-SIZE-UNCERTAINTY JUMP-OR-TAG)
(- (THIRD RANGE)
(SECOND RANGE))))))))))

(IL:|while| (REDUCE-UNCERTAINTY JUMP-LIST))
(SPLICE-IN-JUMPS JUMP-LIST)

;; We need to convert the PC's in the tags from zero-based to START-PC-based.
(LET ((START-PC (START-PC-FROM-NT-COUNT-LOCAL (LENGTH (DCODE-NAME-TABLE *DCODE*))))
(IL:|for| JUMP-OR-TAG IL:|in| JUMP-LIST IL:|when| (DTAG-P JUMP-OR-TAG) IL:|do| (INCF (DTAG-PC JUMP-OR-TAG)
START-PC))))

(DEFUN REDUCE-UNCERTAINTY (JUMP-LIST)
(LET ((DECREASE-IN-UNCERTAINTY 0)
(INCREASE-IN-MIN-PC 0)
(CUMULATIVE-UNCERTAINTY 0))
(IL:FOR JUMP-OR-TAG IL:IN JUMP-LIST
IL:DO (ETYPESCASE JUMP-OR-TAG
(DTAG ; Just record the current uncertainty at this tag.
(SETF (DTAG-PC-UNCERTAINTY JUMP-OR-TAG)
CUMULATIVE-UNCERTAINTY)
(INCF (DTAG-MIN-PC JUMP-OR-TAG)
INCREASE-IN-MIN-PC))
(DJUMP
(INCF (DJUMP-MIN-PC JUMP-OR-TAG)
INCREASE-IN-MIN-PC)
(WHEN (> (DJUMP-SIZE-UNCERTAINTY JUMP-OR-TAG)
0) ; This is a jump we can hope to improve.
(LET ((TAG (DJUMP-TAG JUMP-OR-TAG))
(KIND (DJUMP-KIND JUMP-OR-TAG))
(JUMP JUMP-OR-TAG)
MIN-DISTANCE MAX-DISTANCE MIN-SIZE MAX-SIZE)
(COND
((DJUMP-FORWARD-P JUMP)
;; In computing the min and max distance between a forward jump and its tag, we must adjust for the
;; changes we've made so far this pass.
(SETQ MIN-DISTANCE (+ (- (DTAG-MIN-PC TAG)
(DJUMP-MIN-PC JUMP))
INCREASE-IN-MIN-PC))
(SETQ MAX-DISTANCE (+ (- (DTAG-PC-UNCERTAINTY TAG)
(+ DECREASE-IN-UNCERTAINTY CUMULATIVE-UNCERTAINTY)
MIN-DISTANCE)))
(T ; This situation is much simpler with backward jumps since both
; tag and jump are in the same units.
(SETQ MIN-DISTANCE (- (DTAG-MIN-PC TAG)
(DJUMP-MIN-PC JUMP)))
(SETQ MAX-DISTANCE (+ (- (DTAG-PC-UNCERTAINTY TAG)
CUMULATIVE-UNCERTAINTY)
MIN-DISTANCE))))
(SETQ MIN-SIZE (COMPUTE-JUMP-SIZE KIND MIN-DISTANCE))
(SETQ MAX-SIZE (COMPUTE-JUMP-SIZE KIND MAX-DISTANCE))
(WHEN (> MIN-SIZE (DJUMP-MIN-SIZE JUMP))
(INCF INCREASE-IN-MIN-PC (- MIN-SIZE (DJUMP-MIN-SIZE JUMP)))
(SETF (DJUMP-MIN-SIZE JUMP)
MIN-SIZE))
(LET ((NEW-SIZE-UNCERTAINTY (- MAX-SIZE MIN-SIZE)))
(WHEN (/= (DJUMP-SIZE-UNCERTAINTY JUMP)
NEW-SIZE-UNCERTAINTY)
(ASSERT (>= NEW-SIZE-UNCERTAINTY 0)
NIL "The size uncertainty went negative")
(INCF DECREASE-IN-UNCERTAINTY (- (DJUMP-SIZE-UNCERTAINTY JUMP)
NEW-SIZE-UNCERTAINTY))
(SETF (DJUMP-SIZE-UNCERTAINTY JUMP)
NEW-SIZE-UNCERTAINTY))
(INCF CUMULATIVE-UNCERTAINTY NEW-SIZE-UNCERTAINTY))))))

;; If we've either got no uncertainty left in the system or didn't manage to achieve anything this pass, give it up; we're done.
(NOT (OR (ZEROP CUMULATIVE-UNCERTAINTY)
(ZEROP DECREASE-IN-UNCERTAINTY))))

(DEFUN SPLICE-IN-JUMPS (JUMP-LIST)
(IL:FOR JUMP IL:IN JUMP-LIST

```

```

IL:DO (IF (DTAG-P JUMP)
  (SETF (DTAG-PC JUMP)
    (DTAG-MIN-PC JUMP))
  (LET* ((PTR (DJUMP-PTR JUMP))
    (TAG (DJUMP-TAG JUMP))
    (DISTANCE (- (DTAG-MIN-PC TAG)
      (DJUMP-MIN-PC JUMP)))
    (KIND (DJUMP-KIND JUMP))
    (SIZE (COMPUTE-JUMP-SIZE KIND DISTANCE))
    (CHOICES (CDR (ASSOC KIND +JUMP-CHOICES+))))
    (ECASE SIZE
      (1) ; One-byte jumps: JUMP, TJUMP, and FJUMP
        (COND
          ((= DISTANCE 1)
            (ASSERT (EQ KIND 'JUMP)
              NIL "BUG: SPLICE-IN-JUMPS found a wierd jump.")
            (RPLACA PTR 'IL:NOP))
          (T (RPLACA PTR (LIST (FIRST CHOICES)
            (- DISTANCE 2))))))
      (2) ; Two-byte jumps: JUMPX, FJUMPX, TJUMPX, NTJUMPX, and
        ; NFJUMPX
        (IL:RPLNODE PTR (SECOND CHOICES)
          (CONS (IF (< DISTANCE 0)
            (+ DISTANCE 256)
            DISTANCE)
            (CDR PTR))))
      (3 4) ; The three-byte jump is JUMPXX. Four-byte jumps are like
        ; (FJUMP 4) JUMPXX to implement TJUMPXX.
        (COND
          ((= SIZE 3)
            (RPLACA PTR (THIRD CHOICES)))
          (T (DECF DISTANCE)
            ; In the four-byte case, the true jump is from one byte later in the
            ; code stream.
            (RPLACA PTR (THIRD CHOICES))
            (RPLACD PTR (CONS 'IL:JUMPXX (CDR PTR)))
            (SETQ PTR (CDR PTR))))
        ;; At this point, PTR is the tail of the code starting with the JUMPXX instruction. We need to fix up the jump offset
        ;; here.
        (RPLACD PTR (LIST* (LOGAND (IL:LRSH DISTANCE 8)
          255)
          (LOGAND DISTANCE 255)
          (CDR PTR))))
      (6) ; Six-byte jumps are long NCJUMPXX's implemented by
        ; NCJUMPX 3 (JUMP 4) JUMPXX
        ; Take into account that the actual jump to the destination is
        ; three bytes later in the code stream.
        (DECF DISTANCE 3)
        (IL:RPLNODE PTR (SECOND CHOICES)
          ` (3 (IL:JUMP 4)
            IL:JUMPXX
            , (LOGAND (IL:LRSH DISTANCE 8)
              255)
            , (LOGAND DISTANCE 255)
            , @ (CDR PTR))))))

(DEFUN COMPUTE-JUMP-SIZE (KIND DISTANCE)
  (LET ((PAIRS (CDR (ASSOC KIND +JUMP-RANGE-SIZE-MAP+))))
    (IL:|find| PAIR IL:|in| PAIRS IL:|suchthat| (< DISTANCE (CAR PAIR)) IL:|finally| (RETURN (CDR PAIR)))))

```

;; Debugging jump resolution

```

(DEFUN PRETTY-JUMPS ()
  (IL:|for| JUMP-OR-TAG IL:|in| (REVERSE *JUMP-LIST*)
    IL:|collect| (ETYPESCASE JUMP-OR-TAG
      (DTAG ` (:TAG :MIN-PC , (DTAG-MIN-PC JUMP-OR-TAG)
        :PC-UNCERTAINTY
        , (DTAG-PC-UNCERTAINTY JUMP-OR-TAG)))
      (DJUMP ` (, (DJUMP-KIND JUMP-OR-TAG)
        :MIN-PC
        , (DJUMP-MIN-PC JUMP-OR-TAG)
        :MIN-SIZE
        , (DJUMP-MIN-SIZE JUMP-OR-TAG)
        :FORWARD-P
        , (DJUMP-FORWARD-P JUMP-OR-TAG)
        :SIZE-UNCERTAINTY
        , (DJUMP-SIZE-UNCERTAINTY JUMP-OR-TAG)
        :TAG
        (:MIN-PC , (DTAG-MIN-PC (DJUMP-TAG JUMP-OR-TAG)))))))

```

;; Conversion to binary

```
(DEFVAR *LOCAL-FN-FIXUPS*)
```

```

(DEFUN CONVERT-TO-BINARY (BYTE-LIST)
  (LET*
    ((CODELEN (LENGTH BYTE-LIST))
     (CODE-ARRAY (MAKE-ARRAY CODELEN :ELEMENT-TYPE '(UNSIGNED-BYTE 8)))
     (UNWIND-OFFSET (+ (IL:CEIL (+ (DCODE-NLOCALS *DCODE*)
                                   (DCODE-NFREEVARS *DCODE*))
                           IL:CELLSPERQUAD)
                       IL:CELLSPERQUAD))
     ; The number of PVAR slots, rounded up to a quadword
     ; boundary, plus an extra quadword for the Dolphin's hardware
     ; stack.
     )
    (PUSH-TAG-FIXUPS NIL))
  (IL:FOR BYTE IL:IN BYTE-LIST IL:AS CODE-INDEX IL:FROM 0
    IL:DO (SETF (AREF CODE-ARRAY CODE-INDEX)
                (ETYPESCASE BYTE
                  (SYMBOL
                    (IL:|fetch| IL:OP# IL:|of| (LET ((OPCODE (IL:\FINDOP BYTE)))
                                                  (ASSERT (NOT (NULL OPCODE)))
                                                  NIL "BUG: Can't find purported opcode ~S" BYTE)
                                                  OPCODE)))
                    ; Symbols represent real D-machine opcodes.
                    ((UNSIGNED-BYTE 8)
                     ; Small integers generally represent themselves, usually either
                     ; as arguments to opcodes or filler bytes for fixups.
                     BYTE)
                    (CONS
                     ; Conses are either fixups or opcodes that take their argument
                     ; inside their bytecode.
                     (CASE (FIRST BYTE)
                       ((:SYM)
                        (PUSH (LIST CODE-INDEX (SECOND BYTE))
                              (DCODE-SYM-FIXUPS *DCODE*)))
                        0)
                       ((:LIT)
                        (PUSH (LIST CODE-INDEX (SECOND BYTE))
                              (DCODE-LIT-FIXUPS *DCODE*)))
                        0)
                       ((:FN)
                        (PUSH (LIST CODE-INDEX (SECOND BYTE))
                              (DCODE-FN-FIXUPS *DCODE*)))
                        0)
                       ((:TYPE)
                        (PUSH (LIST CODE-INDEX (SECOND BYTE))
                              (DCODE-TYPE-FIXUPS *DCODE*)))
                        0)
                       ((:LAMBDA)
                        (PUSH (LIST CODE-INDEX (DCODE-FROM-DLAMBDA (THIRD BYTE)
                                                                    (SECOND BYTE)))
                              (DCODE-LIT-FIXUPS *DCODE*)))
                        0)
                       ((:LOCAL-FUNCTION)
                        (DESTRUCTURING-BIND (IGNORE DCODE-FIXUP DCODE-FOR-FIXUP)
                          (FIND (SECOND BYTE)
                                *LOCAL-FN-FIXUPS* :TEST 'EQ :KEY 'CAR)
                          (DECLARE (IGNORE IGNORE))
                          (ETYPESCASE DCODE-FIXUP
                            (DLAMBDA (SETQ DCODE-FIXUP (DCODE-FROM-DLAMBDA DCODE-FIXUP
                                                                              (DVAR-LEVEL (SECOND BYTE)))))
                            (DCODE NIL))
                          (PUSH (LIST *DCODE* CODE-INDEX DCODE-FIXUP)
                                (DCODE-LOCAL-FN-FIXUPS DCODE-FOR-FIXUP)))
                        0)
                       ((:UNWIND) (+ UNWIND-OFFSET (SECOND BYTE)))
                       ((:PUSH-TAG)
                        (PUSH (LIST CODE-INDEX (DTAG-PC (SECOND BYTE)))
                              PUSH-TAG-FIXUPS)
                        0)
                       (OTHERWISE (LET ((RANGE (IL:|fetch| IL:OP# IL:|of| (LET ((OPCODE (IL:\FINDOP
                                                                              (FIRST BYTE)))
                                                                              (ASSERT (NOT (NULL OPCODE)))
                                                                              NIL "BUG: Can't find
                                                                              purported opcode ~S"
                                                                              (FIRST BYTE))
                                                                              OPCODE))))
                                     (ASSERT (AND (CONSP RANGE)
                                                  (INTEGERP (FIRST RANGE))
                                                  (INTEGERP (SECOND RANGE)))
                                     NIL "BUG: Argument given to the ~A opcode, but it doesn't
                                     take one." (FIRST BYTE))
                                     (ASSERT (<= 0 (SECOND BYTE)
                                               (- (SECOND RANGE)
                                                  (FIRST RANGE)))
                                     NIL "BUG: Illegal argument to ~A opcode: ~S" (FIRST BYTE)
                                     (SECOND BYTE))
                                     (+ (FIRST RANGE)
                                       (SECOND BYTE))))))))))
  ;; Do the push-tag fixups

```

;; Do the push-tag fixups


```

(IL:|for| FIXUP IL:|in| PUSH-TAG-FIXUPS IL:|do| (DESTRUCTURING-BIND (OFFSET PC)
FIXUP
(SETF (AREF CODE-ARRAY OFFSET)
(LDB (BYTE 8 8)
PC))
(SETF (AREF CODE-ARRAY (1+ OFFSET))
(LDB (BYTE 8 0)
PC))))

```

;; We're done making the bytes. Stuff the code-array into the DCODE.

```

(SETF (DCODE-CODE-ARRAY *DCODE*)
CODE-ARRAY))

```

;; Setting up the debugging information

```

(DEFUN COMPUTE-DEBUGGING-INFO (DLAMBDA)
(SETF (DCODE-DEBUGGING-INFO *DCODE*)
`((,@(MAPCAR #'DVAR-NAME (DLAMBDA-REQUIRED DLAMBDA))
,@(AND (DLAMBDA-OPTIONAL DLAMBDA)
(CONS '&OPTIONAL (MAPCAR #'(LAMBDA (OPT-VAR)
(DVAR-NAME (FIRST OPT-VAR)))
(DLAMBDA-OPTIONAL DLAMBDA))))))
,@(AND (DLAMBDA-REST DLAMBDA)
(NOT (EQ :IGNORED (DLAMBDA-REST DLAMBDA))))
(LIST '&REST (DVAR-NAME (DLAMBDA-REST DLAMBDA))))))
,@(AND (OR (DLAMBDA-KEY DLAMBDA)
(DLAMBDA-ALLOW-OTHER-KEYS DLAMBDA))
(CONS '&KEY (MAPCAR #'FIRST (DLAMBDA-KEY DLAMBDA))))))
,@(AND (DLAMBDA-ALLOW-OTHER-KEYS DLAMBDA)
'(&ALLOW-OTHER-KEYS)))
,@(AND (DLAMBDA-ARG-TYPE DLAMBDA)
'(:INTERLISP T))))))

```

;; Fixup resolution and DCODE internig

```

(DEFUN START-PC-FROM-NT-COUNT (NT-COUNT)

```

;; If a given compiled-code object has a name table NT-COUNT entries long, what will its starting PC be?

;; IF YOU CHANGE THIS FUNCTION, CHANGE START-PC-FROM-NT-COUNT-LOCAL TO MATCH IT.

```

(LET* ((NT-SIZE (IL:CEIL (1+ (IL:UNFOLD NT-COUNT (IL:CONSTANT (IL:WORDSPERNAMEENTRY))))
IL:WORDSPERQUAD))
(NT-WORDS (IF (ZEROP NT-COUNT)
IL:WORDSPERQUAD
(+ NT-SIZE NT-SIZE))))
(* (+ (IL:|fetch| (IL:CODEARRAY IL:OVERHEADWORDS) IL:|of| T)
NT-WORDS IL:WORDSPERCELL)
IL:BYTESPERWORD)))

```

```

(DEFUN START-PC-FROM-NT-COUNT-LOCAL (NT-COUNT)

```

;; If a given compiled-code object has a name table NT-COUNT entries long, what will its starting PC be? This version computes the value at run-time, rather than having the architecture compiled in, as START-PC-FROM-NT-COUNT does.

;; If you change this function, change START-PC-FROM-NT-COUNT to match!

```

(LET* ((NT-SIZE (IL:CEIL (1+ (IL:LLSH NT-COUNT (COND
((IL:FMEMB :3-BYTE (COMPILER::ENV-TARGET-ARCHITECTURE
COMPILER::*ENVIRONMENT*))
1)
((AND IL:CROSSCOMPILING (IL:FMEMB :3-BYTE-INIT (
COMPILER::ENV-TARGET-ARCHITECTURE
COMPILER::*ENVIRONMENT*
)))
1)
(T 0))))))
IL:WORDSPERQUAD))
(NT-WORDS (IF (ZEROP NT-COUNT)
IL:WORDSPERQUAD
(+ NT-SIZE NT-SIZE))))
(* (+ (IL:|fetch| (IL:CODEARRAY IL:OVERHEADWORDS) IL:|of| T)
NT-WORDS IL:WORDSPERCELL)
IL:BYTESPERWORD)))

```

```

(DEFUN ALLOCATE-CODE-BLOCK (NT-COUNT CODE-LEN)

```

;; Return a code-array that is large enough to hold a compiled function with a name-table NT-COUNT entries long and with CODE-LEN bytecodes. Also return, as a second value, the index in that code-array of the place to put the first bytecode.

```

(LET* ((START-PC (START-PC-FROM-NT-COUNT NT-COUNT))
(TOTAL-SIZE (IL:CEIL (+ START-PC CODE-LEN)
IL:BYTESPERQUAD))

```

```

        (CODE-BASE (IL:\\ALLOC.CODE.BLOCK TOTAL-SIZE (IL:CEIL (1+ (CEILING START-PC IL:BYTESPERCELL))
                                                    IL:CELLSPERQUAD))))
        (VALUES CODE-BASE START-PC)))

```

(DEFUN **FIXUP-PTR** (BASE OFFSET PTR)

;; Fix up a pointer within a code block.

```

(LET ((LOW (IL:\\LOLOC PTR)))
  (IL:UNINTERRUPTABLY
    (IL:\\ADDRF PTR)
    (COND
      ((IL:FMEMB :4-BYTE COMPILER::*HOST-ARCHITECTURE*)
        (IL:\\PUTBASEBYTE BASE OFFSET (IL:LOGOR (IL:\\GETBASEBYTE BASE OFFSET)
                                                  (IL:LRSH (IL:\\HILOC PTR)
                                                    8))))
      (IL:\\PUTBASEBYTE BASE (+ 1 OFFSET)
        (IL:LOGOR (IL:\\GETBASEBYTE BASE (+ 1 OFFSET))
          (LOGAND 255 (IL:\\HILOC PTR))))
      (IL:\\PUTBASEBYTE BASE (+ 2 OFFSET)
        (IL:LRSH LOW 8))
      (IL:\\PUTBASEBYTE BASE (+ 3 OFFSET)
        (LOGAND LOW 255)))
    ((IL:FMEMB :3-BYTE COMPILER::*HOST-ARCHITECTURE*)
      (IL:\\PUTBASEBYTE BASE OFFSET (IL:LOGOR (IL:\\GETBASEBYTE BASE OFFSET)
                                                (LOGAND 255 (IL:\\HILOC PTR))))
      (IL:\\PUTBASEBYTE BASE (+ 1 OFFSET)
        (IL:LRSH LOW 8))
      (IL:\\PUTBASEBYTE BASE (+ 2 OFFSET)
        (LOGAND LOW 255))))
  PTR))

```

(DEFUN **FIXUP-PTR-NO-REF** (BASE OFFSET PTR)

;; Only used for code self-references: doesn't ADDRFB the pointer.

```

(LET ((LOW (IL:\\LOLOC PTR)))
  (IL:UNINTERRUPTABLY
    (COND
      ((IL:FMEMB :4-BYTE COMPILER::*HOST-ARCHITECTURE*)
        (IL:\\PUTBASEBYTE BASE OFFSET (IL:LOGOR (IL:\\GETBASEBYTE BASE OFFSET)
                                                  (IL:LRSH (IL:\\HILOC PTR)
                                                    8))))
      (IL:\\PUTBASEBYTE BASE (+ 1 OFFSET)
        (IL:LOGOR (IL:\\GETBASEBYTE BASE (+ 1 OFFSET))
          (LOGAND 255 (IL:\\HILOC PTR))))
      (IL:\\PUTBASEBYTE BASE (+ 2 OFFSET)
        (IL:LRSH LOW 8))
      (IL:\\PUTBASEBYTE BASE (+ 3 OFFSET)
        (LOGAND LOW 255)))
    ((IL:FMEMB :3-BYTE COMPILER::*HOST-ARCHITECTURE*)
      (IL:\\PUTBASEBYTE BASE OFFSET (IL:LOGOR (IL:\\GETBASEBYTE BASE OFFSET)
                                                (LOGAND 255 (IL:\\HILOC PTR))))
      (IL:\\PUTBASEBYTE BASE (+ 1 OFFSET)
        (IL:LRSH LOW 8))
      (IL:\\PUTBASEBYTE BASE (+ 2 OFFSET)
        (LOGAND LOW 255))))
  PTR))

```

(DEFUN **FIXUP-SYMBOL** (BASE OFFSET SYMBOL)

;; Fix up an atom number (GVAR or FNx or ACONST) in a compiled-code object.

```

(LET ((WORD (COND
  ((SYMBOLP SYMBOL)
    (IL:\\LOLOC SYMBOL))
  (T SYMBOL)))
  (HIBYTE (COND
    ((SYMBOLP SYMBOL)
      (IL:\\HILOC SYMBOL))
    (T 0))))
  (COND
    ((IL:FMEMB :4-BYTE COMPILER::*HOST-ARCHITECTURE*)
      ;; For 4-byte-atom architecture, treat it as a pointer.
      (IL:UNINTERRUPTABLY
        (IL:\\PUTBASEBYTE BASE OFFSET (IL:LOGOR (IL:\\GETBASEBYTE BASE OFFSET)
                                                  (IL:LRSH HIBYTE 8)))
        (IL:\\PUTBASEBYTE BASE (+ 1 OFFSET)
          (IL:LOGOR (IL:\\GETBASEBYTE BASE (+ 1 OFFSET))
            (LOGAND 255 HIBYTE)))
        (IL:\\PUTBASEBYTE BASE (+ 2 OFFSET)
          (LOGAND 255 (IL:LRSH WORD 8)))
        (IL:\\PUTBASEBYTE BASE (+ 3 OFFSET)
          (LOGAND WORD 255)))
      WORD)
    ((IL:FMEMB :3-BYTE COMPILER::*HOST-ARCHITECTURE*)

```

;; For 4-byte-atom architecture, treat it as a pointer.

```
(IL:UNINTERRUPTABLY
  (IL:\\PUTBASEBYTE BASE OFFSET (IL:LOGOR (IL:\\GETBASEBYTE BASE OFFSET)
                                            (LOGAND 255 HIBYTE)))
  (IL:\\PUTBASEBYTE BASE (+ 1 OFFSET)
    (LOGAND 255 (IL:LRSH WORD 8)))
  (IL:\\PUTBASEBYTE BASE (+ 2 OFFSET)
    (LOGAND WORD 255)))
WORD)
(T
  (IL:\\PUTBASEBYTE BASE OFFSET (IL:LRSH WORD 8))
  (IL:\\PUTBASEBYTE BASE (1+ OFFSET)
    (LOGAND WORD 255))
  WORD)))
```

; Otherwise, it's a 2-byte #.

(DEFUN **FIXUP-NTENTRY** (BASE OFFSET SYMBOL)

;; Fix up a NAMETABLE entry. OFFSET is in BYTES.

```
(LET ((WORD (COND
  ((SYMBOLP SYMBOL)
    (IL:\\LOLOC SYMBOL))
  (T SYMBOL)))
  (HIBYTE (COND
  ((SYMBOLP SYMBOL)
    (IL:\\HILOC SYMBOL))
  (T 0))))
(COND
  ((IL:FMEMB :3-BYTE COMPILER::*HOST-ARCHITECTURE*)
```

;; For 3-byte-atom architecture, treat it as a pointer.

(IL:* SETQ OFFSET (IL:ADD1 OFFSET))

; Pointer WAS 3 bytes, right-justified in a 4-byte field

```
(IL:UNINTERRUPTABLY
  (IL:\\PUTBASEBYTE BASE OFFSET (IL:LOGOR (IL:\\GETBASEBYTE BASE OFFSET)
                                            (LOGAND 255 (IL:LRSH HIBYTE 8))))
  (IL:\\PUTBASEBYTE BASE (+ 1 OFFSET)
    (IL:LOGOR (IL:\\GETBASEBYTE BASE (+ 1 OFFSET))
      (LOGAND 255 HIBYTE)))
  (IL:\\PUTBASEBYTE BASE (+ 2 OFFSET)
    (LOGAND 255 (IL:LRSH WORD 8)))
  (IL:\\PUTBASEBYTE BASE (+ 3 OFFSET)
    (LOGAND WORD 255)))
WORD)
(T
  (IL:\\PUTBASEBYTE BASE OFFSET (IL:LRSH WORD 8))
  (IL:\\PUTBASEBYTE BASE (1+ OFFSET)
    (LOGAND WORD 255))
  WORD)))
```

; Otherwise, it's a 2-byte #.

(DEFUN **FIXUP-WORD** (BASE OFFSET WORD)

;; Fix up a 16-bit loadtime constant in the code stream. Used now only for type #s in a compiled-code object.

```
(IL:\\PUTBASEBYTE BASE OFFSET (IL:LRSH WORD 8))
(IL:\\PUTBASEBYTE BASE (1+ OFFSET)
  (LOGAND WORD 255))
WORD)
```

(DEFUN **INTERN-DCODE** (DCODE &OPTIONAL (COPY-P (ARRAYP (DCODE-CODE-ARRAY DCODE))))

;;; NOTE: For unfortunately unavoidable performance reasons, this code is essentially duplicated in the FASL loader. If you change something here, change it there as well. And don't change anything unless you've got a pointy hat with a lot of stars on it.

;; NTSIZE and NTBYTESIZE are the length of one-half of the name table in words and bytes, respectively. NTWORDS is the length of the whole name table in words.

```
(LET* ((NAME-TABLE (DCODE-NAME-TABLE DCODE))
  (NAME-TABLE-SIZE (LENGTH NAME-TABLE))
  (NTSIZE (IL:CEIL (1+ (IL:UNFOLD NAME-TABLE-SIZE (IL:CONSTANT (IL:WORDSPERNAMEENTRY))))
    IL:WORDSPERQUAD))
  (NTBYTESIZE (* NTSIZE IL:BYTESPERWORD))
  (NTWORDS (IF (ZEROP NAME-TABLE-SIZE)
    IL:WORDSPERQUAD
    (+ NTSIZE NTSIZE))))
(OVERHEADBYTES (* (IL:[fetch] (IL:FNHEADER IL:OVERHEADWORDS) IL:[of] T)
  IL:BYTESPERWORD))
RAW-CODE FVAROFFSET RESULT)
```

;; Copy the bytes into a raw code block if necessary.

```
(IF (NULL COPY-P)
  (SETQ RAW-CODE (DCODE-CODE-ARRAY DCODE))
  (LET ((CODE-ARRAY (DCODE-CODE-ARRAY DCODE))
    (MULTIPLE-VALUE-BIND (CODE-BLOCK START-INDEX)
      (ALLOCATE-CODE-BLOCK (LENGTH NAME-TABLE)
        (LENGTH CODE-ARRAY))
      (IL:[for] CA-INDEX IL:[from] 0 IL:[to] (1- (LENGTH CODE-ARRAY)) IL:[as] CB-INDEX IL:[from] START-INDEX
```

```

      IL:|do| (IL:|\\PUTBASEBYTE CODE-BLOCK CB-INDEX (AREF CODE-ARRAY CA-INDEX)))
      (SETQ RAW-CODE CODE-BLOCK)))
;; Set up the free-variable lookup name table.
(DO ((END (LENGTH NAME-TABLE))
     (I 0 (1+ I))
     (INDEX OVERHEADBYTES (+ INDEX (IL:BYTESPERNAMEENTRY))))
    ((>= I END))
  (LET ((ENTRY (ELT NAME-TABLE I)))
    (FIXUP-NTENTRY RAW-CODE INDEX (THIRD ENTRY)) ; Atom index (or atom itself for 3-byte case)
    (FASL::FIXUP-NTOFFSET RAW-CODE (+ INDEX NTBYTESIZE)
      (IL:LLSH (FIRST ENTRY)
        14)
      (SECOND ENTRY)) ; VAR TYPE AND OFFSET
    (WHEN (AND (NULL FVAROFFSET)
               (= (FIRST ENTRY)
                  +FVAR-CODE+))
      (SETQ FVAROFFSET (FLOOR INDEX IL:BYTESPERWORD)))))
;; Fill in the fixed-size fields at the front of the block.
(IL:|replace| (IL:FNHEADER IL:NA) IL:|of| RAW-CODE IL:|with| (DCODE-NUM-ARGS DCODE))
(IL:|replace| (IL:FNHEADER IL:PV) IL:|of| RAW-CODE IL:|with| (1- (CEILING (+ (DCODE-NLOCALS DCODE)
                                     (DCODE-NFREEVARS DCODE)))
  IL:CELLSPERQUAD)))
;; The start-pc is after the fixed-size stuff, the name-table, and a cell in which to store the debugging info.
(IL:|replace| (IL:FNHEADER IL:STARTPC) IL:|of| RAW-CODE IL:|with| (+ OVERHEADBYTES (* NTWORDS IL:BYTESPERWORD)
  IL:BYTESPERCELL))
(IL:|replace| (IL:FNHEADER IL:ARGTYPE) IL:|of| RAW-CODE IL:|with| (DCODE-ARG-TYPE DCODE))
(LET ((FRAME-NAME (DCODE-FRAME-NAME DCODE)))
  (IL:UNINTERRUPTABLY
    (IL:|\\ADDRESS FRAME-NAME)
    (IL:|replace| (IL:FNHEADER IL:|\\#FRAMENAME) IL:|of| RAW-CODE IL:|with| FRAME-NAME)))
(IL:|replace| (IL:FNHEADER IL:NTSIZE) IL:|of| RAW-CODE IL:|with| (IF (ZEROP NAME-TABLE-SIZE)
  0
  NTSIZE))
(IL:|replace| (IL:FNHEADER IL:NLOCALS) IL:|of| RAW-CODE IL:|with| (DCODE-NLOCALS DCODE))
(IL:|replace| (IL:FNHEADER IL:FVAROFFSET) IL:|of| RAW-CODE IL:|with| (OR FVAROFFSET 0))
(IL:|replace| (IL:FNHEADER IL:CLOSUREP) IL:|of| RAW-CODE IL:|with| (EQ :CLOSURE (DCODE-CLOSURE-P DCODE)))
(IL:|replace| (IL:FNHEADER IL:FIXED) IL:|of| RAW-CODE IL:|with| T)
;; Fill in the debugging information and perform the fixups. There WAS a + 1 in the + OVERHEADBYTES... is to allow for the fact that four
;; bytes are allocated for the debugging information, but pointers are only three bytes long, so we right-justify the pointer in the cell.
(FIXUP-PTR RAW-CODE (+ OVERHEADBYTES (* NTWORDS IL:BYTESPERWORD))
  (DCODE-DEBUGGING-INFO DCODE))
(LET ((START-PC (IL:|fetch| (IL:FNHEADER IL:STARTPC) IL:|of| RAW-CODE)))
  (DO ((END (LENGTH (DCODE-FN-FIXUPS DCODE)))
       (I 0 (1+ I))
       ((>= I END))
      (DESTRUCTURING-BIND (OFFSET ITEM)
        (ELT (DCODE-FN-FIXUPS DCODE)
          I)
        (FIXUP-SYMBOL RAW-CODE (+ START-PC OFFSET)
          ITEM)))
    ((END (LENGTH (DCODE-SYM-FIXUPS DCODE)))
     (I 0 (1+ I))
     ((>= I END))
    (DESTRUCTURING-BIND (OFFSET ITEM)
      (ELT (DCODE-SYM-FIXUPS DCODE)
        I)
      (FIXUP-SYMBOL RAW-CODE (+ START-PC OFFSET)
        ITEM)))
    ((END (LENGTH (DCODE-LIT-FIXUPS DCODE)))
     (I 0 (1+ I))
     ((>= I END))
    (DESTRUCTURING-BIND (OFFSET ITEM)
      (ELT (DCODE-LIT-FIXUPS DCODE)
        I)
      (FIXUP-PTR RAW-CODE (+ START-PC OFFSET)
        (TYPECASE ITEM
          (DCODE (INTERN-DCODE ITEM)) ; Some kinds of literals get special treatment.
          (COMPILER::EVAL-WHEN-LOAD (EVAL (COMPILER::EVAL-WHEN-LOAD-FORM ITEM)))
          (OTHERWISE ITEM)))))
    ((END (LENGTH (DCODE-TYPE-FIXUPS DCODE)))
     (I 0 (1+ I))
     ((>= I END))
    (DESTRUCTURING-BIND (OFFSET ITEM)
      (ELT (DCODE-TYPE-FIXUPS DCODE)
        I)
      (FIXUP-WORD RAW-CODE (+ START-PC OFFSET)
        (IL:|\\RESOLVE.TYPENUMBER ITEM)))))
;; Wrap this up in a closure-object if requested.
(SETF (DCODE-INTERN-RESULT DCODE)
  (SETQ RESULT (IF (EQ :FUNCTION (DCODE-CLOSURE-P DCODE))
    (IL:MAKE-COMPILED-CLOSURE RAW-CODE NIL)
    RAW-CODE)))

```

;; Finally,do the mutual code reference fixups, if necessary.

(**PERFORM-LOCAL-FN-FIXUPS** DCODE)
RESULT))

```
(DEFUN PERFORM-LOCAL-FN-FIXUPS (DCODE)
  (LET ((FIXUP-LIST (DCODE-LOCAL-FN-FIXUPS DCODE)))
    (UNLESS (NULL FIXUP-LIST)
      (ASSERT (NOT (NULL (DCODE-INTERN-RESULT DCODE)))
        ' (DCODE)
        "BUG: Attempt to fix up an uninterned DCODE.")
      (MAPC #' (LAMBDA (FIXUP)
        (DESTRUCTURING-BIND (DCODE-TO-FIX OFFSET DCODE-TO-INSTALL)
          FIXUP
          (FLET ((GET-CODE (THING)
            (IF (TYPEP THING 'IL:COMPILED-CLOSURE)
              (IL:FETCH (IL:COMPILED-CLOSURE IL:FNHEADER) IL:OF THING)
              THING))
            (GET-FIXUP-VALUE (DCODE)
              (OR (DCODE-INTERN-RESULT DCODE)
                (INTERN-DCODE DCODE))))
          (LET* ((VALUE-TO-FIX (GET-CODE (GET-FIXUP-VALUE DCODE-TO-FIX)))
            (VALUE-TO-INSTALL (GET-FIXUP-VALUE DCODE-TO-INSTALL)))
            (IF (EQ DCODE-TO-FIX DCODE-TO-INSTALL)
              (FIXUP-PTR-NO-REF VALUE-TO-FIX (+ (IL:|fetch| (IL:FNHEADER
                IL:STARTPC)
                IL:|of| VALUE-TO-FIX)
                OFFSET)
                VALUE-TO-INSTALL)
              (FIXUP-PTR VALUE-TO-FIX (+ (IL:|fetch| (IL:FNHEADER IL:STARTPC)
                IL:|of| VALUE-TO-FIX)
                OFFSET)
                VALUE-TO-INSTALL))))))
        FIXUP-LIST))))
```

;; Arrange for the correct compiler to be used

(IL:PUTPROPS **IL:D-ASSEM** **IL:FILETYPE** COMPILE-FILE)

;; Arrange for the proper makefile environment

(IL:PUTPROPS **IL:D-ASSEM** **IL:MAKEFILE-ENVIRONMENT** (:READTABLE "XCL" :PACKAGE (DEFPACKAGE "D-ASSEM"
 (:USE "LISP" "XCL"))))

(IL:DECLARE\ : IL:EVAL@COMPILE IL:DONTCOPY

(IL:FILESLOAD (IL:LOADCOMP)
 IL:LLBASIC IL:LLCODE IL:LLGC IL:MODARITH)
)

(IL:PUTPROPS **IL:D-ASSEM** **IL:COPYRIGHT** ("Xerox Corporation" 1986 1987 1988 1990 1991 1992))

FUNCTION INDEX

ALLOCATE-CODE-BLOCK	25	END-BYTES	5	MAX-ARG	7
ASSEMBLE	17	FETCH-HUNK	6	MAXF	3
ASSEMBLE-CODE	17	FIXUP-NTENTRY	27	PERFORM-LOCAL-FN-FIXUPS	29
ASSEMBLE-FUNCTION	7	FIXUP-PTR	26	PRETTY-JUMPS	23
CHOOSE-OP	5	FIXUP-PTR-NO-REF	26	PUSH-INTEGER	7
COMPUTE-DEBUGGING-INFO	25	FIXUP-SYMBOL	26	REACH-TAGS	15
COMPUTE-JUMP-SIZE	23	FIXUP-WORD	27	REDUCE-UNCERTAINTY	22
CONVERT-TO-BINARY	24	GATHER-ROOTS	15	REF-VAR	6
COPY-LAP-CODE	3	GATHER-TAGS	15	RELEASE-DCODE	3
COPY-LAP-FN	3	GENERATE-ARG-CHECK	13	RESOLVE-JUMPS	21
CREATE-HUNK	3	GENERATE-EASY-ENTRY	12	SPLICE-IN-JUMPS	22
DCODE-FROM-DLAMBDA	8	GENERATE-HARD-ENTRY	13	STACK-ANALYZE	15
DETERMINE-LOCAL-FN-LEXICAL-LEVEL	10	GENERATE-KEY	14	STACK-ANALYZE-CODE	15
DIGEST-CODE	10	GENERATE-OPT-AND-REST	14	START-BYTES	4
DIGEST-FUNCTION	9	INSTALL-LOCAL	12	START-PC-FROM-NT-COUNT	25
DLAMBDA-FROM-LAMBDA	8	INSTALL-VAR	12	START-PC-FROM-NT-COUNT-LOCAL	25
DVAR-FROM-LAP-VAR	11	INTERN-DCODE	27	STORE-DIGEST-INFO	11
EASY-ENTRY-P	12	INTERN-TAG	12	STORE-VAR	6
EMIT-BYTE	4	INTERN-VAR	12	TYPE-NAME-FROM-SIZE	3
EMIT-BYTE-LIST	5	LAP-VAR-ID	12		

CONSTANT INDEX

+CONSTANT-OPCODES+	4	+JUMP-SIZES+	21	+NLAMBDA-NO-SPREAD+	4
+FVAR-CODE+	4	+LAMBDA-NO-SPREAD+	4	+NLAMBDA-SPREAD+	4
+IVAR-CODE+	3	+LAMBDA-SPREAD+	4	+PVAR-CODE+	4
+JUMP-CHOICES+	21	+MAX-ALLOWABLE-PVAR-COUNT+	9	+SLOW-FVAR-SLOT+	9
+JUMP-RANGE-SIZE-MAP+	21	+MAX-ALLOWABLE-SPECIAL-COUNT+	9		

VARIABLE INDEX

BOUND-SPECIALS	9	*DCODE*	7	*ENDING-DEPTH*	15	*HUNK-SIZE*	8	*LOCAL-FN-FIXUPS*	23
BYTE-COUNT	4	*DTAG-ENV*	7	*FREE-VARS*	9	*JUMP-LIST*	21	*PVAR-COUNT*	8
BYTES	4	*DVAR-ENV*	7	*HUNK-MAP*	7	*LEVEL*	7	*STACK-ENV*	7

STRUCTURE INDEX

DCODE	2	DJUMP	2	DLAMBDA	2	DTAG	2	DVAR	3
-------------	---	-------------	---	---------------	---	------------	---	------------	---

PROPERTY INDEX

IL:D-ASSEM	29
------------------	----
