File created: 4-Jun-2024 23:32:50 **{DSK}<home>matt>Interlisp>medley>SOURCES>CMLTYPES.;2**

edit by: mth

changes to: (IL:FUNCTIONS SYMBOL-TYPE)

previous date: 4-Jan-93 17:55:42 {DSK}<home>matt>Interlisp>medley>SOURCES>CMLTYPES.;1

Read Table: XCL

Package: LISP

Format: XCCS

```
(IL:RPAQQ IL:CMLTYPESCOMS
          (

;;; Implementation of Common Lisp type system.


;;; implementation by Greg Nuyens ,Larry Masinter and Jan Pedersen.


;;; Predicates

           (IL:FUNCTIONS COMMONP)

;;; Typep and friends

           (IL:VARIABLES *TYPEP-HASH-TABLE*)
           (IL:FUNCTIONS TYPEP TYPE-OF COERCE TYPECASE)
           (IL:FUNCTIONS %VALID-TYPE-P)
           (XCL:OPTIMIZERS TYPEP COERCE)

;;; for DEFTYPE

           (IL:DEFINE-TYPES IL:TYPES)
           (IL:FUNCTIONS DEFTYPE TYPE-EXPAND TYPE-EXPANDER SETF-TYPE-EXPANDER)
           (IL:SETFS TYPE-EXPANDER)
           (IL:DECLARE\: IL:DOCOPY IL:DONTEVAL@LOAD

                  ;; There is still code out there that calls the IL: versions

                  (IL:P (IL:MOVD 'TYPE-EXPAND 'IL:TYPE-EXPAND)
                        (IL:MOVD 'TYPE-EXPANDER 'IL:TYPE-EXPANDER)))

;;; Support functions

           (IL:FUNCTIONS ARRAY-TYPE SYMBOL-TYPE XCL:FALSE XCL:TRUE %RANGE-TYPE)
           (IL:FUNCTIONS NUMBERP FLOATP)
           (XCL:OPTIMIZERS NUMBERP FLOATP XCL:FALSE XCL:TRUE)

;;; For TYPEP

           (IL:FUNCTIONS %TYPEP-PRED BIGNUMP)

;;; for SUBTYPEP

           (IL:VARIABLES %NO-SUPER-TYPE *COMMON-LISP-BASE-TYPES* *BASE-TYPE-LATTICE*)
           (IL:FUNCTIONS SUBTYPEP SUBTYPEP-TYPE-EXPAND SI::DATATYPE-P SI::SUB-DATATYPE-P EQUAL-DIMENSIONS
                  COMPLETE-ARRAY-TYPE-DIMENSIONS COMPLETE-META-EXPRESSION-DEFAULTS RANGE<= BASE-SUBTYPEP
                  EQUAL-ELEMENT-TYPE USEFUL-TYPE-EXPANSION-P)

;;; Basic deftypes

           (IL:TYPES ATOM BIGNUM BIT CHARACTER CONS DOUBLE-FLOAT FIXNUM STREAM FLOAT FUNCTION HASH-TABLE INTEGER
                  KEYWORD LIST LONG-FLOAT MEMBER MOD NULL NUMBER PACKAGE SHORT-FLOAT SIGNED-BYTE STANDARD-CHAR
                  STRING-CHAR SINGLE-FLOAT SYMBOL UNSIGNED-BYTE RATIONAL READTABLE COMMON COMPILED-FUNCTION
                  SEQUENCE)

;;; Array Types

           (IL:TYPES ARRAY VECTOR STRING SIMPLE-STRING SIMPLE-ARRAY SIMPLE-VECTOR BIT-VECTOR SIMPLE-BIT-VECTOR)

;;; Fast predicates for typep

           (IL:DEFINE-TYPES TYPEP)
           (IL:FUNCTIONS DEFTYPEP)
           (TYPEP LIST SEQUENCE MEMBER ARRAY SIMPLE-ARRAY VECTOR SIMPLE-VECTOR COMPLEX INTEGER MOD SIGNED-BYTE
                  UNSIGNED-BYTE RATIONAL FLOAT STRING SIMPLE-STRING BIT-VECTOR SIMPLE-BIT-VECTOR)
```

;;; for TYPE-OF Interlisp types that have different common Lisp names

```
           (IL:PROP CMLTYPE IL:CHARACTER IL:FIXP IL:FLOATP IL:GENERAL-ARRAY IL:LISTP IL:LITATOM IL:ONED-ARRAY
                  IL:SMALLP IL:HARRAYP IL:TWOD-ARRAY)
           (IL:PROP CMLSUBTYPE-DESCRIMINATOR SYMBOL ARRAY)
```

;;; tell the filepkg what to do with the type-expander property

```
           (IL:PROP IL:PROPTYPE :TYPE-EXPANDER IL:TYPE-EXPANDER)
```

;;; Compiler options

```
           (IL:PROP (IL:FILETYPE IL:MAKEFILE-ENVIRONMENT)
                  IL:CMLTYPES)
           (IL:DECLARE\: IL:DONTEVAL@LOAD IL:DOEVAL@COMPILE IL:DONTCOPY (IL:LOCALVARS . T))))
```

;;; Implementation of Common Lisp type system.

;;; implementation by Greg Nuyens ,Larry Masinter and Jan Pedersen.

;;; Predicates

```
(DEFUN COMMONP (OBJECT)
   (TYPEP OBJECT 'COMMON))
```

;;; Typep and friends

```
(DEFPARAMETER *TYPEP-HASH-TABLE* (MAKE-HASH-TABLE :TEST 'EQ))

(DEFUN TYPEP (OBJECT TYPE)
   ;; Check if OBJECT is of type TYPE
   (LET* ((SYMBOL-TYPE (IF (CONSP TYPE)
                            (CAR TYPE)
                            TYPE))
          (FN (GETHASH SYMBOL-TYPE *TYPEP-HASH-TABLE*)))
         (IF FN
             (IF (CONSP TYPE)
                 (FUNCALL FN OBJECT (CDR TYPE))
                 (FUNCALL FN OBJECT))
             ;; Expand the type
             (IF (CONSP TYPE)
                 (CASE SYMBOL-TYPE
                     (SATISFIES (FUNCALL (CADR TYPE)
                                       OBJECT))
                     ((:DATATYPE IL:DATATYPE) (IL:TYPENAMEP OBJECT (CADR TYPE)))
                     (NOT (NOT (TYPEP OBJECT (CADR TYPE))))
                     (AND (DOLIST (SUB-TYPE (CDR TYPE)
                                         T)
                              (IF (NOT (TYPEP OBJECT SUB-TYPE))
                                  (RETURN NIL))))
                     (OR (DOLIST (SUB-TYPE (CDR TYPE)
                                        NIL)
                             (IF (TYPEP OBJECT SUB-TYPE)
                                 (RETURN T))))
                     (OTHERWISE (LET ((EXPANDER (TYPE-EXPANDER SYMBOL-TYPE)))
                                     (IF EXPANDER
                                         (TYPEP OBJECT (FUNCALL EXPANDER TYPE))
                                         (ERROR "Unknown type expression: ~s" TYPE)))))
                 (CASE SYMBOL-TYPE
                     ((T) T)
                     ((NIL) NIL)
                     (OTHERWISE (LET ((EXPANDER (TYPE-EXPANDER SYMBOL-TYPE)))
                                     (IF EXPANDER
                                         (TYPEP OBJECT (FUNCALL EXPANDER (LIST TYPE)))
                                         (ERROR "Unknown type expression: ~s" TYPE)))))))))

(DEFUN TYPE-OF (X)
   (LET ((TYPENAME (IL:\\INDEXATOMPNAME (IL:|fetch| IL:DTDNAME IL:|of| (IL:\\GETDTD (IL:NTYPX X))))))
        (SETQ TYPENAME (OR (GET TYPENAME 'CMLTYPE)
                           TYPENAME))
        (OR (LET ((D (GET TYPENAME 'CMLSUBTYPE-DESCRIMINATOR)))
                 (AND D (FUNCALL D X)))
```

```
            TYPENAME)))

(DEFUN COERCE (OBJECT RESULT-TYPE)
   ;; Coerce object to result-type if possible
   (IF (TYPEP OBJECT RESULT-TYPE)
       OBJECT
       (COND
          ((EQ RESULT-TYPE 'CHARACTER)
           (CHARACTER OBJECT))
          ((MEMBER RESULT-TYPE '(FLOAT SINGLE-FLOAT SHORT-FLOAT LONG-FLOAT DOUBLE-FLOAT)
                  :TEST
                  #'EQ)
           (FLOAT OBJECT))
          ((EQ (IF (CONSP RESULT-TYPE)
                   (CAR RESULT-TYPE)
                   RESULT-TYPE)
               'COMPLEX)
           (IF (CONSP RESULT-TYPE)
               (LET ((SUBTYPE (CADR RESULT-TYPE)))
                    (IF (COMPLEXP OBJECT)
                        (COMPLEX (COERCE (REALPART OBJECT)
                                         SUBTYPE)
                                 (COERCE (IMAGPART OBJECT)
                                         SUBTYPE))
                        (COMPLEX (COERCE OBJECT SUBTYPE))))
               (COMPLEX OBJECT)))
          ((TYPEP OBJECT 'SEQUENCE)
           (MAP RESULT-TYPE 'IDENTITY OBJECT))
          (T (ERROR "Cannot coerce ~S to type: ~S" OBJECT RESULT-TYPE)))))

(DEFMACRO TYPECASE (KEYFORM &REST FORMS)
   "Type dispatch, order is important, more specific types should appear first"
   `(LET (($$TYPE-VALUE ,KEYFORM))
         (COND
            ,@(MAPCAR #'(LAMBDA (FORM)
                               (LET ((PRED (IF (MEMBER (CAR FORM)
                                                       '(OTHERWISE T)
                                                       :TEST
                                                       #'EQ)
                                               T
                                               `(TYPEP $$TYPE-VALUE ',(CAR FORM))))
                                     (FORM (IF (NULL (CDR FORM))
                                               '(NIL)
                                               (CDR FORM))))
                                    `(,PRED ,@FORM)))
                     FORMS))))

(DEFUN %VALID-TYPE-P (TYPE)
   (IF (CONSP TYPE)
       (CASE (CAR TYPE)
           (SATISFIES T)
           ((OR AND) (EVERY '%VALID-TYPE-P (CDR TYPE)))
           (NOT (%VALID-TYPE-P (CADR TYPE)))
           ((:DATATYPE IL:DATATYPE) T)
           (OTHERWISE (AND (TYPE-EXPANDER TYPE)
                           T)))
       (OR (AND (TYPE-EXPANDER TYPE)
                T)
           (EQ TYPE T)
           (NULL TYPE))))

(XCL:DEFOPTIMIZER TYPEP (OBJ TYPE)
                        (IF (CONSTANTP TYPE)
                            (LET ((TYPE-EXPR (EVAL TYPE)))
                                 (IF (%VALID-TYPE-P TYPE-EXPR)
                                     `(,(%TYPEP-PRED TYPE-EXPR)
                                       ,OBJ)
                                     (PROGN (WARN "Can't optimize (typep ~s ~s); type not known." OBJ TYPE)
                                            'COMPILER:PASS)))
                            'COMPILER:PASS))

(XCL:DEFOPTIMIZER COERCE (OBJECT RESULT-TYPE)
                         ;; Open code the simple coerce cases
                         (IF (CONSTANTP RESULT-TYPE)
                             (CASE (EVAL RESULT-TYPE)
                                 (CHARACTER `(CHARACTER ,OBJECT))
                                 ((FLOAT SINGLE-FLOAT SHORT-FLOAT LONG-FLOAT DOUBLE-FLOAT)
                                    `(FLOAT ,OBJECT))
                                 (OTHERWISE 'COMPILER:PASS))
                             'COMPILER:PASS))
```

```
;;; for DEFTYPE

(XCL:DEF-DEFINE-TYPE IL:TYPES "Common Lisp type definitions")

(XCL:DEFDEFINER (DEFTYPE (:PROTOTYPE (LAMBDA (NAME)
                                               (AND (SYMBOLP NAME)
                                                    `(DEFTYPE ,NAME ("Arg list")
                                                       "Body")))))
    IL:TYPES (NAME DEFTYPE-ARGS &BODY BODY)
   (UNLESS (AND NAME (SYMBOLP NAME))
          (ERROR "Illegal name used in DEFTYPE: ~S" NAME))
   (LET
    ((EXPANDER-NAME (XCL:PACK (LIST "type-expand-" NAME)
                           (SYMBOL-PACKAGE NAME))))
    (MULTIPLE-VALUE-BIND (PARSED-BODY DECLS DOCSTRING)
        (IL:PARSE-DEFMACRO DEFTYPE-ARGS 'SI::%$$TYPE-FORM BODY NAME NIL :DEFAULT-DEFAULT ''*)
      `(EVAL-WHEN (EVAL COMPILE LOAD)
              (SETF (SYMBOL-FUNCTION ',EXPANDER-NAME)
                    #'(LAMBDA (SI::%$$TYPE-FORM)
                             ,@DECLS
                             (BLOCK ,NAME ,PARSED-BODY)))
              (SETF (TYPE-EXPANDER ',NAME)
                    ',EXPANDER-NAME)
              ,@(AND DOCSTRING `((SETF (DOCUMENTATION ',NAME 'TYPE)
                                       ,DOCSTRING)))
              ,@(IF (NULL DEFTYPE-ARGS)
                    (LET ((TYPEP-NAME (XCL:PACK (LIST "typep-evaluate-" NAME)
                                             (SYMBOL-PACKAGE NAME))))
                         `((EVAL-WHEN (LOAD)
                                  (SETF (SYMBOL-FUNCTION ',TYPEP-NAME)
                                        #'(LAMBDA (SI::%$$OBJECT)
                                                 (TYPEP SI::%$$OBJECT ',NAME)))
                                  (PUTHASH ',NAME *TYPEP-HASH-TABLE* ',TYPEP-NAME))
                           (EVAL-WHEN (EVAL)
                                  (PUTHASH ',NAME *TYPEP-HASH-TABLE* NIL)))))))))

(DEFUN TYPE-EXPAND (FORM &OPTIONAL (EXPANDER (TYPE-EXPANDER FORM)))
   ;; Expands a type form according to deftypes in effect.  The caller must ensure there is an expander for the form
   (IF EXPANDER
       (VALUES (FUNCALL EXPANDER (ETYPECASE FORM
                                     (SYMBOL (LIST FORM))
                                     (CONS FORM)))
              T)
       (VALUES FORM NIL)))

(DEFUN TYPE-EXPANDER (TYPE)
   (LET* ((SYMBOL-TYPE (ETYPECASE TYPE
                            (SYMBOL TYPE)
                            (CONS (CAR TYPE))))
          (EXPANDER (OR (GET SYMBOL-TYPE ':TYPE-EXPANDER)
                        (GET SYMBOL-TYPE 'IL:TYPE-EXPANDER))))
         (IF (AND (NULL EXPANDER)
                  (SYMBOLP TYPE)
                  (SI::DATATYPE-P TYPE))
             ;; Install a deftype
             (LET ((DEFTYPE-FORM `(DEFTYPE ,TYPE ()
                                     '(:DATATYPE ,TYPE))))
                  (IF (FBOUNDP 'XCL:COMPILE-FORM)
                      ;; Compile form on the fly
                      (XCL:COMPILE-FORM DEFTYPE-FORM)
                      (LET ((IL:DFNFLG NIL)
                            (IL:FILEPKGFLG NIL)
                            ;; DFNFLG nil makes sure this has an effect and filepkgflg nil makes sure it isn't remembered.

                            )
                           (EVAL DEFTYPE-FORM)))
                  (TYPE-EXPANDER TYPE))
             EXPANDER)))

(DEFMACRO SETF-TYPE-EXPANDER (SYMBOL EXPANDER)
   `(SETF (GET ,SYMBOL ':TYPE-EXPANDER)
          ,EXPANDER))

(DEFSETF TYPE-EXPANDER SETF-TYPE-EXPANDER)

(IL:DECLARE\: IL:DOCOPY IL:DONTEVAL@LOAD
(IL:MOVD 'TYPE-EXPAND 'IL:TYPE-EXPAND)

(IL:MOVD 'TYPE-EXPANDER 'IL:TYPE-EXPANDER)
)


;;; Support functions


(DEFUN ARRAY-TYPE (ARRAY)
   (LET ((RANK (ARRAY-RANK ARRAY)))
        (IF (XCL:SIMPLE-ARRAY-P ARRAY)
            (IF (EQ 1 RANK)
                (LET ((SIZE (ARRAY-TOTAL-SIZE ARRAY)))
                     (COND
                        ((SIMPLE-STRING-P ARRAY)
                         (LIST 'SIMPLE-STRING SIZE))
                        ((SIMPLE-BIT-VECTOR-P ARRAY)
                         (LIST 'SIMPLE-BIT-VECTOR SIZE))
                        (T (LET ((ELT-TYPE (ARRAY-ELEMENT-TYPE ARRAY)))
                                (IF (EQ ELT-TYPE T)
                                    (LIST 'SIMPLE-VECTOR SIZE)
                                    (LIST 'SIMPLE-ARRAY ELT-TYPE (LIST SIZE)))))))
                (LIST 'SIMPLE-ARRAY (ARRAY-ELEMENT-TYPE ARRAY)
                      (ARRAY-DIMENSIONS ARRAY)))
            (IF (EQ 1 RANK)
                (LET ((SIZE (ARRAY-TOTAL-SIZE ARRAY)))
                     (COND
                        ((STRINGP ARRAY)
                         (LIST 'STRING SIZE))
                        ((BIT-VECTOR-P ARRAY)
                         (LIST 'BIT-VECTOR SIZE))
                        (T (LIST 'VECTOR (ARRAY-ELEMENT-TYPE ARRAY)
                                 SIZE))))
                (LIST 'ARRAY (ARRAY-ELEMENT-TYPE ARRAY)
                      (ARRAY-DIMENSIONS ARRAY))))))


(DEFUN SYMBOL-TYPE (SYMBOL)                                     ; Edited  4-Jun-2024 23:23 by mth
   (COND
      ((NULL SYMBOL)
       'NULL)
      ((KEYWORDP SYMBOL)
       'KEYWORD)
      (T 'SYMBOL)))


(DEFUN XCL:FALSE ()
   NIL)


(DEFUN XCL:TRUE ()
   T)


(DEFUN %RANGE-TYPE (BASE-TYPE LOW HIGH RANGE-LIST)

   ;; Returns a type form discriminating basetype.  Rangelist is a list of (decreasing) subranges of the full range of basetype (represented as a list of
   ;; low, high and subtype).  If low and high fall within its range, a form is returned which discriminates on the subtype, and checks the range.  If low
   ;; and high are exactly the range of the subtype then no range checking form is returned.

   (COND
      ((AND (EQ LOW '*)
            (EQ HIGH '*))
       BASE-TYPE)
      ((OR (EQ LOW '*)
           (EQ HIGH '*))
       `(AND ,BASE-TYPE (SATISFIES (LAMBDA (X)
                                          ,@(IF (NOT (EQ LOW '*))
                                                `((,(COND
                                                       ((CONSP LOW)
                                                        (SETQ LOW (CAR LOW))
                                                        '<)
                                                       (T '<=))
                                                   ,LOW X)))
                                          ,@(IF (NOT (EQ HIGH '*))
                                                `((,(COND
                                                       ((CONSP HIGH)
                                                        (SETQ HIGH (CAR HIGH))
                                                        '<)
                                                       (T '<=))
                                                   X
                                                   ,HIGH))))))
      (T (DOLIST (X RANGE-LIST `(AND ,BASE-TYPE (SATISFIES (LAMBDA (X)
                                                                  (AND (,(COND
                                                                            ((CONSP LOW)
                                                                             (SETQ LOW (CAR LOW))
                                                                             '<)
                                                                            (T '<=))
                                                                      ,LOW X)
                                                                      (,(COND
                                                                            ((CONSP HIGH)
                                                                             (SETQ HIGH (CAR HIGH))
                                                                             '<)
                                                                            (T '<=))
                                                                       X
                                                                       ,HIGH))))))
             ;; If the limits are exactly the range specified in the rangelist, then return the corresponding type (since no range-check will be required
             ;; in the result).
             (IF (AND (EQUAL LOW (CAR X))
                      (EQUAL HIGH (CADR X)))
                 (RETURN (CADDR X)))
             ;; If the limits are within the range, then remember the basetype.
             (IF (<= (CAR X)
                     (IF (CONSP LOW)
                         (1+ (CAR LOW))
                         LOW)
                     (IF (CONSP HIGH)
                         (1- (CAR HIGH))
                         HIGH)
                     (CADR X))
                 (SETQ BASE-TYPE (CADDR X)))))))

(DEFUN NUMBERP (X)
   (AND (IL:NUMBERP X)
        T))

(DEFUN FLOATP (X)
   (AND (IL:FLOATP X)
        T))

(XCL:DEFOPTIMIZER NUMBERP (X)
                             `(AND (IL:NUMBERP ,X)
                                   T))

(XCL:DEFOPTIMIZER FLOATP (X)
                           `(AND (IL:FLOATP ,X)
                                 T))

(XCL:DEFOPTIMIZER XCL:FALSE (&BODY IL:FORMS)
                               `(PROG1 NIL ,@IL:FORMS))

(XCL:DEFOPTIMIZER XCL:TRUE (&BODY XCL::FORMS)
                             `(PROG1 T ,@XCL::FORMS))

;;; For TYPEP

(DEFUN %TYPEP-PRED (TYPE)
   ;; returns the predicate of one argument that determines this type.
   (COND
      ((CONSP TYPE)
       (CASE (CAR TYPE)
           (SATISFIES (CADR TYPE))
           ((:DATATYPE IL:DATATYPE) `(LAMBDA (SI::%$$OBJECT)
                                            (IL:TYPENAMEP SI::%$$OBJECT ',(CADR TYPE))))
           ((AND OR NOT) `(LAMBDA (SI::%$$OBJECT)
                                 (,(CAR TYPE)
                                  ,@(MAPCAR #'(LAMBDA (SUBTYPE)
                                                     (LIST (%TYPEP-PRED SUBTYPE)
                                                           'SI::%$$OBJECT))
                                           (CDR TYPE)))))
           (OTHERWISE (LET ((EXPANDER (TYPE-EXPANDER (CAR TYPE))))
                           (IF EXPANDER
                               (%TYPEP-PRED (FUNCALL EXPANDER TYPE))
                               (CERROR "Look again for a deftype on ~S." "No type definition for ~S. Specify one
                                      with DEFTYPE." TYPE))))))
      (T (COND
            ((EQ TYPE T)
             'XCL:TRUE)
            ((EQ TYPE NIL)
             'XCL:FALSE)
            (T (LET ((EXPANDER (TYPE-EXPANDER TYPE)))
                    (COND
                       (EXPANDER (%TYPEP-PRED (FUNCALL EXPANDER (LIST TYPE))))
                       (T ;; there is no deftype on this  non-list type.
                          (LOOP (IF (TYPE-EXPANDER TYPE)
                                    (RETURN NIL))
                                (CERROR "Use the deftype you have specified." "No type definition for ~S.
                                       Specify one with DEFTYPE." TYPE))
                          (%TYPEP-PRED TYPE))))))))


(DEFUN BIGNUMP (X)
   (OR (IL:TYPENAMEP X 'IL:FIXP)
       (IL:TYPENAMEP X 'BIGNUM)))


;;; for SUBTYPEP


(DEFCONSTANT %NO-SUPER-TYPE 0
   "the value in the dtdsupertype field which indicates no super type.")


(DEFCONSTANT *COMMON-LISP-BASE-TYPES*
   ;; The types which are known to be disjoint from any type explicitly handled by subtypep.

   '(;; The only types that need to be in this list are types on page 43 that expand into a satisfies or datatype clause, i.e. any type that expands into
     ;; something that base-subtypep doesn't know to handle, e.g. satisfies.

     ARRAY ATOM BIGNUM                                              ; even though bignum expands into a datatype, that datatype is
                                                                    ; not a subdatatype of integer, etc. so must be explicitly handled.
     CHARACTER COMMON COMPLEX COMPILED-FUNCTION CONS IL:DATATYPE
                                                                    ; this is only here for back-compatibility.  The first global
                                                                    ; recompile, this can go.
     :DATATYPE FLOAT FUNCTION HASH-TABLE INTEGER KEYWORD NIL NULL NUMBER PACKAGE PATHNAME RANDOM-STATE RATIO
                                                                    ; same comment for ratio as bignum.
     RATIONAL READTABLE SIMPLE-ARRAY STANDARD-CHAR STREAM STRING-CHAR SYMBOL T))


(DEFCONSTANT *BASE-TYPE-LATTICE*
   '((NUMBER RATIONAL INTEGER RATIO FIXNUM BIGNUM COMPLEX FLOAT)
     (RATIONAL INTEGER RATIO FIXNUM BIGNUM)
     (INTEGER FIXNUM BIGNUM)
     (CHARACTER STRING-CHAR STANDARD-CHAR)
     (STRING-CHAR STANDARD-CHAR)
     (LIST NULL)
     (SYMBOL KEYWORD NULL)
     (ARRAY SIMPLE-ARRAY)
     #'COMPILED-FUNCTION
     (NIL)
     (IL:DATATYPE :DATATYPE)                                        ; the presence of il:datatype is for back compatibility.
     (:DATATYPE IL:DATATYPE))
   "the lattice which tells the (base) subtypes of any base type.")


(DEFUN SUBTYPEP (TYPE1 TYPE2)
   ;; Returns T if type1 is a subtype of type2.  If second value is nil, couldn't decide.

   (IF (EQUAL TYPE1 TYPE2)
       ;; no need to complete any further recursion, so just return success.

       (VALUES T T)
       (CASE (IF (CONSP TYPE1)
                 (CAR TYPE1)
                 TYPE1)
           (AND
              ;; (subtypep '(and t1 t2 ...) 't3) <= (or (subtypep 't1 't3) (subtypep 't2 't3) ...  ) because '(and t1 t2 ...) denotes the intersection of types
              ;; t1, t2, ...

              ;; Even if none of the conjuncts is a subtype, we still can't return (NIL T) because the intersection might still be a subtype.

              (LET ((RESULT NIL)
                    CERTAINTY CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                   (SETQ CERTAINTY (DOLIST (TYPE1-CONJUNCT (CDR TYPE1)
                                                  NIL)
                                     (MULTIPLE-VALUE-SETQ (CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                                            (SUBTYPEP TYPE1-CONJUNCT TYPE2))
                                     (WHEN CONJUNCT-RESULT
                                         (SETQ RESULT T)
                                         (IF CONJUNCT-CERTAINTY (RETURN T)))))
                   (VALUES RESULT CERTAINTY)))
           (OR
              ;; (subtypep '(or t1 t2 ...) 't3) <=> (and (subtypep 't1 't3) (subtypep 't2 't3) ...)

              (LET ((RESULT T)
                    CERTAINTY
                    (LOOP-CERTAINTY T)
                    CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                   (SETQ CERTAINTY (DOLIST (TYPE1-CONJUNCT (CDR TYPE1)
                                                  LOOP-CERTAINTY)
                                    ;; returns t only if every conjunct clause is a certain subtype, or if one conjunct clause is certainly
                                    ;; not a subtype
                                     (MULTIPLE-VALUE-SETQ (CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                                            (SUBTYPEP TYPE1-CONJUNCT TYPE2))
                                     (COND
                                        ((NULL CONJUNCT-RESULT)
                                         (SETQ RESULT NIL)
                                         (IF CONJUNCT-CERTAINTY
                                             (RETURN T)

                                            ;; else continue to look for a more cetain result

                                             (SETQ LOOP-CERTAINTY NIL)))
                                        (T (IF (NULL CONJUNCT-CERTAINTY)
                                               (SETQ LOOP-CERTAINTY NIL))))))
                   (VALUES RESULT CERTAINTY)))
           (OTHERWISE
              ;; Try to expand type1
              (MULTIPLE-VALUE-BIND (NEW-TYPE1 EXPANDED?)
                  (SUBTYPEP-TYPE-EXPAND TYPE1)
                (IF (USEFUL-TYPE-EXPANSION-P NEW-TYPE1 EXPANDED?)
                    (SUBTYPEP NEW-TYPE1 TYPE2)

                    ;; We now have a base type for type1, there is nothing further to be done with it, by itself.  So we check for special cases in
                    ;; type2

                    (CASE (IF (CONSP TYPE2)
                              (CAR TYPE2)
                              TYPE2)
                        (AND

                           ;;  (subtypep 't1 '(and t2 t3 ...)) <=> (and (subtypep 't1 't2) (subtypep 't1 't3) ...) because '(and t2 t3 ...) denotes the
                           ;; intersection of types t2, t3, ...

                           (LET ((RESULT T)
                                 CERTAINTY
                                 (LOOP-CERTAINTY T)
                                 CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                                (SETQ CERTAINTY (DOLIST (TYPE2-CONJUNCT (CDR TYPE2)
                                                               LOOP-CERTAINTY)
                                                    (MULTIPLE-VALUE-SETQ (CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                                                           (SUBTYPEP TYPE1 TYPE2-CONJUNCT))
                                                    (COND
                                                       ((NULL CONJUNCT-RESULT)
                                                        (SETQ RESULT NIL)
                                                        (IF CONJUNCT-CERTAINTY
                                                            (RETURN T)

                                                           ;; else continue to look for a more cetain result

                                                            (SETQ LOOP-CERTAINTY NIL)))
                                                       (T (IF (NULL CONJUNCT-CERTAINTY)
                                                              (SETQ LOOP-CERTAINTY NIL))))))
                                (VALUES RESULT CERTAINTY)))
                        (OR

                           ;; (subtypep 't1 '(or t2 t3 ...)) <=> (or (subtypep 't1 't2) (subtypep 't1 't3) ...  ) because '(or t1 t2 ...) denotes the union
                           ;; of types t1, t2, ...

                           ;; We can't ever return (values nil t) because the t2..tn might form a partition of t1, i.e.

                           ;; (deftype evenp nil '(and integer (satisfies %evenp)))

                           ;; (deftype oddp nil '(and integer (satisfies %oddp)))

                           ;; (subtypep 'integer '(or evenp oddp)) is true, but the satisfies makes it undecidable, so we must return (nil nil).

                           (LET ((RESULT NIL)
                                 CERTAINTY CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                                (SETQ CERTAINTY (DOLIST (TYPE2-CONJUNCT (CDR TYPE2)
                                                               NIL)
                                                    (MULTIPLE-VALUE-SETQ (CONJUNCT-RESULT CONJUNCT-CERTAINTY)
                                                           (SUBTYPEP TYPE1 TYPE2-CONJUNCT))
                                                    (WHEN CONJUNCT-RESULT
                                                        (SETQ RESULT T)
                                                        (IF CONJUNCT-CERTAINTY (RETURN T)))))
                                (VALUES RESULT CERTAINTY)))
                        (OTHERWISE
                           ;; try to expand type2.
                           (MULTIPLE-VALUE-BIND (NEW-TYPE2 EXPANDED?)
                               (SUBTYPEP-TYPE-EXPAND TYPE2)
                             (IF (USEFUL-TYPE-EXPANSION-P NEW-TYPE2 EXPANDED?)
                                 (SUBTYPEP TYPE1 NEW-TYPE2)

                                 ;; we have now handled everything but base types.  There is no further expansion etc, to be done.

                                 (BASE-SUBTYPEP TYPE1 TYPE2))))))))))

(DEFUN SUBTYPEP-TYPE-EXPAND (TYPE)
   ;; Like type-expand, except it doesn't expand base-types.
   (IF (MEMBER (IF (CONSP TYPE)
                   (CAR TYPE)
                   TYPE)
              *COMMON-LISP-BASE-TYPES* :TEST #'EQ)
       (VALUES TYPE NIL)
       (TYPE-EXPAND TYPE)))


(DEFUN SI::DATATYPE-P (SI::NAME)
   ;; Returns T if name is a datatype known to the XAIE type system
   (AND (IL:\\TYPENUMBERFROMNAME SI::NAME)
        T))


(DEFUN SI::SUB-DATATYPE-P (TYPE1 TYPE2)
   ;; Returns T if type2 is a (not necessarily proper) supertype of type1.
   (DO* ((TYPE-NUMBER-1 (IL:\\TYPENUMBERFROMNAME TYPE1))
         (TYPE-NUMBER-2 (IL:\\TYPENUMBERFROMNAME TYPE2))
         (SUPER-TYPE-NUMBER TYPE-NUMBER-1 (IL:|fetch| IL:DTDSUPERTYPE IL:|of| (IL:\\GETDTD SUPER-TYPE-NUMBER))))
        ((EQ %NO-SUPER-TYPE SUPER-TYPE-NUMBER)
         ;; we didn't find type2 on type1's super chain so return NIL
         NIL)
      (IF (EQ SUPER-TYPE-NUMBER TYPE-NUMBER-2)
          (RETURN T))))


(DEFUN EQUAL-DIMENSIONS (DIMS1 DIMS2)
   ;; Says if dims1 and dims2 are the same in each dimension (allowing for wildcard's (*'s)).
   (OR (EQ DIMS1 '*)
       (EQ DIMS2 '*)
       (AND (EQUAL (LENGTH DIMS1)
                   (LENGTH DIMS2))
            (DO ((DIM1 DIMS1 (CDR DIM1))
                 (DIM2 DIMS2 (CDR DIM2)))
                ((NULL DIM1)
                 T)
               (IF (NOT (OR (EQ (CAR DIM1)
                                '*)
                            (EQ (CAR DIM2)
                                '*)
                            (EQ (CAR DIM1)
                                (CAR DIM2))))
                   (RETURN NIL))))))


(DEFUN COMPLETE-ARRAY-TYPE-DIMENSIONS (DIMENSIONS)
   (ETYPECASE DIMENSIONS
       (CONS DIMENSIONS)
       ((OR NULL (MEMBER *)) '*)
       (INTEGER (MAKE-LIST DIMENSIONS :INITIAL-ELEMENT '*))))


(DEFUN COMPLETE-META-EXPRESSION-DEFAULTS (TYPE)
   ;; given a type expression finishes the defaults the same way as the type-expander.
   (LET ((LIST-TYPE (IF (LISTP TYPE)
                        TYPE
                        (LIST TYPE))))
        (CASE (CAR LIST-TYPE)
            ((SIMPLE-ARRAY ARRAY) (XCL:DESTRUCTURING-BIND (ARRAY-TYPE &OPTIONAL (ELEMENT-TYPE '*)
                                                                 (DIMENSIONS '*))
                                         LIST-TYPE
                                         (LIST ARRAY-TYPE ELEMENT-TYPE (COMPLETE-ARRAY-TYPE-DIMENSIONS
                                                                              DIMENSIONS))))
            ((INTEGER FLOAT RATIONAL) (XCL:DESTRUCTURING-BIND (NUMERIC-TYPE &OPTIONAL (LOWER '*)
                                                                     (HIGHER '*))
                                             LIST-TYPE
                                             (LIST NUMERIC-TYPE LOWER HIGHER)))
            (COMPLEX (XCL:DESTRUCTURING-BIND (NUMERIC-TYPE &OPTIONAL (ELEMENT-TYPE '*))
                            LIST-TYPE
                            (LIST NUMERIC-TYPE ELEMENT-TYPE)))
            (T TYPE))))


(DEFUN RANGE<= (LOW2 LOW1 HIGH1 HIGH2 TYPE1 TYPE2)

;;; Returns t if bound1 is less than or equal bound2, allowing for wildcards.

   (IF (EQ TYPE1 'INTEGER)
       (COND
          ((CONSP LOW1)
           (SETQ LOW1 (+ (CAR LOW1)
                         1)))
          ((CONSP HIGH1)
           (SETQ HIGH1 (- (CAR HIGH1)
                          1)))))
   (IF (EQ TYPE2 ’INTEGER)
       (COND
          ((CONSP LOW2)
           (SETQ LOW2 (+ (CAR LOW2)
                         1)))
          ((CONSP HIGH2)
           (SETQ HIGH2 (- (CAR HIGH2)
                          1)))))
   (AND ;; check the low bounds

        (COND
           ((EQ LOW2 ’*)
            T)
           ((EQ LOW1 ’*)
            NIL)
           (T (IF (CONSP LOW2)
                  (IF (CONSP LOW1)
                      (<= (CAR LOW2)
                          (CAR LOW1))
                      (< (CAR LOW2)
                         LOW1))
                  (IF (CONSP LOW1)
                      (<= LOW2 (CAR LOW1))
                      (<= LOW2 LOW1)))))

        ;; Check the high bounds

        (COND
           ((EQ HIGH2 ’*)
            T)
           ((EQ HIGH1 ’*)
            NIL)
           (T (IF (CONSP HIGH2)
                  (IF (CONSP HIGH1)
                      (>= (CAR HIGH2)
                          (CAR HIGH1))
                      (> (CAR HIGH2)
                         HIGH1))
                  (IF (CONSP HIGH1)
                      (>= HIGH2 (CAR HIGH1))
                      (>= HIGH2 HIGH1)))))))

(DEFUN BASE-SUBTYPEP (TYPE1 TYPE2)

   ;; Contains subtypep’s special cases for base types.

   (LET ((SYMBOL-TYPE1 (IF (CONSP TYPE1)
                           (CAR TYPE1)
                           TYPE1))
         (SYMBOL-TYPE2 (IF (CONSP TYPE2)
                           (CAR TYPE2)
                           TYPE2)))
        (COND
           ((OR (EQ TYPE1 NIL)
                (EQ TYPE2 T)
                (EQUAL TYPE1 TYPE2))
            (VALUES T T))
           ((EQ TYPE2 ’COMMON)                                  ; Common does not list it’s subtypes in the lattice, since their
                                                                ; presence indicates that they are in COMMON.
            (IF (MEMBER SYMBOL-TYPE1 *COMMON-LISP-BASE-TYPES* :TEST #’EQ)

                ;; then this is part of common. Note this will include structures etc.

                (VALUES T T)
                (VALUES NIL T)))
           ((OR (NOT (MEMBER SYMBOL-TYPE1 *COMMON-LISP-BASE-TYPES* :TEST #’EQ))
                (NOT (MEMBER SYMBOL-TYPE2 *COMMON-LISP-BASE-TYPES* :TEST #’EQ)))
                                                                ; one of the types is something we can’t reason about (for
                                                                ; instance a user defined type that expands into satisfies.)
            (VALUES NIL NIL))

           ;; from this point on, we are only dealing with Common Lisp base types.

           ((EQ TYPE1 T)                                        ; t is not a subtype of anything but t, and that’s checked above).
            (VALUES NIL T))
           ((EQ TYPE2 NIL)                                      ; nil is not a supertype of anything but nil, and that’s checked
                                                                ; above).
            (VALUES NIL T))
           ((EQ TYPE2 ’ATOM)

            ;; this case could be explicitly added to the type lattice.  But if someone adds a base type, then they would have to remember to add it
            ;; as a sub type of atom, (which they wouldn’t.)

            (IF (EQ TYPE1 ’CONS)                                ; this is the only base type that isn’t a subtype of atom.
                (VALUES NIL T)
                (VALUES T T)))
           ((NOT (OR (EQ SYMBOL-TYPE1 SYMBOL-TYPE2)
                     (MEMBER SYMBOL-TYPE1 (ASSOC SYMBOL-TYPE2 *BASE-TYPE-LATTICE* :TEST #’EQ)
                            :TEST
                            #'EQ)))
            ;; since we are now dealing with only base types, we can make sure that type1 (without its arguments) is a subtype of type2, before
            ;; checking the constraints on the arguments.
            (VALUES NIL T))
           (T ;; Now check the constraints on the type arguments.
              (LET ((TYPE1 (COMPLETE-META-EXPRESSION-DEFAULTS TYPE1))
                    (TYPE2 (COMPLETE-META-EXPRESSION-DEFAULTS TYPE2)))
                   (CASE (IF (CONSP TYPE1)
                             (CAR TYPE1)
                             TYPE1)
                       ((ARRAY SIMPLE-ARRAY)
                          ;; the type will look like (simple-array element-type dimensions)
                          (XCL:DESTRUCTURING-BIND (ARRAY-TYPE1 ELEMENT-TYPE-1 DIMS-1)
                                 TYPE1
                                 (XCL:DESTRUCTURING-BIND (ARRAY-TYPE2 ELEMENT-TYPE-2 DIMS-2)
                                        TYPE2
                                        (IF (AND (EQUAL-ELEMENT-TYPE ELEMENT-TYPE-1 ELEMENT-TYPE-2)
                                                 (EQUAL-DIMENSIONS DIMS-1 DIMS-2))
                                            (VALUES T T)
                                            (VALUES NIL T)))))
                       ((:DATATYPE IL:DATATYPE)
                          ;; we wouldn't have made it here if they weren't both datatypes, since only datatype is a subtype of datatype in the
                          ;; lattice.
                          (VALUES (SI::SUB-DATATYPE-P (CADR TYPE1)
                                         (CADR TYPE2))
                                 T))
                       ((INTEGER RATIONAL FLOAT) (CASE TYPE2
                                                     (NUMBER
                                                        ;; number doesn't take ranges, there's nothing  to verify.
                                                        (VALUES T T))
                                                     (OTHERWISE (XCL:DESTRUCTURING-BIND
                                                                 (NUMERIC-TYPE1 LOW1 HIGH1)
                                                                 TYPE1
                                                                 (XCL:DESTRUCTURING-BIND (NUMERIC-TYPE2 LOW2
                                                                                                 HIGH2)
                                                                        TYPE2
                                                                        (IF (RANGE<= LOW2 LOW1 HIGH1 HIGH2
                                                                                   NUMERIC-TYPE1 NUMERIC-TYPE2)
                                                                            (VALUES T T)
                                                                            (VALUES NIL T)))))))
                       (COMPLEX (CASE TYPE2
                                    (NUMBER (VALUES T T))
                                    (OTHERWISE
                                       ;; typep2 must be complex
                                       (LET ((ELT-TYPE1 (CADR TYPE1))
                                             (ELT-TYPE2 (CADR TYPE2)))
                                            (COND
                                               ((EQ ELT-TYPE2 '*)
                                                (VALUES T T))
                                               ((EQ ELT-TYPE1 '*)
                                                (VALUES NIL T))
                                               (T (SUBTYPEP ELT-TYPE1 ELT-TYPE2)))))))
                       (OTHERWISE
                          ;; these are two base types.  the lattice said they are subtypes, and there are no special rules on the arguments, so
                          ;; the result is (t t) if they are equal
                          (VALUES T T))))))))


(DEFUN EQUAL-ELEMENT-TYPE (ELEMENT-TYPE-1 ELEMENT-TYPE-2)
   ;; returns t if they are element types for compatible array types.
   (COND
      ((EQ ELEMENT-TYPE-2 '*)
       T)
      ((EQ ELEMENT-TYPE-1 '*)
       NIL)
      (T (EQUAL (IL:%GET-CANONICAL-CML-TYPE ELEMENT-TYPE-1)
                (IL:%GET-CANONICAL-CML-TYPE ELEMENT-TYPE-2)))))


(DEFUN USEFUL-TYPE-EXPANSION-P (EXPANSION EXPANDED)
   ;; a type expansion only gained information if some expansion happened and the result isn't solely a satisfies form.
   (AND EXPANDED (NOT (AND (CONSP EXPANSION)
                           (EQ (CAR EXPANSION)
                               'SATISFIES)))))


;;; Basic deftypes
```

```
(DEFTYPE ATOM ()
   '(SATISFIES ATOM))

(DEFTYPE BIGNUM ()
   '(SATISFIES BIGNUMP))

(DEFTYPE BIT ()
   '(INTEGER 0 1))

(DEFTYPE CHARACTER ()
   '(SATISFIES CHARACTERP))

(DEFTYPE CONS ()
   '(:DATATYPE IL:LISTP))

(DEFTYPE DOUBLE-FLOAT (&OPTIONAL LOW HIGH)
   `(FLOAT ,LOW ,HIGH))

(DEFTYPE FIXNUM ()
   `(INTEGER ,MOST-NEGATIVE-FIXNUM ,MOST-POSITIVE-FIXNUM))

(DEFTYPE STREAM ()
   '(:DATATYPE STREAM))

(DEFTYPE FLOAT (&OPTIONAL LOW HIGH)
   (%RANGE-TYPE '(:DATATYPE IL:FLOATP)
          LOW HIGH))

(DEFTYPE FUNCTION ()
   '(SATISFIES FUNCTIONP))

(DEFTYPE HASH-TABLE ()
   '(:DATATYPE IL:HARRAYP))

(DEFTYPE INTEGER (&OPTIONAL LOW HIGH)
   (%RANGE-TYPE '(SATISFIES INTEGERP)
          LOW HIGH `((,IL:MIN.INTEGER ,IL:MAX.INTEGER (SATISFIES INTEGERP))
                     (,IL:MIN.FIXP ,IL:MAX.FIXP (OR (SATISFIES IL:SMALLP)
                                                    (:DATATYPE IL:FIXP)))
                     (,IL:MIN.SMALLP ,IL:MAX.SMALLP (SATISFIES IL:SMALLP))
                     (0 1 (MEMBER 0 1)))))

(DEFTYPE KEYWORD ()
   '(SATISFIES KEYWORDP))

(DEFTYPE LIST (&OPTIONAL TYPE)
   (IF (EQ TYPE '*)
       '(OR NULL CONS)
       `(AND LIST (SATISFIES (LAMBDA (X)
                                    (EVERY #'(LAMBDA (ELEMENT)
                                                    (TYPEP ELEMENT ',TYPE))
                                           X))))))

(DEFTYPE LONG-FLOAT (&OPTIONAL LOW HIGH)
   `(FLOAT ,LOW ,HIGH))

(DEFTYPE MEMBER (&REST VALUES)
   `(SATISFIES (LAMBDA (X)
                      (MEMBER X ',VALUES))))

(DEFTYPE MOD (N)
   `(INTEGER 0 ,(1- N)))

(DEFTYPE NULL ()
   '(SATISFIES NULL))

(DEFTYPE NUMBER ()
   '(SATISFIES NUMBERP))

(DEFTYPE PACKAGE ()
   '(:DATATYPE PACKAGE))

(DEFTYPE SHORT-FLOAT (&OPTIONAL LOW HIGH)
   `(FLOAT ,LOW ,HIGH))

(DEFTYPE SIGNED-BYTE (&OPTIONAL S)
   (IF (EQ S '*)
       'INTEGER
       (LET ((SIZE (EXPT 2 (1- S))))
            `(INTEGER ,(- SIZE)
                    ,(1- SIZE)))))

(DEFTYPE STANDARD-CHAR ()
   '(SATISFIES STANDARD-CHAR-P))

(DEFTYPE STRING-CHAR ()
   '(AND CHARACTER (SATISFIES STRING-CHAR-P)))

(DEFTYPE SINGLE-FLOAT (&OPTIONAL LOW HIGH)
   `(FLOAT ,LOW ,HIGH))

(DEFTYPE SYMBOL ()
   '(:DATATYPE IL:LITATOM))

(DEFTYPE UNSIGNED-BYTE (&OPTIONAL S)
   (IF (EQ S '*)
       '(INTEGER 0 *)
       `(INTEGER 0 (,(EXPT 2 S)))))

(DEFTYPE RATIONAL (&OPTIONAL LOW HIGH)
   (%RANGE-TYPE '(OR RATIO INTEGER)
          LOW HIGH))

(DEFTYPE READTABLE ()
   '(:DATATYPE READTABLEP))

(DEFTYPE COMMON ()
   ;; This is a hack.  (You can tell, because it uses TYPE-OF.)  However, it is correct.  (Note that even though subtypep uses expanders, there is no
   ;; danger of  a loop because it quits when it reachs a satisfies clause.)
   `(SATISFIES (LAMBDA (OBJ)
                      (VALUES (SUBTYPEP (TYPE-OF OBJ)
                                     'COMMON)))))

(DEFTYPE COMPILED-FUNCTION ()
   '(SATISFIES COMPILED-FUNCTION-P))

(DEFTYPE SEQUENCE (&OPTIONAL TYPE)
   ;; Larry's dubious extension, that I can't remove because he wrote code that relies on it.  Actually the extension is somewhat useful, but confusing.
   ;; (it simulates the DECL facility for saying (LIST user-type).)
   (IF (EQ TYPE '*)
       '(OR VECTOR LIST)
       `(AND SEQUENCE (SATISFIES (LAMBDA (X)
                                        (EVERY #'(LAMBDA (ELEMENT)
                                                        (TYPEP ELEMENT ',TYPE))
                                               X))))))

;;; Array Types

(DEFTYPE ARRAY (&OPTIONAL ELEMENT-TYPE DIMENSIONS)

;;; This type definition should not return anything other than satisfies.  Other array types are determined in terms of this one, (for subtypep's sake) so this
;;; one must bottom out.

   (IF (TYPEP DIMENSIONS 'FIXNUM)
       (SETQ DIMENSIONS (MAKE-LIST DIMENSIONS :INITIAL-ELEMENT '*)))
   (IF (NOT (EQ ELEMENT-TYPE '*))
       (SETQ ELEMENT-TYPE (IL:%GET-CANONICAL-CML-TYPE ELEMENT-TYPE)))
   (COND
      ((EQ DIMENSIONS '*)
       (IF (EQ ELEMENT-TYPE '*)
           '(SATISFIES ARRAYP)
           `(SATISFIES (LAMBDA (X)
                              (AND (ARRAYP X)
                                   (EQUAL (ARRAY-ELEMENT-TYPE X)
                                          ',ELEMENT-TYPE))))))
      ((EQ (LENGTH DIMENSIONS)
           1)
       (LET ((SIZE (CAR DIMENSIONS)))
            (COND
               ((EQ ELEMENT-TYPE '*)
                (IF (EQ SIZE '*)
                    '(SATISFIES VECTORP)
                    `(SATISFIES (LAMBDA (X)
                                       (AND (VECTORP X)
                                            (EQ (ARRAY-TOTAL-SIZE X)
                                                ,SIZE))))))
               ((EQ ELEMENT-TYPE 'STRING-CHAR)
                (IF (EQ SIZE '*)
                    '(SATISFIES STRINGP)
                    `(SATISFIES (LAMBDA (X)
                                       (AND (STRINGP X)
                                            (EQ (ARRAY-TOTAL-SIZE X)
                                                ,SIZE))))))
               ((OR (EQ ELEMENT-TYPE 'BIT)
                    (EQUAL ELEMENT-TYPE '(UNSIGNED-BYTE 1)))
                (IF (EQ SIZE '*)
                    '(SATISFIES BIT-VECTOR-P)
                    `(SATISFIES (LAMBDA (X)
                                       (AND (BIT-VECTOR-P X)
                                            (EQ (ARRAY-TOTAL-SIZE X)
                                                ,SIZE))))))
               (T  ;; vector of explicit element-type
                  `(SATISFIES (LAMBDA (X)
                                     (AND (VECTORP X)
                                          ,@(IF (NOT (EQ SIZE '*))
                                                `((EQ (ARRAY-TOTAL-SIZE X)
                                                      ,SIZE)))
                                          (EQUAL (ARRAY-ELEMENT-TYPE X)
                                                 ',ELEMENT-TYPE))))))))
      ((EVERY #'(LAMBDA (DIM)
                       (EQ DIM '*))
              DIMENSIONS)
       `(SATISFIES (LAMBDA (X)
                          (AND (ARRAYP X)
                               (EQ (ARRAY-RANK X)
                                   ,(LENGTH DIMENSIONS))
                               ,@(IF (NOT (EQ ELEMENT-TYPE '*))
                                     `((EQUAL (ARRAY-ELEMENT-TYPE X)
                                              ',ELEMENT-TYPE)))))))
      ((EVERY #'(LAMBDA (DIM)
                       (OR (EQ DIM '*)
                           (TYPEP DIM 'FIXNUM)))
              DIMENSIONS)
       `(SATISFIES (LAMBDA (X)
                          (AND (ARRAYP X)
                               (EQ (ARRAY-RANK X)
                                   ,(LENGTH DIMENSIONS))
                               ,@(DO ((DIM-SPEC DIMENSIONS (CDR DIM-SPEC))
                                      (DIM 0 (1+ DIM))
                                      FORMS)
                                     ((NULL DIM-SPEC)
                                      FORMS)
                                    (IF (NOT (EQ (CAR DIM-SPEC)
                                                 '*))
                                        (PUSH `(EQ (ARRAY-DIMENSION X ,DIM)
                                                   ,(CAR DIM-SPEC))
                                              FORMS)))
                               ,@(IF (NOT (EQ ELEMENT-TYPE '*))
                                     `((EQUAL (ARRAY-ELEMENT-TYPE X)
                                              ',ELEMENT-TYPE)))))))
      (T (ERROR "Bad (final) array type designator: ~S" `(ARRAY ,ELEMENT-TYPE ,DIMENSIONS)))))

(DEFTYPE VECTOR (&OPTIONAL ELEMENT-TYPE SIZE)
   ;; this type must be defined in terms of array so that subtypep can reason(?) about them.
   `(ARRAY ,ELEMENT-TYPE (,SIZE)))

(DEFTYPE STRING (&OPTIONAL SIZE)
   `(ARRAY STRING-CHAR (,SIZE)))
```

```
(DEFTYPE SIMPLE-STRING (&OPTIONAL SIZE)
  `(SIMPLE-ARRAY STRING-CHAR (,SIZE)))


(DEFTYPE SIMPLE-ARRAY (&OPTIONAL ELEMENT-TYPE DIMENSIONS)
   ;; Simple-array type expander
   (IF (TYPEP DIMENSIONS 'FIXNUM)
       (SETQ DIMENSIONS (MAKE-LIST DIMENSIONS :INITIAL-ELEMENT '*)))
   (IF (NOT (EQ ELEMENT-TYPE '*))
       (SETQ ELEMENT-TYPE (IL:%GET-CANONICAL-CML-TYPE ELEMENT-TYPE)))
   ;; at this point, dimensions is always a list of integers or *'s, and element-type is a canonical type.
   (COND
      ((EQ DIMENSIONS '*)
       (IF (EQ ELEMENT-TYPE '*)
           '(SATISFIES XCL:SIMPLE-ARRAY-P)
           `(SATISFIES (LAMBDA (X)
                         (AND (XCL:SIMPLE-ARRAY-P X)
                              (EQUAL (ARRAY-ELEMENT-TYPE X)
                                     ',ELEMENT-TYPE))))))
      ((EQ (LENGTH DIMENSIONS)
           1)
       (LET ((SIZE (CAR DIMENSIONS)))
            (COND
               ((EQ ELEMENT-TYPE 'STRING-CHAR)
                (IF (EQ SIZE '*)
                    '(SATISFIES SIMPLE-STRING-P)
                    `(SATISFIES (LAMBDA (X)
                                   (AND (SIMPLE-STRING-P X)
                                        (EQ (ARRAY-TOTAL-SIZE X)
                                            ,SIZE))))))
               ((OR (EQ ELEMENT-TYPE 'BIT)
                    (EQUAL ELEMENT-TYPE '(UNSIGNED-BYTE 1)))
                (IF (EQ SIZE '*)
                    '(SATISFIES SIMPLE-BIT-VECTOR-P)
                    `(SATISFIES (LAMBDA (X)
                                   (AND (SIMPLE-BIT-VECTOR-P X)
                                        (EQ (ARRAY-TOTAL-SIZE X)
                                            ,SIZE))))))
               ((EQ ELEMENT-TYPE T)
                (IF (EQ SIZE '*)
                    '(SATISFIES SIMPLE-VECTOR-P)
                    `(SATISFIES (LAMBDA (X)
                                   (AND (SIMPLE-VECTOR-P X)
                                        (EQ (ARRAY-TOTAL-SIZE X)
                                            ,SIZE))))))
               (T `(SATISFIES (LAMBDA (X)
                                 (AND (XCL:SIMPLE-ARRAY-P X)
                                      (EQ 1 (ARRAY-RANK X))
                                      ,@(IF (NOT (EQ SIZE '*))
                                            `((EQ (ARRAY-TOTAL-SIZE X)
                                                  ,SIZE)))
                                      ,@(IF (NOT (EQ ELEMENT-TYPE '*))
                                            `((EQUAL (ARRAY-ELEMENT-TYPE X)
                                                     ',ELEMENT-TYPE)))))))))
      ((EVERY #'(LAMBDA (DIM)
                       (EQ DIM '*))
              DIMENSIONS)
       `(SATISFIES (LAMBDA (X)
                          (AND (XCL:SIMPLE-ARRAY-P X)
                               (EQ (ARRAY-RANK X)
                                   ,(LENGTH DIMENSIONS))
                               ,@(IF (NOT (EQ ELEMENT-TYPE '*))
                                     `((EQUAL (ARRAY-ELEMENT-TYPE X)
                                              ',ELEMENT-TYPE)))))))
      ((EVERY #'(LAMBDA (DIM)
                       (OR (EQ DIM '*)
                           (TYPEP DIM 'FIXNUM)))
              DIMENSIONS)
       `(SATISFIES (LAMBDA (X)
                          (AND (XCL:SIMPLE-ARRAY-P X)
                               (EQ (ARRAY-RANK X)
                                   ,(LENGTH DIMENSIONS))
                               ,@(DO ((DIM-SPEC DIMENSIONS (CDR DIM-SPEC))
                                      (DIM 0 (1+ DIM))
                                      FORMS)
                                     ((NULL DIM-SPEC)
                                      FORMS)
                                    (IF (NOT (EQ (CAR DIM-SPEC)
                                                 '*))
                                        (PUSH `(EQ (ARRAY-DIMENSION X ,DIM)
                                                   ,(CAR DIM-SPEC))
                                              FORMS)))
                               ,@(IF (NOT (EQ ELEMENT-TYPE '*))
                                     `((EQUAL (ARRAY-ELEMENT-TYPE X)
                                              ',ELEMENT-TYPE)))))))
      (T (ERROR "Bad (final) array type designator: ~S" `(SIMPLE-ARRAY ,ELEMENT-TYPE ,DIMENSIONS)))))

(DEFTYPE SIMPLE-VECTOR (&OPTIONAL SIZE)
   `(SIMPLE-ARRAY T (,SIZE)))


(DEFTYPE BIT-VECTOR (&OPTIONAL SIZE)
   `(ARRAY (UNSIGNED-BYTE 1)
           (,SIZE)))


(DEFTYPE SIMPLE-BIT-VECTOR (&OPTIONAL SIZE)
   `(SIMPLE-ARRAY (UNSIGNED-BYTE 1)
           (,SIZE)))


;;; Fast predicates for typep


(XCL:DEF-DEFINE-TYPE TYPEP "Typep evaluator for a type")

(XCL:DEFDEFINER DEFTYPEP TYPEP (NAME TYPE-ARGS OBJECT-ARG &BODY BODY)

;;; The comment below is not necessarily true for deftype, so until the PavCompiler groks deftype, leave the eval-when alone.
   ;; The EVAL-WHEN below should be a PROGN as soon as the old ByteCompiler/COMPILE-FILE hack is done away with.  The PavCompiler
   ;; understands DEFMACRO's correctly and doesn't side-effect the environment.
   (UNLESS (AND NAME (SYMBOLP NAME))
          (ERROR "Illegal name used in DEFTYPEP: ~S" NAME))
   (MULTIPLE-VALUE-BIND (PARSED-BODY DECLS DOCSTRING)
       (IL:PARSE-DEFMACRO TYPE-ARGS 'SI::%$$TYPE-ARGS BODY NAME NIL :DEFAULT-DEFAULT ''* :PATH
              'SI::%$$TYPE-ARGS)
     (LET ((TYPEP-NAME (XCL:PACK (LIST "typep-evaluate-" NAME)
                              (SYMBOL-PACKAGE NAME))))
          ;; the eval-when insures  that the functions in the hash table are always  compiled
          `(PROGN (EVAL-WHEN (LOAD)
                         (SETF (SYMBOL-FUNCTION ',TYPEP-NAME)
                               #'(LAMBDA (SI::%$$OBJECT &OPTIONAL SI::%$$TYPE-ARGS)
                                        ,@DECLS
                                        (BLOCK ,NAME
                                            (LET ((,(CAR OBJECT-ARG)
                                                   SI::%$$OBJECT))
                                                 ,PARSED-BODY))))
                         (SETF (GETHASH ',NAME *TYPEP-HASH-TABLE*)
                               ',TYPEP-NAME)
                         ,@(AND DOCSTRING `((SETF (DOCUMENTATION ',NAME 'TYPEP)
                                                  ,DOCSTRING))))
                  (EVAL-WHEN (EVAL)
                         ;; With redefinition, clear the hash table
                         (SETF (GETHASH ',NAME *TYPEP-HASH-TABLE*)
                               NIL))))))


(DEFTYPEP LIST (&OPTIONAL ELEMENT-TYPE) (OBJECT)
   (AND (LISTP OBJECT)
        (IF (EQ ELEMENT-TYPE '*)
            T
            (DOLIST (L OBJECT T)
                (IF (NOT (TYPEP L ELEMENT-TYPE))
                    (RETURN NIL))))))


(DEFTYPEP SEQUENCE (&OPTIONAL ELEMENT-TYPE) (OBJECT)
   (AND (TYPEP OBJECT 'SEQUENCE)
        (IF (EQ ELEMENT-TYPE '*)
            T
            (EVERY #'(LAMBDA (S)
                            (TYPEP S ELEMENT-TYPE))
                   OBJECT))))


(DEFTYPEP MEMBER (&REST VALUES) (OBJECT)
   (MEMBER OBJECT VALUES))


(DEFTYPEP ARRAY (&OPTIONAL ELEMENT-TYPE DIMS) (OBJECT)
   (IF (NOT (EQ ELEMENT-TYPE '*))
       (SETQ ELEMENT-TYPE (IL:%GET-CANONICAL-CML-TYPE ELEMENT-TYPE)))
   (AND (ARRAYP OBJECT)
        (IF (EQ ELEMENT-TYPE '*)
            T
            (EQUAL (ARRAY-ELEMENT-TYPE OBJECT)
                   ELEMENT-TYPE))
        (COND
           ((EQ DIMS '*)
            T)
           ((TYPEP DIMS 'FIXNUM)
            (EQ (ARRAY-RANK OBJECT)
                DIMS))
           (T ;; Must be a cons
              (AND (EQ (ARRAY-RANK OBJECT)
                       (LENGTH DIMS))
                   (DO ((I 0 (1+ I))
                        (D DIMS (CDR D)))
                       ((NULL D)
                        T)
                     (IF (AND (TYPEP (CAR D)
                                     'FIXNUM)
                              (NOT (EQ (ARRAY-DIMENSION OBJECT I)
                                       (CAR D))))
                         (RETURN NIL))))))))


(DEFTYPEP SIMPLE-ARRAY (&OPTIONAL ELEMENT-TYPE DIMS) (OBJECT)
   (IF (NOT (EQ ELEMENT-TYPE '*))
       (SETQ ELEMENT-TYPE (IL:%GET-CANONICAL-CML-TYPE ELEMENT-TYPE)))
   (AND (XCL:SIMPLE-ARRAY-P OBJECT)
        (IF (EQ ELEMENT-TYPE '*)
            T
            (EQUAL (ARRAY-ELEMENT-TYPE OBJECT)
                   ELEMENT-TYPE))
        (COND
           ((EQ DIMS '*)
            T)
           ((TYPEP DIMS 'FIXNUM)
            (EQ (ARRAY-RANK OBJECT)
                DIMS))
           (T ;; Must be a cons
              (AND (EQ (ARRAY-RANK OBJECT)
                       (LENGTH DIMS))
                   (DO ((I 0 (1+ I))
                        (D DIMS (CDR D)))
                       ((NULL D)
                        T)
                     (IF (AND (TYPEP (CAR D)
                                     'FIXNUM)
                              (NOT (EQ (ARRAY-DIMENSION OBJECT I)
                                       (CAR D))))
                         (RETURN NIL))))))))


(DEFTYPEP VECTOR (&OPTIONAL ELEMENT-TYPE SIZE) (OBJECT)
   (IF (NOT (EQ ELEMENT-TYPE '*))
       (SETQ ELEMENT-TYPE (IL:%GET-CANONICAL-CML-TYPE ELEMENT-TYPE)))
   (AND (VECTORP OBJECT)
        (IF (EQ ELEMENT-TYPE '*)
            T
            (EQUAL (ARRAY-ELEMENT-TYPE OBJECT)
                   ELEMENT-TYPE))
        (IF (EQ SIZE '*)
            T
            (EQ (ARRAY-TOTAL-SIZE OBJECT)
                SIZE))))


(DEFTYPEP SIMPLE-VECTOR (&OPTIONAL SIZE) (OBJECT)
   (AND (SIMPLE-VECTOR-P OBJECT)
        (IF (EQ SIZE '*)
            T
            (EQ (ARRAY-TOTAL-SIZE OBJECT)
                SIZE))))


(DEFTYPEP COMPLEX (&OPTIONAL TYPE) (OBJECT)
   (AND (COMPLEXP OBJECT)
        (IF (EQ TYPE '*)
            T
            (AND (TYPEP (REALPART OBJECT)
                        TYPE)
                 (TYPEP (IMAGPART OBJECT)
                        TYPE)))))


(DEFTYPEP INTEGER (&OPTIONAL LOW HIGH) (OBJECT)
   (AND (INTEGERP OBJECT)
        (COND
           ((EQ LOW '*)
            T)
           ((CONSP LOW)
            (> OBJECT (CAR LOW)))
           (T (>= OBJECT LOW)))
        (COND
           ((EQ HIGH '*)
            T)
           ((CONSP HIGH)
            (> (CAR HIGH)
               OBJECT))
           (T (>= HIGH OBJECT)))))


(DEFTYPEP MOD (&OPTIONAL N) (OBJECT)
   (AND (INTEGERP OBJECT)
        (>= OBJECT 0)
        (IF (EQ N '*)
            T
            (> N OBJECT))))


(DEFTYPEP SIGNED-BYTE (&OPTIONAL S) (OBJECT)
   (AND (INTEGERP OBJECT)
        (IF (EQ S '*)
            T
            (LET ((BOUND (ASH 1 (1- S))))
                 (AND (>= OBJECT (- BOUND))
                      (> BOUND OBJECT))))))


(DEFTYPEP UNSIGNED-BYTE (&OPTIONAL S) (OBJECT)
   (AND (INTEGERP OBJECT)
        (>= OBJECT 0)
        (IF (EQ S '*)
            T
            (> (ASH 1 S)
               OBJECT))))


(DEFTYPEP RATIONAL (&OPTIONAL LOW HIGH) (OBJECT)
   (AND (RATIONALP OBJECT)
        (COND
           ((EQ LOW '*)
            T)
           ((CONSP LOW)
            (> OBJECT (CAR LOW)))
           (T (>= OBJECT LOW)))
        (COND
           ((EQ HIGH '*)
            T)
           ((CONSP HIGH)
            (> (CAR HIGH)
               OBJECT))
           (T (>= HIGH OBJECT)))))


(DEFTYPEP FLOAT (&OPTIONAL LOW HIGH) (OBJECT)
   (AND (FLOATP OBJECT)
        (COND
           ((EQ LOW '*)
            T)
           ((CONSP LOW)
            (> OBJECT (CAR LOW)))
           (T (>= OBJECT LOW)))
        (COND
           ((EQ HIGH '*)
            T)
           ((CONSP HIGH)
            (> (CAR HIGH)
               OBJECT))
           (T (>= HIGH OBJECT)))))


(DEFTYPEP STRING (&OPTIONAL SIZE) (OBJECT)
   (AND (STRINGP OBJECT)
        (IF (EQ SIZE '*)
            T
            (EQ (ARRAY-TOTAL-SIZE OBJECT)
                SIZE))))


(DEFTYPEP SIMPLE-STRING (&OPTIONAL SIZE) (OBJECT)
   (AND (SIMPLE-STRING-P OBJECT)
        (IF (EQ SIZE '*)
            T
            (EQ (ARRAY-TOTAL-SIZE OBJECT)
                SIZE))))
```

```
(DEFTYPEP BIT-VECTOR (&OPTIONAL SIZE) (OBJECT)
   (AND (BIT-VECTOR-P OBJECT)
        (IF (EQ SIZE '*)
            T
            (EQ (ARRAY-TOTAL-SIZE OBJECT)
                SIZE))))


(DEFTYPEP SIMPLE-BIT-VECTOR (&OPTIONAL SIZE) (OBJECT)
   (AND (SIMPLE-BIT-VECTOR-P OBJECT)
        (IF (EQ SIZE '*)
            T
            (EQ (ARRAY-TOTAL-SIZE OBJECT)
                SIZE))))


;;; for TYPE-OF Interlisp types that have different common Lisp names

(IL:PUTPROPS IL:CHARACTER CMLTYPE CHARACTER)

(IL:PUTPROPS IL:FIXP CMLTYPE BIGNUM)

(IL:PUTPROPS IL:FLOATP CMLTYPE SINGLE-FLOAT)

(IL:PUTPROPS IL:GENERAL-ARRAY CMLTYPE ARRAY)

(IL:PUTPROPS IL:LISTP CMLTYPE CONS)

(IL:PUTPROPS IL:LITATOM CMLTYPE SYMBOL)

(IL:PUTPROPS IL:ONED-ARRAY CMLTYPE ARRAY)

(IL:PUTPROPS IL:SMALLP CMLTYPE FIXNUM)

(IL:PUTPROPS IL:HARRAYP CMLTYPE HASH-TABLE)

(IL:PUTPROPS IL:TWOD-ARRAY CMLTYPE ARRAY)

(IL:PUTPROPS SYMBOL CMLSUBTYPE-DESCRIMINATOR SYMBOL-TYPE)

(IL:PUTPROPS ARRAY CMLSUBTYPE-DESCRIMINATOR ARRAY-TYPE)


;;; tell the filepkg what to do with the type-expander property

(IL:PUTPROPS :TYPE-EXPANDER IL:PROPTYPE IGNORE)

(IL:PUTPROPS IL:TYPE-EXPANDER IL:PROPTYPE IGNORE)


;;; Compiler options

(IL:PUTPROPS IL:CMLTYPES IL:FILETYPE COMPILE-FILE)

(IL:PUTPROPS IL:CMLTYPES IL:MAKEFILE-ENVIRONMENT (:READTABLE "XCL" :PACKAGE "LISP"))

(IL:DECLARE\: IL:DONTEVAL@LOAD IL:DOEVAL@COMPILE IL:DONTCOPY

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

(IL:LOCALVARS . T)
)
)

(IL:PUTPROPS IL:CMLTYPES IL:COPYRIGHT ("Venue & Xerox Corporation" 1985 1986 1987 1988 1990 1993 2024))
```

{MEDLEY}<sources>CMLTYPES.;1 28-Jun-2024 18:34:03

-- Listed on 30-Jun-2024 13:15:36 --

FUNCTION INDEX

TYPE INDEX

TYPEP INDEX

PROPERTY INDEX

OPTIMIZER INDEX

CONSTANT INDEX

DEFINER INDEX

DEFINE-TYPE INDEX

MACRO INDEX

SETF INDEX

VARIABLE INDEX