

File created: 3-Jul-2022 17:43:01 {DSK}<users>kaplan>local>medley3.5>working-medley>sources>CMLFORM
AT.;2

previous date: 16-May-90 13:19:59 {DSK}<users>kaplan>local>medley3.5>working-medley>sources>CMLFORMAT.;1

Read Table: INTERLISP

Package: INTERLISP

Format: XCCS

::
:: Copyright (c) 1986-1990 by Venue & Xerox Corporation.

(RPAQQ **CMLFORMATCOMS**

(; The FORMAT facility

```
(STRUCTURES FORMAT-ERROR)
(FUNCTIONS MAKE-DISPATCH-VECTOR SCALE-EXPONENT SCALE-EXPT-AUX)
(FUNCTIONS FORMAT-ERROR)
(VARIABLES *DIGIT-STRING* *DIGITS*)
(FUNCTIONS FLONUM-TO-STRING FORMAT-WITH-CONTROL-STRING FORMAT-STRINGIFY-OUTPUT POP-FORMAT-ARG
  WITH-FORMAT-PARAMETERS NEXTCHAR FORMAT-PEEK FORMAT-FIND-CHAR)
(FUNCTIONS FORMAT-GET-PARAMETER PARSE-FORMAT-OPERATION FORMAT-FIND-COMMAND CL:FORMAT SUB-FORMAT
  FORMAT-CAPITALIZATION FORMAT-ESCAPE FORMAT-SEMICOLON-ERROR FORMAT-UNTAGGED-CONDITION
  FORMAT-FUNNY-CONDITION FORMAT-BOOLEAN-CONDITION FORMAT-CONDITION FORMAT-ITERATION
  FORMAT-DO-ITERATION FORMAT-GET-TRAILING-SEGMENTS FORMAT-GET-SEGMENTS MAKE-PAD-SEGS
  FORMAT-ROUND-COLUMNS FORMAT-JUSTIFICATION FORMAT-TERPRI FORMAT-FRESHLINE FORMAT-PAGE FORMAT-TILDE
  FORMAT-EAT-WHITESPACE FORMAT-NEWLINE FORMAT-PLURAL FORMAT-SKIP-ARGUMENTS FORMAT-INDIRECTION
  FORMAT-TAB FORMAT-PRINC FORMAT-PRIN1 FORMAT-PRINT-CHARACTER FORMAT-PRINT-NAMED-CHARACTER
  FORMAT-ADD-COMMAS FORMAT-WRITE-FIELD FORMAT-PRINT-NUMBER FORMAT-PRINT-SMALL-CARDINAL
  FORMAT-PRINT-CARDINAL FORMAT-PRINT-CARDINAL-AUX FORMAT-PRINT-ORDINAL FORMAT-PRINT-OLD-ROMAN
  FORMAT-PRINT-ROMAN FORMAT-PRINT-DECIMAL FORMAT-PRINT-BINARY FORMAT-PRINT-OCTAL
  FORMAT-PRINT-HEXADECIMAL FORMAT-PRINT-RADIX FORMAT-PRINT-RADIX-AUX FORMAT-FIXED FORMAT-FIXED-AUX
  FORMAT-EXPONENTIAL FORMAT-EXPONENT-MARKER FORMAT-EXP-AUX FORMAT-GENERAL-FLOAT FORMAT-GENERAL-AUX
  FORMAT-DOLLARS)
(FUNCTIONS CHARPOS WHITESPACE-CHAR-P)
(FUNCTIONS NAME-ARRAY)
(VARIABLES *FORMAT-ARGUMENTS* *FORMAT-CONTROL-STRING* *FORMAT-DISPATCH-TABLE* *FORMAT-INDEX*
  *FORMAT-LENGTH* *FORMAT-ORIGINAL-ARGUMENTS* CARDINAL-ONES CARDINAL-TENS CARDINAL-TEENS
  CARDINAL-PERIODS ORDINAL-ONES ORDINAL-TENS)
(DECLARE%: DONTEVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILERVERS (ADDVARS (NLAMA)
  (NLAML)
  (LAMA)))
```

:: Arrange to use the correct compiler.

(PROP FILETYPE CMLFORMAT))

:: The FORMAT facility

```
(DEFINE-CONDITION FORMAT-ERROR (CL:ERROR)
  (ARGS)
  [:REPORT (CL:LAMBDA (CONDITION *STANDARD-OUTPUT*)
    (CL:FORMAT T "~%~: {~@?~%~}" (FORMAT-ERROR-ARGS CONDITION]))
```

```
(DEFMACRO MAKE-DISPATCH-VECTOR (&BODY ENTRIES)
```

:: Hairy dispatch-table initialization macro. Takes a list of two-element lists (<character> <function-object>) and returns a vector char-code-limit
:: elements in length, where the lth element is the function associated with the character with char-code l. If the character is case-convertible, it must
:: be given in only one case however, an entry in the vector will be made for both.

```
(LET ((ENTRIES (CL:MAPCAN #'[CL:LAMBDA (X)
  (LET [(LOWER (CL:CHAR-DOWNCASE (CAR X)))
    (UPPER (CL:CHAR-UPCASE (CAR X))
    (CL:IF (CL:CHAR= LOWER UPPER)
      (LIST X)
      (LIST (CONS UPPER (CDR X))
        (CONS LOWER (CDR X))))])
  ENTRIES)))
  (CL:DO ([ENTRIES (SORT ENTRIES #'(CL:LAMBDA (X Y)
    (CL:CHAR< (CAR X)
      (CAR Y)
      (CHARIDX 0 (CL:1+ CHARIDX))
      (COMTAB NIL (CONS (CL:IF ENTRIES
        (CL:IF (= (CL:CHAR-CODE (CAAR ENTRIES))
          CHARIDX)
          (CADR (pop ENTRIES))
          NIL)
        NIL)
      COMTAB)))
    [(= CHARIDX 256)
      (CL:IF ENTRIES (CL:ERROR "Garbage in dispatch vector - ~S" ENTRIES))
      `(CL:MAKE-ARRAY '(256)
        :ELEMENT-TYPE T :INITIAL-CONTENTS ', (CL:NREVERSE COMTAB))])
```

```
(CL:DEFUN SCALE-EXPONENT (X)
  (SCALE-EXPT-AUX X 0.0 1.0 10.0 0.1 (CONSTANT (CL:LOG 2.0 10.0))))
```

```
(CL:DEFUN SCALE-EXPT-AUX (X ZERO ONE TEN ONE-TENTH LOG10-OF-2)
  (CL:MULTIPLE-VALUE-BIND (SIG EXPONENT)
    (CL:DECODE-FLOAT X)
    (DECLARE (IGNORE SIG))
    (CL:IF (= X ZERO)
      (CL:VALUES ZERO 1)
      [LET* [(E (ROUND (CL:* EXPONENT LOG10-OF-2)))
              (NEWX (CL:IF (MINUSP E)
                            (CL:* X TEN (CL:EXPT TEN (- -1 E)))
                            (/ X TEN (CL:EXPT TEN (CL:1- E)))))]
        (CL:DO ((D TEN (CL:* D TEN))
                (Y NEWX (/ NEWX D))
                (E E (CL:1+ E)))
          [(< Y ONE)
            (CL:DO ((M TEN (CL:* M TEN))
                    (Z Y (CL:* Z M))
                    (E E (CL:1- E))
                    (>= Z ONE-TENTH)
                    (CL:VALUES (/ X (CL:EXPT 10 E)
                                E))))]))])
```

```
(CL:DEFUN FORMAT-ERROR (COMPLAINT &REST FORMAT-ARGS)
  [CL:ERROR 'FORMAT-ERROR :ARGS (LIST (LIST "~?~%~S~%~V@T^" COMPLAINT FORMAT-ARGS *FORMAT-CONTROL-STRING*
                                         (CL:1+ *FORMAT-INDEX*))
```

```
(CL:DEFVAR *DIGIT-STRING* (CL:MAKE-ARRAY 50 :ELEMENT-TYPE 'CL:STRING-CHAR :FILL-POINTER 0 :ADJUSTABLE T))
```

```
(CL:DEFCONSTANT *DIGITS* "0123456789")
```

```
(CL:DEFUN FLONUM-TO-STRING (X &OPTIONAL WIDTH DECPACES SCALE FMIN)
```

;; Returns FIVE values: a string of digits with one decimal point, the string's length, T if the point is at the front, T if the point is at the end, the index
 ;; of the point in the string

```
(CL:IF (ZEROP X)
  (CL:VALUES "." 1 T T)
  [LET* ((REALDP (COND
                 (DECPACES (CL:IF FMIN
                                   (MAX DECPACES FMIN)
                                   DECPACES))
                 (FMIN)))
        (ROUND (COND
                 [REALDP
                   (MIN 9 (+ (DIGITSBDP X)
                             REALDP
                             (OR SCALE 0)
                             (WIDTH (MAX 1 (MIN 9 (CL:1- WIDTH)
                                                MANTSTR INTEXP)
                                     (CL:MULTIPLE-VALUE-SETQ (MANTSTR INTEXP)
                                                                (FLTSTR X ROUND))
                                     (CL:IF SCALE (CL:INCF INTEXP SCALE)))
                   ; OK, now copy the digit string into *digit-string* with the decimal point set appropriately
                   (CL:MACROLET [(STRPUT (C)
                                         ^ (CL:VECTOR-PUSH-EXTEND ,C *DIGIT-STRING*)
                                         (LET* ((DIGITS (CL:LENGTH MANTSTR))
                                               (INDEX -1)
                                               (POINTPLACE (+ DIGITS INTEXP))
                                               (DECPNT))
                                           ;; MANTSTR may have more digits than necessary; prune off its zeros. Doing this will lose if X is zero.
                                           (IF (NOT (ZEROP X))
                                             THEN (WHILE (AND (CL:PLUSP DIGITS)
                                                                (CL:CHAR= (CL:CHAR MANTSTR (CL:1- DIGITS))
                                                                #\0))
                                               DO (CL:DECF DIGITS)
                                                  (CL:INCF INTEXP)))
                                           (CL:SETF (CL:FILL-POINTER *DIGIT-STRING*
                                                                    0)
                                                  (COND
                                                    ((NOT (CL:PLUSP POINTPLACE)) ; <.digits>
                                                     (STRPUT #\.)
                                                     (CL:DOTIMES (I (- POINTPLACE))
                                                       (STRPUT #\0))
                                                     (CL:DOTIMES (I DIGITS)
                                                       (STRPUT (CL:CHAR MANTSTR I))))
                                                    (SETQ DECPNT 0)
                                                    (MINUSP INTEXP) ; <digits>.<digits>
                                                     (CL:1+ DECPNT)
                                                     (CL:1+ INTEXP))
                                                  (CL:1+ DECPNT)
                                                  (CL:1+ INTEXP))
                                         (CL:1+ DECPNT)
                                         (CL:1+ INTEXP))
                   (CL:1+ DECPNT)
                   (CL:1+ INTEXP))
  (CL:1+ DECPNT)
  (CL:1+ INTEXP))
```

```

(CL:DOTIMES (I POINTPLACE)
  (STRPUT (CL:CHAR MANTSTR (CL:INCF INDEX))))
(STRPUT #\.)
(CL:DOTIMES (I (- INTEXP))
  (STRPUT (CL:CHAR MANTSTR (CL:INCF INDEX))))
(SETQ DECPNT (+ DIGITS INTEXP))
(T ; <digits>00.
  (CL:DOTIMES (I DIGITS)
    (STRPUT (CL:CHAR MANTSTR I)))
  (CL:DOTIMES (I INTEXP)
    (STRPUT #\0))
  (STRPUT #\.)
  (SETQ DECPNT (+ DIGITS INTEXP)
    (SETQ DIGITS (CL:1- (CL:LENGTH *DIGIT-STRING*)))
    (IF DECPLACES
      THEN ;; Need extra 0s to get enough decimal places
        (CL:DOTIMES (I (- DECPLACES (- DIGITS DECPNT)))
          (STRPUT #\0)
          (CL:INCF DIGITS)))
        (CL:VALUES *DIGIT-STRING* (CL:1+ DIGITS)
          (= DECPNT 0)
          (= DECPNT DIGITS)
          DECPNT]))

```

(DEFMACRO **FORMAT-WITH-CONTROL-STRING** (CONTROL-STRING &BODY FORMS)

;; This macro establishes the correct environment for processing an indirect control string. CONTROL-STRING is the string to process, and FORMS are the forms to do the processing. They invariably will involve a call to SUB-FORMAT. CONTROL-STRING is guaranteed to be evaluated exactly once.

```

`[LET ((STRING ,CONTROL-STRING)
  (CONDITION-CASE (LET ((*FORMAT-CONTROL-STRING* STRING)
    (*FORMAT-LENGTH* (CL:LENGTH STRING))
    (*FORMAT-INDEX* 0))
    ,@FORMS)
  (FORMAT-ERROR (C)
    (CL:ERROR 'FORMAT-ERROR :ARGS (CONS (LIST "While processing indirect control
      string~%%~S~%%~V@T^" *FORMAT-CONTROL-STRING*
      (CL:1+ *FORMAT-INDEX*))
      (FORMAT-ERROR-ARGS C]

```

(DEFMACRO **FORMAT-STRINGIFY-OUTPUT** (&BODY FORMS)

;; This macro collects output to the standard output stream in a string. It used to try to avoid consing new string streams if possible.

```

` (CL:WITH-OUTPUT-TO-STRING (*STANDARD-OUTPUT*
  ,@FORMS))

```

(DEFMACRO **POP-FORMAT-ARG** ())

;; Pops an argument from the current argument list. This is either the list of arguments given to the top-level call to FORMAT, or the argument list for the current iteration in a ~{~} construct. An error is signalled if the argument list is empty. *

```

` (CL:IF *FORMAT-ARGUMENTS*
  (CL:POP *FORMAT-ARGUMENTS*)
  (FORMAT-ERROR "Missing argument"))

```

(DEFMACRO **WITH-FORMAT-PARAMETERS** (PARMVAR PARMDEFS &BODY FORMS)

;; This macro decomposes the argument list returned by PARSE-FORMAT-OPERATION. PARMVAR is the list of parameters. PARMDEFS is a list of lists of the form (<var> <default>). The FORMS are evaluated in an environment where each <var> is bound to either the value of the parameter supplied in the parameter list, or to its <default> value if the parameter was omitted or explicitly defaulted.

```

` (LET , [FOR PARMDEF IN PARMDEFS COLLECT ` (, (CL:FIRST PARMDEF)
  (OR (CL:IF ,PARMVAR
    (POP ,PARMVAR)
    , (CL:SECOND PARMDEF]
  (CL:WHEN ,PARMVAR (FORMAT-ERROR "Too many parameters"))
  ,@FORMS))

```

(DEFMACRO **NEXTCHAR** ())

;; Gets the next character from the current control string. It is an error if there is none. Leave *format-index* pointing to the character returned. *

```

` (CL:IF (< (CL:INCF *FORMAT-INDEX*)
  *FORMAT-LENGTH*)
  (CL:CHAR *FORMAT-CONTROL-STRING* *FORMAT-INDEX*)
  (FORMAT-ERROR "Syntax error"))

```

(DEFMACRO **FORMAT-PEEK** ())

;; Returns the current character, i.e. the one pointed to by *format-index*.

```

` (CL:CHAR *FORMAT-CONTROL-STRING* *FORMAT-INDEX*)

```

(DEFMACRO **FORMAT-FIND-CHAR** (CHAR START END)

;; Returns the index of the first occurrence of the specified character between indices START (inclusive) and END (exclusive) in the control string.
 ` (CL:POSITION ,CHAR *FORMAT-CONTROL-STRING* :START ,START :END ,END :TEST 'CL:CHAR=)

(CL:DEFUN **FORMAT-GET-PARAMETER** ()

;; Attempts to parse a parameter, starting at the current index. Returns the value of the parameter, or NIL if none is found. On exit, *format-index* points to the first character which is not a part of the recognized parameter.

```
(LET [(NUMSIGN (CASE (FORMAT-PEEK)
    (#\+
      (NEXTCHAR)
      NIL)
    (#\-
      (NEXTCHAR)
      T)
    (T NIL))])
  (CASE (FORMAT-PEEK)
    (#\#
      (NEXTCHAR)
      (CL:LENGTH *FORMAT-ARGUMENTS*))
    ((#\V #\v) (PROG1 (POP-FORMAT-ARG)
      (NEXTCHAR)))
    (#\' (PROG1 (NEXTCHAR)
      (NEXTCHAR)))
    ((#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9) (CL:DO* [(CL:NUMBER (CL:DIGIT-CHAR-P (FORMAT-PEEK))
      (+ (CL:* 10 CL:NUMBER)
        (CL:DIGIT-CHAR-P (FORMAT-PEEK))]
      (NOT (CL:DIGIT-CHAR-P (NEXTCHAR)))
      (CL:IF NUMSIGN
        (- CL:NUMBER)
        CL:NUMBER))))
      (T NIL))))
```

(CL:DEFUN **PARSE-FORMAT-OPERATION** ()

(* amd " 1-May-86 14:33")

;; Parses a format directive, including flags and parameters. On entry, *format-index* should point to the '~' preceding the command. On exit, *format-index* points to the command character itself. Returns the list of parameters, the ':' flag, the '@' flag, and the command character as multiple values. Explicitly defaulted parameters appear in the list of parameters as NIL. Omitted parameters are simply not included in the list at all. *

```
(LET ((CH (NEXTCHAR))
  PARMS COLON ATSIGN)
  ;; First get the parameters
  (SETQ PARMS (CL:IF (OR (CL:DIGIT-CHAR-P CH)
    (CL:MEMBER CH '(#\, #\# #\V #\v #\'')
      :TEST
      (FUNCTION CL:CHAR=)))
    (CL:DO ((PARMS (LIST (FORMAT-GET-PARAMETER))
      (CONS (FORMAT-GET-PARAMETER)
        PARMS)))
      ((CL:CHAR/= (FORMAT-PEEK)
        #\,)
      (CL:NREVERSE PARMS))
      (NEXTCHAR))
    'NIL))
  ;; Then check for : and @ (not necessarily in that order)
  [CL:LOOP (CASE (FORMAT-PEEK)
    (#\: (CL:IF COLON
      (RETURN NIL)
      (SETQ COLON (NEXTCHAR))))
    (#\@ (CL:IF ATSIGN
      (RETURN NIL)
      (SETQ ATSIGN (NEXTCHAR))))
    (T (RETURN NIL)))]
  (CL:VALUES PARMS COLON ATSIGN (FORMAT-PEEK)))
```

(CL:DEFUN **FORMAT-FIND-COMMAND** (COMMAND-LIST)

;; Starting at the current value of *format-index*, finds the first occurrence of one of the specified directives. Embedded constructs, i.e. those inside ~ (~) %, ~[~], ~{~}, or ~<~>, are ignored. And error is signalled if no satisfactory command is found. Otherwise, the following are returned as multiple values: The value of *format-index* at the start of the search The index of the '~' character preceding the command The parameter list of the command The ':' flag The '@' flag The command character Implementation note: The present implementation is not particularly careful with storage allocation. It would be a good idea to have a separate function for skipping embedded constructs which did not bother to cons parameter lists and then throw them away. We go to some trouble here to use POSITION for most of the searching.

```
(LET ((START *FORMAT-INDEX*))
  (CL:DO ((PLACE START *FORMAT-INDEX*)
    (TILDE (FORMAT-FIND-CHAR #\~ START *FORMAT-LENGTH*)
      (FORMAT-FIND-CHAR #\~ PLACE *FORMAT-LENGTH*)))
    ((NOT TILDE)
      (FORMAT-ERROR "Expecting one of ~S" COMMAND-LIST))
    (SETQ *FORMAT-INDEX* TILDE)
```



```

(CL:CATCH 'FORMAT-ESCAPE
  (SUB-FORMAT PREV TILDE)
  (SETQ SUB-ESCAPE NIL))
(CL:SETQ ESCAPE SUB-ESCAPE)))
[WRITE-STRING* (COND
  (AND ATSIGN COLON)
  (CL:NSTRING-UPCASE STRING))
(COLON (CL:NSTRING-CAPITALIZE STRING))
  ATSIGN ; Capitalize the first word only
  (LET ((STRLEN (CL:LENGTH STRING)))
    (CL:NSTRING-DOWNCASE STRING)
    (CL:DO ((I 0 (CL:1+ I))
      ((OR (<= STRLEN I)
        (CL:ALPHA-CHAR-P (CL:CHAR STRING I)))
        (CL:SETF (CL:CHAR STRING I)
          (CL:CHAR-UPCASE (CL:CHAR STRING I)))
        STRING))]
      (T (CL:NSTRING-DOWNCASE STRING)
        (AND ESCAPE (CL:THROW ESCAPE NIL))))))

```

```

(CL:DEFUN FORMAT-ESCAPE (COLON ATSIGN PARMS)
  ;; Up and Out (Escape) ~^
  (CL:WHEN ATSIGN (FORMAT-ERROR "FORMAT command ~::~[~::~~]@^ is undefined" COLON))
  (CL:WHEN (CL:IF (CL:FIRST PARMS)
    (CL:IF (CL:SECOND PARMS)
      (CL:IF (CL:THIRD PARMS)
        (CL:TYPECASE (CL:SECOND PARMS)
          (INTEGER (<= (CL:FIRST PARMS)
            (CL:SECOND PARMS)
            (CL:THIRD PARMS)))
          (CL:CHARACTER (CL:CHAR< (CL:FIRST PARMS)
            (CL:SECOND PARMS)
            (CL:THIRD PARMS)))
          (T NIL))
        (EQUAL (CL:FIRST PARMS)
          (CL:SECOND PARMS)))
        (ZEROP (CL:FIRST PARMS)))
        (NOT *FORMAT-ARGUMENTS*))
      (CL:THROW (CL:IF COLON
        'FORMAT-COLON-ESCAPE
        'FORMAT-ESCAPE)
        NIL)))

```

```

(CL:DEFUN FORMAT-SEMICOLON-ERROR (COLON ATSIGN PARMS)
  (DECLARE (IGNORE COLON ATSIGN PARMS))
  (FORMAT-ERROR "Unexpected semicolon (probably a missing ~ somewhere)."))

```

```

(CL:DEFUN FORMAT-UNTAGGED-CONDITION ()
  ;; ~[
  [LET ((TEST (POP-FORMAT-ARG)))
    (CL:UNLESS (CL:INTEGERP TEST)
      (FORMAT-ERROR "Argument to ~::~[ must be integer - ~S" TEST))
    (CL:DO ((CL:COUNT 0 (CL:1+ CL:COUNT))
      [(= CL:COUNT TEST)
        (CL:MULTIPLE-VALUE-BIND (PREV TILDE PARMS COLON ATSIGN CMD)
          (FORMAT-FIND-COMMAND '#\; #\))
          (DECLARE (IGNORE COLON))
          (CL:WHEN ATSIGN (FORMAT-ERROR "Atsign flag not allowed"))
          (CL:WHEN PARMS (FORMAT-ERROR "No parameters allowed"))
          (SUB-FORMAT PREV TILDE)
          (CL:UNLESS (CL:CHAR= CMD #\))
            (FORMAT-FIND-COMMAND '#\)))]))
      (CL:MULTIPLE-VALUE-BIND (PREV TILDE PARMS COLON ATSIGN CMD)
        (FORMAT-FIND-COMMAND '#\; #\))
        (DECLARE (IGNORE PREV TILDE))
        (CL:WHEN ATSIGN (FORMAT-ERROR "Atsign flag not allowed"))
        (CL:WHEN PARMS (FORMAT-ERROR "Parameters not allowed"))
        (CL:WHEN (CL:CHAR= CMD #\))
          (RETURN))
        (CL:WHEN COLON
          (NEXTCHAR)
          (CL:MULTIPLE-VALUE-BIND (PREV TILDE PARMS COLON ATSIGN CMD)
            (FORMAT-FIND-COMMAND '#\; #\))
            (DECLARE (IGNORE PARMS COLON ATSIGN))
            (SUB-FORMAT PREV TILDE)
            (CL:UNLESS (CL:CHAR= CMD #\))
              (FORMAT-FIND-COMMAND '#\)))]))
          (RETURN))
        (NEXTCHAR)))]

```

```

(CL:DEFUN FORMAT-FUNNY-CONDITION ()

```

```

;; ~@[
(CL: MULTIPLE-VALUE-BIND (PREV TILDE PARMS COLON ATSIGN)
  (FORMAT-FIND-COMMAND ' (#\)))
(CL: WHEN (OR COLON ATSIGN PARMS)
  (FORMAT-ERROR "Flags or arguments not allowed"))
(CL: IF *FORMAT-ARGUMENTS*
  (CL: IF (CAR *FORMAT-ARGUMENTS*)
    (SUB-FORMAT PREV TILDE)
    (CL: POP *FORMAT-ARGUMENTS*))
  (FORMAT-ERROR "Missing argument"))))

```

(CL:DEFUN **FORMAT-BOOLEAN-CONDITION** ())

```

;; ~:[
(CL: MULTIPLE-VALUE-BIND (PREV TILDE PARMS COLON ATSIGN)
  (FORMAT-FIND-COMMAND ' (#\;)))
(CL: WHEN (OR PARMS COLON ATSIGN)
  (FORMAT-ERROR "Flags or parameters not allowed"))
(NEXTCHAR)
(CL: IF (POP-FORMAT-ARG)
  (CL: MULTIPLE-VALUE-BIND (PREV TILDE PARMS COLON ATSIGN)
    (FORMAT-FIND-COMMAND ' (#\]))
  (CL: WHEN (OR COLON ATSIGN PARMS)
    (FORMAT-ERROR "Flags or parameters not allowed"))
  (SUB-FORMAT PREV TILDE))
[PROGN (SUB-FORMAT PREV TILDE)
  (FORMAT-FIND-COMMAND ' (#\]))))

```

(CL:DEFUN **FORMAT-CONDITION** (COLON ATSIGN PARMS)

```

(CL: WHEN PARMS
  (CL: PUSH (POP PARMS)
    *FORMAT-ARGUMENTS*))
(CL: UNLESS (NULL PARMS)
  (FORMAT-ERROR "Too many parameters to ~["]))
(NEXTCHAR)
(COND
  (COLON (CL: WHEN ATSIGN (FORMAT-ERROR "~~:@[ undefined")
    (FORMAT-BOOLEAN-CONDITION)))
  (ATSIGN (FORMAT-FUNNY-CONDITION))
  (T (FORMAT-UNTAGGED-CONDITION))))

```

(CL:DEFUN **FORMAT-ITERATION** (COLON ATSIGN PARMS)

```

;; Iteration ~{ ... ~}
[WITH-FORMAT-PARAMETERS PARMS ((MAX-ITER -1))
  (NEXTCHAR)
  (CL: MULTIPLE-VALUE-BIND (PREV TILDE END-PARMS END-COLON END-ATSIGN)
    (FORMAT-FIND-COMMAND ' (#\}))
  (CL: WHEN (OR END-ATSIGN END-PARMS)
    (FORMAT-ERROR "Illegal terminator for ~{"))
  (CL: IF (= PREV TILDE)
    (LET ((STRING (POP-FORMAT-ARG)))
      ;; Use an argument as the control string if ~{~} is empty
      (CL: UNLESS (CL: STRINGP STRING)
        (FORMAT-ERROR "Control string is not a string"))
      (FORMAT-WITH-CONTROL-STRING STRING (FORMAT-DO-ITERATION 0 *FORMAT-LENGTH* MAX-ITER COLON
        ATSIGN END-COLON))))
    (FORMAT-DO-ITERATION PREV TILDE MAX-ITER COLON ATSIGN END-COLON)))]

```

(CL:DEFUN **FORMAT-DO-ITERATION** (START END MAX-ITER COLON ATSIGN AT-LEAST-ONCE-P)

;; The two catch tags FORMAT-ESCAPE and FORMAT-COLON-ESCAPE are needed here to correctly implement ~^ and ~:^. The former aborts only the current iteration, but the latter aborts the entire iteration process. *

```

(CL: CATCH 'FORMAT-COLON-ESCAPE
  (CL: CATCH 'FORMAT-ESCAPE
    (CL: IF ATSIGN
      (CL: DO ((CL: COUNT 0 (CL: 1+ CL: COUNT)))
        [(OR (= CL: COUNT MAX-ITER)
          (AND (NULL *FORMAT-ARGUMENTS*)
            (CL: IF (= CL: COUNT 0)
              (NOT AT-LEAST-ONCE-P)
              T)])
          (CL: CATCH 'FORMAT-ESCAPE
            (CL: IF COLON
              (LET* ((*ORIGINAL-ARGUMENTS* (POP-FORMAT-ARG))
                (*FORMAT-ARGUMENTS* *ORIGINAL-ARGUMENTS*))
                (CL: UNLESS (CL: LISTP *FORMAT-ARGUMENTS*)
                  (FORMAT-ERROR "Argument must be a list"))
                (SUB-FORMAT START END))
              (SUB-FORMAT START END))))
            [LET* ((*ORIGINAL-ARGUMENTS* (POP-FORMAT-ARG))

```

```

(*FORMAT-ARGUMENTS* *ORIGINAL-ARGUMENTS*)
(CL:UNLESS (CL:LISTP *FORMAT-ARGUMENTS*)
  (FORMAT-ERROR "Argument must be a list"))
(CL:DO ((CL:COUNT 0 (CL:1+ CL:COUNT))
  [(OR (= CL:COUNT MAX-ITER)
    (AND (NULL *FORMAT-ARGUMENTS*)
      (CL:IF (= CL:COUNT 0)
        (NOT AT-LEAST-ONCE-P)
        T)])
  (CL:CATCH 'FORMAT-ESCAPE
    (CL:IF COLON
      (LET* ((*ORIGINAL-ARGUMENTS* (POP-FORMAT-ARG))
        (*FORMAT-ARGUMENTS* *ORIGINAL-ARGUMENTS*)
        (CL:UNLESS (CL:LISTP *FORMAT-ARGUMENTS*)
          (FORMAT-ERROR "Argument must be a list of lists")))
        (SUB-FORMAT START END))
      (SUB-FORMAT START END)))))))))

```

(CL:DEFUN **FORMAT-GET-TRAILING-SEGMENTS** ())

:: Parses a list of clauses delimited by ~ and terminated by ~>. Recursively invoke SUB-FORMAT to process them, and return a list of the results, :: the length of this list, and the total number of characters in the strings composing the list.

(NEXTCHAR)

```

(CL:MULTIPLE-VALUE-BIND (PREV TILDE COLON ATSIGN PARMS CMD)
  (FORMAT-FIND-COMMAND '#\; #\>))
(CL:WHEN COLON (FORMAT-ERROR "~::~; allowed only after first segment in ~<>"))
(CL:WHEN (OR ATSIGN PARMS)
  (FORMAT-ERROR "Flags and parameters not allowed"))
(LET [(STR (CL:CATCH 'FORMAT-ESCAPE
  (FORMAT-STRINGIFY-OUTPUT (SUB-FORMAT PREV TILDE))))]
  (CL:IF STR
    (CL:IF (CL:CHAR= CMD #\;)
      (CL:MULTIPLE-VALUE-BIND (SEGMENTS NUMSEGS NUMCHARS)
        (FORMAT-GET-TRAILING-SEGMENTS)
        (CL:VALUES (CONS STR SEGMENTS)
          (CL:1+ NUMSEGS)
          (+ NUMCHARS (CL:LENGTH STR))))
      (CL:VALUES (LIST STR)
        1
        (CL:LENGTH STR)))
    (CL:VALUES NIL 0 0))))))

```

(CL:DEFUN **FORMAT-GET-SEGMENTS** ())

:: Gets the first segment, which is treated specially. Call FORMAT-GET-TRAILING-SEGMENTS to get the rest.

```

(CL:MULTIPLE-VALUE-BIND (PREV TILDE PARMS COLON ATSIGN CMD)
  (FORMAT-FIND-COMMAND '#\; #\>))
(CL:WHEN ATSIGN (FORMAT-ERROR "Atsign flag not allowed"))
[LET [(FIRST-SEG (FORMAT-STRINGIFY-OUTPUT (SUB-FORMAT PREV TILDE)
  (CL:IF (CL:CHAR= CMD #\;)
    (CL:MULTIPLE-VALUE-BIND (SEGMENTS NUMSEGS NUMCHARS)
      (FORMAT-GET-TRAILING-SEGMENTS)
      (CL:IF COLON
        (CL:VALUES FIRST-SEG PARMS SEGMENTS NUMSEGS NUMCHARS)
        (CL:VALUES NIL NIL (CONS FIRST-SEG SEGMENTS)
          (CL:1+ NUMSEGS)
          (+ (CL:LENGTH FIRST-SEG)
            NUMCHARS))))
      (CL:VALUES NIL NIL (LIST FIRST-SEG)
        1
        (CL:LENGTH FIRST-SEG))))))]

```

(CL:DEFUN **MAKE-PAD-SEGS** (SPACES PADDINGS)

:: Given the total number of SPACES needed for padding, and the number of padding segments needed (PADDINGS), returns a list of such :: segments. We try to allocate the spaces equally to each segment. When this is not possible, we allocate the left-over spaces randomly, to :: improve the appearance of many successive lines of justified text.

:: Query: Is this right? Perhaps consistency might be better for the kind of applications ~<~> is used for.

```

(CL:DO* ([EXTRA-SPACE NIL (AND (CL:PLUSP EXTRA-SPACES)
  (< (RAND 0 (FLOAT 1))
    (/ SEGS EXTRA-SPACES]
  (RESULT NIL (CONS (CL:IF EXTRA-SPACE
    (CL:1+ MIN-SPACE)
    MIN-SPACE)
    RESULT))
  (MIN-SPACE (CL:TRUNCATE SPACES PADDINGS))
  (EXTRA-SPACES (- SPACES (CL:* PADDINGS MIN-SPACE))
    (CL:IF EXTRA-SPACE
      (CL:1- EXTRA-SPACES)
      EXTRA-SPACES))
  (SEGS PADDINGS (CL:1- SEGS)))
  ((ZEROP SEGS)
  RESULT)))

```


(CL:DEFUN **FORMAT-ROUND-COLUMNS** (WIDTH MINCOL COLINC)

:: Determine the actual width to be used for a field requiring WIDTH characters according to the following rule: If WIDTH is less than or equal to MINCOL, use WIDTH as the actual width. Otherwise, round up to MINCOL + k * COLINC for the smallest possible positive integer k.

(CL:IF (> WIDTH MINCOL)
WIDTH
(+ WIDTH (CL:* COLINC (CL:CEILING (- MINCOL WIDTH)
COLINC))))))

(CL:DEFUN **FORMAT-JUSTIFICATION** (COLON ATSIGN PARMS)

[WITH-FORMAT-PARAMETERS PARMS ((MINCOL 0)
(COLINC 1)
(MINPAD 0)
(PADCHAR #\Space))
(CL:UNLESS (AND (CL:INTEGERP MINCOL)
(NOT (MINUSP MINCOL)))
(FORMAT-ERROR "Mincol must be a non-negative integer - ~S" MINCOL))
(CL:UNLESS (AND (CL:INTEGERP COLINC)
(CL:PLUSP COLINC))
(FORMAT-ERROR "Colinc must be a positive integer - ~S" COLINC))
(CL:UNLESS (AND (CL:INTEGERP MINPAD)
(NOT (MINUSP MINPAD)))
(FORMAT-ERROR "Minpad must be a non-negative integer - ~S" MINPAD))
(CL:UNLESS (CL:CHARACTERP PADCHAR)
(FORMAT-ERROR "Padchar must be a character - ~S" PADCHAR))
(NEXTCHAR)

(CL:MULTIPLE-VALUE-BIND (SPECIAL-ARG SPECIAL-PARMS SEGMENTS NUMSEGS NUMCHARS)
(FORMAT-GET-SEGMENTS)

[LET* ([PADSEGS (CL:IF (= NUMSEGS 1)
(CL:IF (AND COLON ATSIGN)
2
1)
(+ (CL:IF COLON
1
0)
(CL:1- NUMSEGS)
(CL:IF ATSIGN
1
0)))]
(WIDTH (FORMAT-ROUND-COLUMNS (+ NUMCHARS (CL:* MINPAD PADSEGS)
MINCOL COLINC))
(SPACES (MAKE-PAD-SEGS (- WIDTH NUMCHARS)
PADSEGS)))

(CL:IF (= NUMSEGS 1)
[COND
((AND ATSIGN (NOT COLON))
(CL:PUSH '0 SPACES))
((OR (AND COLON (NOT ATSIGN))
(AND (NOT ATSIGN)
(NOT COLON)))
(NCONC SPACES ' (0))
(PROGN (CL:IF (OR (AND COLON (NOT ATSIGN))
(AND (NOT ATSIGN)
(NOT COLON)))
(NCONC SPACES ' (0))
(CL:IF (OR (AND ATSIGN (NOT COLON))
(AND (NOT ATSIGN)
(NOT COLON)))
(CL:PUSH '0 SPACES))))
(CL:WHEN SPECIAL-ARG
[WITH-FORMAT-PARAMETERS SPECIAL-PARMS ((SPARE 0)
(LINEL (OR (LINELENGTH)
72)))

(LET ((POS (OR (CHARPOS *STANDARD-OUTPUT*)
0)))
(CL:WHEN (> (+ POS WIDTH SPARE)
LINEL)
(WRITE-STRING* SPECIAL-ARG]))

(CL:DO ((SEGS SEGMENTS (CDR SEGS))
(SPCS SPACES (CDR SPCS)))
((NULL SEGS)
(CL:DOTIMES (I (CAR SPCS))
(CL:WRITE-CHAR PADCHAR)))
(CL:DOTIMES (I (CAR SPCS))
(CL:WRITE-CHAR PADCHAR))
(WRITE-STRING* (CAR SEGS))))])

(CL:DEFUN **FORMAT-TERPRI** (COLON ATSIGN PARMS)

:: Newline ~&

(CL:WHEN (OR COLON ATSIGN)
(FORMAT-ERROR "Flags not allowed"))
(WITH-FORMAT-PARAMETERS PARMS ((REPEAT-COUNT 1))
(CL:DOTIMES (I REPEAT-COUNT)

(TERPRI *STANDARD-OUTPUT*))

(CL:DEFUN **FORMAT-FRESHLINE** (COLON ATSIGN PARMS)

;; Fresh-line ~%

(CL:WHEN (OR COLON ATSIGN)
 (FORMAT-ERROR "Flags not allowed"))
 (WITH-FORMAT-PARAMETERS PARMS ((REPEAT-COUNT 1))
 (CL:FRESH-LINE *STANDARD-OUTPUT*)
 (CL:DOTIMES (I (CL:1- REPEAT-COUNT))
 (TERPRI *STANDARD-OUTPUT*)))

(CL:DEFUN **FORMAT-PAGE** (COLON ATSIGN PARMS)

;; Page ~|

(CL:WHEN (OR COLON ATSIGN)
 (FORMAT-ERROR "Flags not allowed"))
 (WITH-FORMAT-PARAMETERS PARMS ((REPEAT-COUNT 1))
 (CL:DOTIMES (I REPEAT-COUNT)
 (CL:WRITE-CHAR #\Page)))

(CL:DEFUN **FORMAT-TILDE** (COLON ATSIGN PARMS)

;; Print a tilde ~~

(CL:WHEN (OR COLON ATSIGN)
 (FORMAT-ERROR "Flags not allowed"))
 (WITH-FORMAT-PARAMETERS PARMS ((REPEAT-COUNT 1))
 (CL:DOTIMES (I REPEAT-COUNT)
 (CL:WRITE-CHAR #\~)))

(CL:DEFUN **FORMAT-EAT-WHITESPACE** ())

;; Continue control string on next line ~<newline>

(NEXTCHAR)
 [SETQ *FORMAT-INDEX* (LET ((NEXT-NON-WHITE (CL:POSITION-IF-NOT (FUNCTION WHITESPACE-CHAR-P)
 FORMAT-CONTROL-STRING :START *FORMAT-INDEX*)))
 (CL:IF NEXT-NON-WHITE
 (CL:1- NEXT-NON-WHITE)
 (CL:LENGTH *FORMAT-CONTROL-STRING*)))]

(CL:DEFUN **FORMAT-NEWLINE** (COLON ATSIGN PARMS)

(CL:WHEN PARMS (FORMAT-ERROR "Parameters not allowed"))
 (COND
 (COLON (CL:WHEN ATSIGN (FORMAT-ERROR "~:@<newline> is undefined"))
 (ATSIGN (TERPRI *STANDARD-OUTPUT*)
 (FORMAT-EAT-WHITESPACE))
 (T (FORMAT-EAT-WHITESPACE))))

(CL:DEFUN **FORMAT-PLURAL** (COLON ATSIGN PARMS)

;; Pluralize word ~P

(CL:WHEN PARMS (FORMAT-ERROR "Parameters not allowed"))
 (CL:WHEN COLON
 ;; Back up one argument first
 [LET ((CDRS (- (CL:LENGTH *FORMAT-ORIGINAL-ARGUMENTS*)
 (CL:LENGTH *FORMAT-ARGUMENTS*)
 1)))
 (CL:IF (MINUSP CDRS)
 (FORMAT-ERROR "No previous argument")
 (SETQ *FORMAT-ARGUMENTS* (CL:NTHCDR CDRS *FORMAT-ORIGINAL-ARGUMENTS*)))])
 (CL:IF (EQL (POP-FORMAT-ARG)
 1)
 (WRITE-STRING* (CL:IF ATSIGN
 "y"
 "n")
 (WRITE-STRING* (CL:IF ATSIGN
 "ies"
 "s"))))

(CL:DEFUN **FORMAT-SKIP-ARGUMENTS** (COLON ATSIGN PARMS)

;; Skip arguments (relative goto) ~*

[WITH-FORMAT-PARAMETERS PARMS ((CL:COUNT (CL:IF ATSIGN
 0
 1))
 (COND
 (ATSIGN (CL:WHEN (OR (MINUSP CL:COUNT)
 (> CL:COUNT (CL:LENGTH *FORMAT-ORIGINAL-ARGUMENTS*))
 (FORMAT-ERROR "Illegal to go to non-existent argument"))


```

(CL:IF COLON
  "()"
  (CL:PRIN1-TO-STRING NIL))
MINCOL COLINC MINPAD PADCHAR ATSIGN)))]

```

(CL:DEFUN **FORMAT-PRINT-CHARACTER** (COLON ATSIGN PARMS)

;; Character ~C

```

[WITH-FORMAT-PARAMETERS PARMS NIL (LET ((CL:CHAR (POP-FORMAT-ARG)))
  (CL:UNLESS (CL:CHARACTERP CL:CHAR)
    (FORMAT-ERROR "Argument must be a character"))
  (COND
    ((AND (NOT COLON)
          (NOT ATSIGN))
     (CL:WRITE-CHAR CL:CHAR))
    ((AND ATSIGN (NOT COLON))
     (CL:PRIN1 CL:CHAR))
    (T (FORMAT-PRINT-NAMED-CHARACTER CL:CHAR COLON]))

```

(CL:DEFUN **FORMAT-PRINT-NAMED-CHARACTER** (CHAR LONGP)

```

[LET* ((CH (CL:CODE-CHAR (CL:CHAR-CODE CHAR)))
       (NAME (CL:CHAR-NAME CH)))
  (COND
    [NAME (WRITE-STRING* (CL:STRING-CAPITALIZE (CL:PRINC-TO-STRING NAME)
      [(<= 0 (CL:CHAR-CODE CHAR)
        31)
        (CL:WRITE-CHAR #\^)]
      (CL:WRITE-CHAR (CL:CODE-CHAR (+ 64 (CL:CHAR-CODE CHAR)
        (T (CL:WRITE-CHAR CH)]

```

(CL:DEFUN **FORMAT-ADD-COMMAS** (STRING COMMACHAR COMMA-INTERVAL)

;; Insert commas after every COMMA-INTERVALth digit, scanning from right to left. Signs don't count in the final length.

```

(CL:DO* ((LENGTH (CL:LENGTH (THE STRING STRING)))
        (NEW-LENGTH (+ LENGTH (CL:FLOOR (- LENGTH (CL:IF (OR (EQL (CL:CHAR STRING 0)
          #\+)
          (EQL (CL:CHAR STRING 0)
            #\-)
            2
          1))
          COMMA-INTERVAL)))
  (NEW-STRING (CL:MAKE-STRING NEW-LENGTH :INITIAL-ELEMENT COMMACHAR)
    (CL:REPLACE (THE STRING NEW-STRING)
      (THE STRING STRING)
      :START1
      (MAX 0 (- NEW-POS COMMA-INTERVAL))
      :END1 NEW-POS :START2 (MAX 0 (- POS COMMA-INTERVAL))
      :END2 POS)
    (POS LENGTH (- POS COMMA-INTERVAL))
    (NEW-POS NEW-LENGTH (- NEW-POS COMMA-INTERVAL 1)))
  (NOT (CL:PLUSP POS))
  ;; If there was a sign, put it back now
  (CL:IF (OR (EQL (CL:CHAR STRING 0)
    #\+)
    (EQL (CL:CHAR STRING 0)
      #\-)
    (CL:SETF (CL:CHAR NEW-STRING 0)
      (CL:CHAR STRING 0)))
    NEW-STRING))

```

(CL:DEFUN **FORMAT-WRITE-FIELD** (STRING MINCOL COLINC MINPAD PADCHAR PADLEFT)

;; Output a string in a field at MINCOL wide, padding with PADCHAR. Pads on the left if PADLEFT is true, else on the right. If the length of the string plus the minimum permissible padding, MINPAD, is greater than MINCOL, the actual field size is rounded up to MINCOL + k * COLINC for the smallest possible positive integer k.

```

(CL:UNLESS (AND (CL:INTEGERP MINCOL)
  (NOT (MINUSP MINCOL)))
  (FORMAT-ERROR "Mincol must be a non-negative integer - ~S" MINCOL))
(CL:UNLESS (AND (CL:INTEGERP COLINC)
  (CL:PLUSP COLINC))
  (FORMAT-ERROR "Colinc must be a positive integer - ~S" COLINC))
(CL:UNLESS (AND (CL:INTEGERP MINPAD)
  (NOT (MINUSP MINPAD)))
  (FORMAT-ERROR "Minpad must be a non-negative integer - ~S" MINPAD))
(CL:UNLESS (CL:CHARACTERP PADCHAR)
  (FORMAT-ERROR "Padchar must be a character - ~S" PADCHAR))
[LET* ((STRLEN (CL:LENGTH (THE STRING STRING)))
       (WIDTH (FORMAT-ROUND-COLUMNS (+ STRLEN MINPAD)
        MINCOL COLINC)))
  (COND
    (PADLEFT (CL:DOTIMES (I (- WIDTH STRLEN))
      (CL:WRITE-CHAR PADCHAR))

```

```
(WRITE-STRING* STRING))
(T (WRITE-STRING* STRING)
  (CL:DOTIMES (I (- WIDTH STRLEN))
    (CL:WRITE-CHAR PADCHAR)))
```

```
(CL:DEFUN FORMAT-PRINT-NUMBER (NUMBER RADIX PRINT-COMMAS-P PRINT-SIGN-P PARMS)
```

;; This functions does most of the work for the numeric printing directives. The parameters are interpreted as defined for ~D.

```
[WITH-FORMAT-PARAMETERS PARMS ((MINCOL 0)
  (PADCHAR #\Space)
  (COMMACHAR #\,)
  (COMMA-INTERVAL 3)) ; comma-interval is an XCL extension.
(LET* ((*PRINT-BASE* RADIX)
  (*PRINT-RADIX* NIL)
  (TEXT (CL:PRINC-TO-STRING NUMBER)))
  (CL:IF (CL:INTEGERP NUMBER)
    (PROGN
      ;; colinc = 1, minpad = 0, padleft = t
      (FORMAT-WRITE-FIELD (CL:IF (AND (CL:PLUSP NUMBER)
        PRINT-SIGN-P)
        (CL:IF PRINT-COMMAS-P
          (CL:CONCATENATE 'STRING "+" (FORMAT-ADD-COMMAS TEXT
            COMMACHAR
            COMMA-INTERVAL))
          (CL:CONCATENATE 'STRING "+" TEXT))
        (CL:IF PRINT-COMMAS-P
          (FORMAT-ADD-COMMAS TEXT COMMACHAR COMMA-INTERVAL)
          TEXT))
        MINCOL 1 0 PADCHAR T))
      (WRITE-STRING* TEXT)))]
```

```
(CL:DEFUN FORMAT-PRINT-SMALL-CARDINAL (N)
```

```
(CL:MULTIPLE-VALUE-BIND (HUNDREDS REM)
  (CL:TRUNCATE N 100)
  (CL:WHEN (CL:PLUSP HUNDREDS)
    (WRITE-STRING* (CL:SVREF CARDINAL-ONES HUNDREDS))
    (WRITE-STRING* " hundred")
    (CL:WHEN (CL:PLUSP REM)
      (CL:WRITE-CHAR #\Space)))
  (CL:WHEN (CL:PLUSP REM)
    (CL:MULTIPLE-VALUE-BIND (TENS ONES)
      (CL:TRUNCATE REM 10)
      [COND
        [(< 1 TENS)
          (WRITE-STRING* (CL:SVREF CARDINAL-TENS TENS))
          (CL:WHEN (CL:PLUSP ONES)
            (CL:WRITE-CHAR #\-)
            (WRITE-STRING* (CL:SVREF CARDINAL-ONES ONES)))]
        [(= TENS 1)
          (WRITE-STRING* (CL:SVREF CARDINAL-TEENS ONES))
          (CL:PLUSP ONES)
          (WRITE-STRING* (CL:SVREF CARDINAL-ONES ONES]))])
```

```
(CL:DEFUN FORMAT-PRINT-CARDINAL (N)
```

```
(COND
  ((MINUSP N)
    (WRITE-STRING* "negative ")
    (FORMAT-PRINT-CARDINAL-AUX (- N)
      0 N))
  ((ZEROP N)
    (WRITE-STRING* "zero"))
  (T (FORMAT-PRINT-CARDINAL-AUX N 0 N))))
```

```
(CL:DEFUN FORMAT-PRINT-CARDINAL-AUX (N PERIOD ERR)
```

```
(CL:MULTIPLE-VALUE-BIND (BEYOND HERE)
  (CL:TRUNCATE N 1000)
  (CL:UNLESS (<= PERIOD 10)
    (FORMAT-ERROR "Number too large to print in English: ~:~D" ERR))
  (CL:UNLESS (ZEROP BEYOND)
    (FORMAT-PRINT-CARDINAL-AUX BEYOND (CL:1+ PERIOD)
      ERR))
  (CL:UNLESS (ZEROP HERE)
    (CL:UNLESS (ZEROP BEYOND)
      (CL:WRITE-CHAR #\Space))
    (FORMAT-PRINT-SMALL-CARDINAL HERE)
    (WRITE-STRING* (CL:SVREF CARDINAL-PERIODS PERIOD)))))
```

```
(CL:DEFUN FORMAT-PRINT-ORDINAL (N)
```

```
(CL:WHEN (MINUSP N)
  (WRITE-STRING* "negative "))
[LET ((CL:NUMBER (ABS N))
  (CL:MULTIPLE-VALUE-BIND (TOP BOT)
```

```

(CL:TRUNCATE CL:NUMBER 100)
(CL:UNLESS (ZEROP TOP)
  (FORMAT-PRINT-CARDINAL (- CL:NUMBER BOT)))
(CL:WHEN (AND (CL:PLUSP TOP)
  (CL:PLUSP BOT))
  (CL:WRITE-CHAR #\Space))
(CL:MULTIPLE-VALUE-BIND (TENS ONES)
  (CL:TRUNCATE BOT 10)
  (COND
    ((= BOT 12)
      (WRITE-STRING* "twelfth"))
    ((= TENS 1)
      (WRITE-STRING* (CL:SVREF CARDINAL-TEENS ONES))
      (WRITE-STRING* "th"))
    ((AND (ZEROP TENS)
      (CL:PLUSP ONES))
      (WRITE-STRING* (CL:SVREF ORDINAL-ONES ONES)))
    ((AND (ZEROP ONES)
      (CL:PLUSP TENS))
      (WRITE-STRING* (CL:SVREF ORDINAL-TENS TENS)))
    ((CL:PLUSP BOT)
      (WRITE-STRING* (CL:SVREF CARDINAL-TENS TENS))
      (CL:WRITE-CHAR #\-)
      (WRITE-STRING* (CL:SVREF ORDINAL-ONES ONES)))
    ((CL:PLUSP CL:NUMBER)
      (WRITE-STRING* "th"))
    (T (WRITE-STRING* "zeroeth")))))

```

(CL:DEFUN **FORMAT-PRINT-OLD-ROMAN** (N)

:: Print Roman numerals

```

(CL:UNLESS (< 0 N 5000)
  (FORMAT-ERROR "Number too large to print in old Roman numerals: ~:~D" N))
(CL:DO [(CHAR-LIST ' (#\D #\C #\L #\X #\V #\I)
  (CDR CHAR-LIST))
  (VAL-LIST '(500 100 50 10 5 1)
  (CDR VAL-LIST))
  (CUR-CHAR #\M (CAR CHAR-LIST))
  (CUR-VAL 1000 (CAR VAL-LIST))
  (START N (CL:DO [(I START (PROGN (CL:WRITE-CHAR CUR-CHAR)
    (- I CUR-VAL)
    ((< I CUR-VAL)
    I))])
  ((ZEROP START)))]

```

(CL:DEFUN **FORMAT-PRINT-ROMAN** (N)

```

(CL:UNLESS (< 0 N 4000)
  (FORMAT-ERROR "Number too large to print in Roman numerals: ~:~D" N))
(CL:DO [(CHAR-LIST ' (#\D #\C #\L #\X #\V #\I)
  (CDR CHAR-LIST))
  (VAL-LIST '(500 100 50 10 5 1)
  (CDR VAL-LIST))
  (SUB-CHARS ' (#\C #\X #\I #\I)
  (CDR SUB-CHARS))
  (SUB-VAL '(100 10 10 1 1 0)
  (CDR SUB-VAL))
  (CUR-CHAR #\M (CAR CHAR-LIST))
  (CUR-VAL 1000 (CAR VAL-LIST))
  (CUR-SUB-CHAR #\C (CAR SUB-CHARS))
  (CUR-SUB-VAL 100 (CAR SUB-VAL))
  (START N (CL:DO [(I START (PROGN (CL:WRITE-CHAR CUR-CHAR)
    (- I CUR-VAL)
    ((< I CUR-VAL)
    (COND
      ((<= (- CUR-VAL CUR-SUB-VAL)
      I)
      (CL:WRITE-CHAR CUR-SUB-CHAR)
      (CL:WRITE-CHAR CUR-CHAR)
      (- I (- CUR-VAL CUR-SUB-VAL)))
      (T I)))]
  ((ZEROP START)))]

```

(CL:DEFUN **FORMAT-PRINT-DECIMAL** (COLON ATSIGN PARMS)

:: Decimal ~D

```

(FORMAT-PRINT-NUMBER (POP-FORMAT-ARG)
  10 COLON ATSIGN PARMS))

```

(CL:DEFUN **FORMAT-PRINT-BINARY** (COLON ATSIGN PARMS)

:: Binary ~B

```

(FORMAT-PRINT-NUMBER (POP-FORMAT-ARG)
  2 COLON ATSIGN PARMS))

```

```
(CL:DEFUN FORMAT-PRINT-OCTAL (COLON ATSIGN PARMS)
  ;; Octal ~O
  (FORMAT-PRINT-NUMBER (POP-FORMAT-ARG)
    8 COLON ATSIGN PARMS))
```

```
(CL:DEFUN FORMAT-PRINT-HEXADECIMAL (COLON ATSIGN PARMS)
  ;; Hexadecimal ~X
  (FORMAT-PRINT-NUMBER (POP-FORMAT-ARG)
    16 COLON ATSIGN PARMS))
```

```
(CL:DEFUN FORMAT-PRINT-RADIX (COLON ATSIGN PARMS)
  ;; Radix ~R
  [LET ((CL:NUMBER (POP-FORMAT-ARG))
        (CL:IF (CAR PARMS)
          (FORMAT-PRINT-NUMBER CL:NUMBER (pop PARMS)
            COLON ATSIGN PARMS)
          (CL:IF PARMS
            (FORMAT-WRITE-FIELD (FORMAT-STRINGIFY-OUTPUT (FORMAT-PRINT-RADIX-AUX CL:NUMBER COLON ATSIGN))
              (CADR PARMS)
              1 0 (COND
                ((CADDR PARMS))
                (T #\Space)
                NIL)
              T)
            (FORMAT-PRINT-RADIX-AUX CL:NUMBER COLON ATSIGN)))))]
```

```
(CL:DEFUN FORMAT-PRINT-RADIX-AUX (CL:NUMBER COLON ATSIGN)
  (CL:IF (TYPEP CL:NUMBER 'INTEGER)
    (CL:IF ATSIGN
      (CL:IF COLON
        (FORMAT-PRINT-OLD-ROMAN CL:NUMBER)
        (FORMAT-PRINT-ROMAN CL:NUMBER))
      (CL:IF COLON
        (FORMAT-PRINT-ORDINAL CL:NUMBER)
        (FORMAT-PRINT-CARDINAL CL:NUMBER)))
    (FORMAT-ERROR "Non-integer ~S can't be FORMATTed ~::~:[~::~~::~:[~::~@~]R" CL:NUMBER COLON ATSIGN)))
```

```
(CL:DEFUN FORMAT-FIXED (COLON ATSIGN PARMS)
  ;; Fixed-format floating point ~F
  (CL:WHEN COLON (FORMAT-ERROR "Colon flag not allowed"))
  [WITH-FORMAT-PARAMETERS PARMS ((W NIL)
    (D NIL)
    (K NIL)
    (OVF NIL)
    (PAD #\Space))
  ;; Note that the scale factor k defaults to nil. This is interpreted as zero by flonum-to-string, but more efficiently.
  (LET ((CL:NUMBER (POP-FORMAT-ARG))
        (CL:IF (FLOATP CL:NUMBER)
          (FORMAT-FIXED-AUX CL:NUMBER W D K OVF PAD ATSIGN)
          (CL:IF (CL:RATIONALP CL:NUMBER)
            (FORMAT-FIXED-AUX (COERCE CL:NUMBER 'FLOAT)
              W D K OVF PAD ATSIGN)
            (LET ((*PRINT-BASE* 10))
              (FORMAT-WRITE-FIELD (CL:PRINC-TO-STRING CL:NUMBER)
                W 1 0 #\Space T)))))]
```

```
(CL:DEFUN FORMAT-FIXED-AUX (NUMBER W D K OVF PAD ATSIGN)
  (CL:IF (NOT (OR W D K))
    (PROGN ;; Code snarfed from Spice printer OUTPUT-FLOAT
      (CL:WHEN (MINUSP NUMBER)
        (CL:WRITE-CHAR #\-)
        (CL:SETQ NUMBER (- NUMBER)))
      ;; When number is reasonable size, use FLONUM-TO-STRING, otherwise punt and PRINC it
      (CL:IF (AND (>= NUMBER 0.001)
        (<= NUMBER 1.0E+7))
        (CL:MULTIPLE-VALUE-BIND (STR LEN LPOINT TPOINT)
          (FLONUM-TO-STRING NUMBER)
          (CL:WHEN LPOINT (CL:WRITE-CHAR #\0))
          (WRITE-STRING* STR)
          (CL:WHEN TPOINT (CL:WRITE-CHAR #\0)))
        (CL:PRINC NUMBER)))
    [LET ((SPACELEFT W)
          (CL:WHEN (AND W (OR ATSIGN (MINUSP NUMBER)))
```

```

(CL:DECF SPACELEFT))
(CL:MULTIPLE-VALUE-BIND (STR LEN LPOINT TPOINT)
  (FLONUM-TO-STRING (ABS NUMBER)
    SPACELEFT D K)
;; if caller specifically requested no fraction digits, suppress the optional trailing zero
(CL:WHEN (AND D (ZEROP D))
  (SETQ TPOINT NIL))
(CL:WHEN W
  (CL:DECF SPACELEFT LEN)
  ;; optional leading zero force at least one digit
  (CL:WHEN LPOINT
    (CL:IF (OR (> SPACELEFT 0)
      TPOINT)
      (CL:DECF SPACELEFT)
      (SETQ LPOINT NIL)))
  ;; optional trailing zero
  (CL:WHEN TPOINT
    (CL:IF (> SPACELEFT 0)
      (CL:DECF SPACELEFT)
      (SETQ TPOINT NIL))))
[COND
  ((AND W (< SPACELEFT 0)
    OVF)
  ;; field width overflow
  (CL:DOTIMES (I W)
    (CL:WRITE-CHAR OVF)))
  (T (CL:WHEN W
    (CL:DOTIMES (I SPACELEFT)
      (CL:WRITE-CHAR PAD)))
    (CL:IF (MINUSP NUMBER)
      (CL:WRITE-CHAR #\-)
      (CL:IF ATSIGN (CL:WRITE-CHAR #\+)))
    (CL:WHEN LPOINT (CL:WRITE-CHAR #\0))
    (WRITE-STRING* STR)
    (CL:WHEN TPOINT (CL:WRITE-CHAR #\0))))))

```

```

(CL:DEFUN FORMAT-EXPONENTIAL (COLON ATSIGN PARMS)

```

;; Exponential-format floating point ~E

```

(CL:WHEN COLON (FORMAT-ERROR "Colon flag not allowed"))
[WITH-FORMAT-PARAMETERS PARMS ((W NIL)
  (D NIL)
  (E NIL)
  (K 1)
  (OVF NIL)
  (PAD #\Space)
  (MARKER NIL))
  (LET ((CL:NUMBER (POP-FORMAT-ARG))
    (CL:IF (FLOATP CL:NUMBER)
      (FORMAT-EXP-AUX CL:NUMBER W D E K OVF PAD MARKER ATSIGN)
      (CL:IF (CL:RATIONALP CL:NUMBER)
        (FORMAT-EXP-AUX (COERCE CL:NUMBER 'FLOAT)
          W D E K OVF PAD MARKER ATSIGN)
        (LET ((*PRINT-BASE* 10))
          (FORMAT-WRITE-FIELD (CL:PRINC-TO-STRING CL:NUMBER)
            W 1 0 #\Space T))))))]

```

```

(CL:DEFUN FORMAT-EXPONENT-MARKER (CL:NUMBER)

```

```

  (CL:IF (TYPEP CL:NUMBER *READ-DEFAULT-FLOAT-FORMAT*)
    #\E
    (CL:ETYPECASE CL:NUMBER
      (CL:SHORT-FLOAT #\S)
      (CL:SINGLE-FLOAT #\F))))

```

```

(CL:DEFUN FORMAT-EXP-AUX (NUMBER W D E K OVF PAD MARKER ATSIGN)

```

;; Here we prevent the scale factor from shifting all significance out of a number to the right. We allow insignificant zeroes to be shifted in to the left
 ;; right, although it is an error to specify k and d such that this occurs. Perhaps we should detect both these conditions and flag them as errors. As
 ;; for now, we let the user get away with it, and merely guarantee that at least one significant digit will appear.

```

(CL:IF (NOT (OR W D))
  (CL:PRIN1 NUMBER)
  (CL:MULTIPLE-VALUE-BIND (NUM EXPT)
    (SCALE-EXPONENT (ABS NUMBER))
    [LET* ((EXPT (- EXPT K))
      (ESTR (CL:PRINC-TO-STRING (ABS EXPT)))
      (ELEN (CL:IF E
        (MAX (CL:LENGTH ESTR)
          E)
        (CL:LENGTH ESTR)))
      (FDIG (CL:IF D

```



```

(CL:IF (CL:PLUSP K)
  (CL:1+ (- D K)
    D)
  NIL)
(FMIN (CL:IF (MINUSP K)
  (- 1 K)
  NIL))
(SPACELEFT (CL:IF W
  (- W 2 ELEN)
  NIL)))
(CL:WHEN (OR ATSIGN (MINUSP NUMBER))
  (CL:DECF SPACELEFT))
(CL:IF (AND W E OVF (> ELEN E))
  (PROGN ;; exponent overflow
    (CL:DOTIMES (I W)
      (CL:WRITE-CHAR OVF)))
    (CL:MULTIPLE-VALUE-BIND (FSTR FLEN LPOINT TPOINT)
      (FLONUM-TO-STRING NUM SPACELEFT FDIG K FMIN))
    (CL:WHEN W
      (CL:DECF SPACELEFT FLEN)
      (CL:WHEN LPOINT
        (CL:IF (> SPACELEFT 0)
          (CL:DECF SPACELEFT)
          (SETQ LPOINT NIL))))))
(COND
  ((AND W (< SPACELEFT 0)
    OVF)
    ;; significand overflow
    (CL:DOTIMES (I W)
      (CL:WRITE-CHAR OVF)))
  (T (CL:WHEN W
    (CL:DOTIMES (I SPACELEFT)
      (CL:WRITE-CHAR PAD)))
    (CL:IF (MINUSP NUMBER)
      (CL:WRITE-CHAR #-)
      (CL:IF ATSIGN (CL:WRITE-CHAR #\+)))
    (CL:WHEN LPOINT (CL:WRITE-CHAR #\0))
    (WRITE-STRING* FSTR)
    ;; (cl:when tpoint (cl:write-char #\0))
    (CL:WRITE-CHAR (CL:IF MARKER
      MARKER
      (FORMAT-EXPONENT-MARKER NUMBER)))
    (CL:WRITE-CHAR (CL:IF (MINUSP EXPT)
      #-
      #\+))
    (CL:WHEN E
      ;; zero-fill before exponent if necessary
      (CL:DOTIMES (I (- E (CL:LENGTH ESTR)))
        (CL:WRITE-CHAR #\0))
      (WRITE-STRING* ESTR)))))))]))

```

(CL:DEFUN **FORMAT-GENERAL-FLOAT** (COLON ATSIGN PARMS)

;; General Floating Point --- ~G

(CL:WHEN COLON (**FORMAT-ERROR** "Colon flag not allowed"))

```

[WITH-FORMAT-PARAMETERS PARMS ((W NIL)
  (D NIL)
  (E NIL)
  (K NIL)
  (OVF #\*)
  (PAD #\Space)
  (MARKER NIL))

```

(LET ((CL:NUMBER (**POP-FORMAT-ARG**)))

;; The Excelsior edition does not say what to do if the argument is not a float. Here, we adopt the conventions used by ~F and ~E.

```

(CL:IF (FLOATP CL:NUMBER)
  (FORMAT-GENERAL-AUX CL:NUMBER W D E K OVF PAD MARKER ATSIGN)
  (CL:IF (CL:RATIONALP CL:NUMBER)
    (FORMAT-GENERAL-AUX (COERCE CL:NUMBER 'FLOAT)
      W D E K OVF PAD MARKER ATSIGN)
    (LET ((*PRINT-BASE* 10)
      (FORMAT-WRITE-FIELD (CL:PRINC-TO-STRING CL:NUMBER)
        W 1 0 #\Space T))))))

```

(CL:DEFUN **FORMAT-GENERAL-AUX** (CL:NUMBER W D E K OVF PAD MARKER ATSIGN)

```

(CL:MULTIPLE-VALUE-BIND (IGNORE N)
  (SCALE-EXPONENT (ABS CL:NUMBER))
  (DECLARE (IGNORE IGNORE)

```

;; Default d if omitted. The procedure is taken directly from the definition given in the manual, and is not very efficient, since we generate the
 ;; digits twice. Future maintainers are encouraged to improve on this.

```

(CL:UNLESS D
  (CL:MULTIPLE-VALUE-BIND (STR LEN)
    (FLONUM-TO-STRING (ABS CL:NUMBER))
    (DECLARE (IGNORE STR))
    [LET [(Q (CL:IF (= LEN 1)
      1
      (CL:1- LEN)))]
      (SETQ D (MAX Q (MIN N 7)))]
    [LET* ((EE (CL:IF E
      (+ E 2)
      4))
      (WW (CL:IF W
      (- W EE)
      NIL))
      (DD (- D N)))
      (COND
        ((<= 0 DD D)
          (FORMAT-FIXED-AUX CL:NUMBER WW DD NIL OVF PAD ATSIGN)
          (CL:DOTIMES (I EE)
            (CL:WRITE-CHAR #\Space))
          (T (FORMAT-EXP-AUX CL:NUMBER W D E (OR K 1)
            OVF PAD MARKER ATSIGN)))]))

(CL:DEFUN FORMAT-DOLLARS (COLON ATSIGN PARMS)
  ;; Dollars floating-point format ~$
  [WITH-FORMAT-PARAMETERS PARMS ((D 2)
    (N 1)
    (FW 0)
    (PAD #\Space))
    (LET* [(CL:NUMBER (POP-FORMAT-ARG))
      (SIGNSTR (CL:IF (MINUSP CL:NUMBER)
        "- "
        (CL:IF ATSIGN
          "+ "
          "")))]
      (CL:MULTIPLE-VALUE-BIND (STR NUMLength IG2 IG3 POINTPLACE)
        (FLONUM-TO-STRING (ABS CL:NUMBER)
          NIL D NIL)
        (DECLARE (IGNORE IG2 IG3))
        (CL:WHEN COLON (WRITE-STRING* SIGNSTR))
        (CL:DOTIMES [I (- FW NUMLength (CL:LENGTH SIGNSTR)
          (MAX 0 (- N POINTPLACE)
            (CL:WRITE-CHAR PAD))
          (CL:UNLESS COLON (WRITE-STRING* SIGNSTR))
          (CL:DOTIMES (I (- N POINTPLACE)
            (CL:WRITE-CHAR #\0))
            (WRITE-STRING* STR)))]))

(CL:DEFUN CHARPOS (STREAM)
  (CL:UNLESS (STREAMP STREAM)
    (CL:ERROR "CHARPOS: ~A isn't a stream" STREAM))
  (fetch (STREAM CHARPOSITION) of STREAM))

(CL:DEFUN WHITESPACE-CHAR-P (CH)
  (CL:MEMBER CH ' (#\Tab #\Page #\Space #\Backspace #\Newline #\Linefeed)
    :TEST
    (FUNCTION EQL))

(DEFMACRO NAME-ARRAY (CONTENTS)
  `(CL:MAKE-ARRAY , (LENGTH CONTENTS)
    :ELEMENT-TYPE T :INITIAL-CONTENTS ', CONTENTS))

(CL:DEFVAR *FORMAT-ARGUMENTS* NIL
  "List of FORMAT args yet unprocessed")

(CL:DEFVAR *FORMAT-CONTROL-STRING* NIL
  "Bound to FORMAT control string")

(CL:DEFVAR *FORMAT-DISPATCH-TABLE*
  (MAKE-DISPATCH-VECTOR (#\B FORMAT-PRINT-BINARY)
    (#\O FORMAT-PRINT-OCTAL)
    (#\D FORMAT-PRINT-DECIMAL)
    (#\X FORMAT-PRINT-HEXADECIMAL)
    (#\R FORMAT-PRINT-RADIX)
    (#\F FORMAT-FIXED)
    (#\E FORMAT-EXPONENTIAL)
    (#\G FORMAT-GENERAL-FLOAT)
    (#\A FORMAT-PRINC)
    (#\C FORMAT-PRINT-CHARACTER)

```

```
(#\P FORMAT-PLURAL)
(#\S FORMAT-PRIN1)
(#\T FORMAT-TAB)
(#\% FORMAT-TERPRI)
(#\& FORMAT-FRESHLINE)
(#\* FORMAT-SKIP-ARGUMENTS)
(#\| FORMAT-PAGE)
(#\~ FORMAT-TILDE)
(#\$ FORMAT-DOLLARS)
(#\? FORMAT-INDIRECTION)
(#\^ FORMAT-ESCAPE)
(#\; FORMAT-SEMICOLON-ERROR)
(#\[ FORMAT-CONDITION)
(#\{ FORMAT-ITERATION)
(#\<< FORMAT-JUSTIFICATION)
(#\ ( FORMAT-CAPITALIZATION)
(#\Newline FORMAT-NEWLINE))
```

"Table of functions called by SUB-FORMAT to process ~foo stuff")

```
(CL:DEFVAR *FORMAT-INDEX* NIL
 "Index into current control string")
```

```
(CL:DEFVAR *FORMAT-LENGTH* NIL
 "Length of current control string")
```

```
(CL:DEFVAR *FORMAT-ORIGINAL-ARGUMENTS* NIL
 "List of original FORMAT arguments")
```

```
(CL:DEFVAR CARDINAL-ONES (NAME-ARRAY (NIL "one" "two" "three" "four" "five" "six" "seven" "eight" "nine"))
 "Table of strings used by ~R")
```

```
(CL:DEFVAR CARDINAL-TENS (NAME-ARRAY (NIL NIL "twenty" "thirty" "forty" "fifty" "sixty" "seventy"
 "eighty" "ninety"))
 "Table of strings used by ~R")
```

```
(CL:DEFVAR CARDINAL-TEENS (NAME-ARRAY ("ten" "eleven" "twelve" "thirteen" "fourteen" "fifteen" "sixteen"
 "seventeen" "eighteen" "nineteen"))
 "Table of strings used by ~R")
```

```
(CL:DEFVAR CARDINAL-PERIODS (NAME-ARRAY (" " "thousand" "million" "billion" "trillion" "quadrillion"
 "quintillion" "sextillion" "septillion" "octillion" "
nonillion" "decillion"))
 "Table of strings used by ~R")
```

```
(CL:DEFVAR ORDINAL-ONES (NAME-ARRAY (NIL "first" "second" "third" "fourth" "fifth" "sixth" "seventh"
 "eighth" "ninth"))
 "Table of strings used by ~R")
```

```
(CL:DEFVAR ORDINAL-TENS (NAME-ARRAY (NIL "tenth" "twentieth" "thirtieth" "fortieth" "fiftieth" "sixtieth"
 "seventieth" "eightieth" "ninetieth"))
 "Table of strings used by ~R")
```

(DECLARE%: DONTEVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILERVARS

(ADDTOVAR **NLAMA**)

(ADDTOVAR **NLAML**)

(ADDTOVAR **LAMA**)

)

;; Arrange to use the correct compiler.

(PUTPROPS **CMLFORMAT FILETYPE** CL:COMPILE-FILE)

(PUTPROPS **CMLFORMAT COPYRIGHT** ("Venue & Xerox Corporation" 1986 1987 1988 1989 1990))

FUNCTION INDEX

CHARPOS	18	FORMAT-GENERAL-FLOAT	17	FORMAT-PRINT-OLD-ROMAN	14
FLONUM-TO-STRING	2	FORMAT-GET-PARAMETER	4	FORMAT-PRINT-ORDINAL	13
CL:FORMAT	5	FORMAT-GET-SEGMENTS	8	FORMAT-PRINT-RADIX	15
FORMAT-ADD-COMMAS	12	FORMAT-GET-TRAILING-SEGMENTS	8	FORMAT-PRINT-RADIX-AUX	15
FORMAT-BOOLEAN-CONDITION	7	FORMAT-INDIRECTION	11	FORMAT-PRINT-ROMAN	14
FORMAT-CAPITALIZATION	5	FORMAT-ITERATION	7	FORMAT-PRINT-SMALL-CARDINAL	13
FORMAT-CONDITION	7	FORMAT-JUSTIFICATION	9	FORMAT-ROUND-COLUMNS	9
FORMAT-DO-ITERATION	7	FORMAT-NEWLINE	10	FORMAT-SEMICOLON-ERROR	6
FORMAT-DOLLARS	18	FORMAT-PAGE	10	FORMAT-SKIP-ARGUMENTS	10
FORMAT-EAT-WHITESPACE	10	FORMAT-PLURAL	10	FORMAT-TAB	11
FORMAT-ERROR	2	FORMAT-PRIN1	11	FORMAT-TERPRI	9
FORMAT-ESCAPE	6	FORMAT-PRINC	11	FORMAT-TILDE	10
FORMAT-EXP-AUX	16	FORMAT-PRINT-BINARY	14	FORMAT-UNTAGGED-CONDITION	6
FORMAT-EXPONENT-MARKER	16	FORMAT-PRINT-CARDINAL	13	FORMAT-WRITE-FIELD	12
FORMAT-EXPONENTIAL	16	FORMAT-PRINT-CARDINAL-AUX	13	MAKE-PAD-SEGS	8
FORMAT-FIND-COMMAND	4	FORMAT-PRINT-CHARACTER	12	PARSE-FORMAT-OPERATION	4
FORMAT-FIXED	15	FORMAT-PRINT-DECIMAL	14	SCALE-EXPONENT	2
FORMAT-FIXED-AUX	15	FORMAT-PRINT-HEXADECIMAL	15	SCALE-EXPT-AUX	2
FORMAT-FRESHLINE	10	FORMAT-PRINT-NAMED-CHARACTER	12	SUB-FORMAT	5
FORMAT-FUNNY-CONDITION	6	FORMAT-PRINT-NUMBER	13	WHITESPACE-CHAR-P	18
FORMAT-GENERAL-AUX	17	FORMAT-PRINT-OCTAL	15		

VARIABLE INDEX

DIGIT-STRING	2	*FORMAT-LENGTH*	19	CARDINAL-TENS	19
FORMAT-ARGUMENTS	18	*FORMAT-ORIGINAL-ARGUMENTS*	19	ORDINAL-ONES	19
FORMAT-CONTROL-STRING	18	CARDINAL-ONES	19	ORDINAL-TENS	19
FORMAT-DISPATCH-TABLE	18	CARDINAL-PERIODS	19		
FORMAT-INDEX	19	CARDINAL-TEENS	19		

MACRO INDEX

FORMAT-FIND-CHAR	4	FORMAT-WITH-CONTROL-STRING	3	NEXTCHAR	3
FORMAT-PEEK	3	MAKE-DISPATCH-VECTOR	1	POP-FORMAT-ARG	3
FORMAT-STRINGIFY-OUTPUT	3	NAME-ARRAY	18	WITH-FORMAT-PARAMETERS	3

PROPERTY INDEX

CMLFORMAT	19
-----------------	----

STRUCTURE INDEX

FORMAT-ERROR	1
--------------------	---

CONSTANT INDEX

DIGITS	2
----------------	---
