CommonLisp-Compatible Array Functions

File: <lispusers>CMLARRAY.tty
Revised: Feb 21, Jul 1, and Sep 28, 1983, by JonL White

The following functions are based on the definitions in the CommonLisp community, and provide many features lacking in Interlisp's ARRAY support, in particular multi-dimensional arrays, shared arrays, and super-fast accessing ("open-coding" of access with "unsafe" primitives). These definitions follow those from the "Excelsior" edition of the CommonLisp manual (5 August 1983, copyright Guy L. Steele Jr.) and some of the prose below is reproduced from the aforesaid "Excelsior" edition by permission.

Documentation Terminology

Not all options and features of the Common Lisp specifiction are implemented; descriptions of limitations are encolsed as notes within double square brackets. Furthermore, for the benefit of Interlisp-D users, some non-portable extensions have been made, and these are described as noted within single square brackets.

A number of functions take arguments in "keyword" format; this means that the arglist, after some point, alternates between an argument which names the meaning of the next argument and the next "real" argument. E.g., suppose COLORMYWORLD is a function with conceptually dozens of parameters, but which is typically called with only one or two of them set to a non-default value. Then you might see

(COLORMYWORLD SOMEBITMAP 'HUE BLUE 'DURATION 5HOURS)

where the first argument is "required", but all the others are optional and are obtained from the appropriate pair in the real arglist; in this example, one such "keyword" argument is called HUE, and it will be set to the value of the variable BLUE; also "coloring" will last for a duration of time found in the variable 5HOURS.

As a general rule, any symbol ("litatom" in the Interlisp sense) which has a "-" in its name, will have those "-"'s removed in the Interlisp incarnation (because of weird potential interaction with CLISP). Thus, MAKE-ARRAY in the CommonLisp manual becomes MAKEARRAY in the Interlisp world. Also, since Interlisp doesn't have &optional, &rest, or &key argument spreading, then any function with such an argument spectrum will be implemented as a no-spread lambda; curly brackets will enclose the name for a sequence of such arguments.

Creation of Arrays

An array, for the purpose of this documentation, is an instance of a new datatype (called CMLARRAY), and is not related to the previously-documented Interlisp array facility. By convention, all indexing in the CommonLisp world is 0-origin.

(MAKEARRAY <dimensionslst> ... {keyword-arguments} ...)

<dimensionslst> is a list of non-negative integers that are to be the dimensions of the array; the length of the list will be the rank, or dimensionality, of the array. Note that if <dimensionslst> is NIL, then a zero-dimensional array is created. For convenience when making a one-dimensional array, the single dimension may be provided as an integer rather than a list of one integer.

The keyword arguments are:

ELEMENTTYPE

- T, [or NIL, or POINTER] means that the elements are all general

Lisp "pointers"; this is the default.
- FIXNUM, [or FIXP, or CELL] for entries which are integers stored as 2's complement 32-bits [[31-bits in Interlisp/VAX, 36-bits in Interlisp-10]]
- FLONUM, [or FLOATP] entries are all 32-bit IEEE format floating-point [[36-bit pdp10 format for Interlisp-10]]

The following type specifiers are sub-types of INTEGER for which the accessing is much more efficient than for other random field sizes.
- (MOD 65536) [or DOUBLEBYTE; or additionally, for Interlisp-D only, WORD or SMALLPOSP] for 16-bit non-negative integers.
- (MOD 256) [or BYTE, or CHARACTER] for 8-bit non-negative integers.
- (MOD 16) [or NIBBLE] for 4-bit non-negative integers.
- (MOD 2), or BIT, for single-bit entries.

[[In general, the CommonLisp type heirarchy isn't supported; thus only the explicit names above will work. However, for Interlisp-D, as of 28-Sep-83, the type (MOD n) for any reasonable "n", is supported.]]

INITIALELEMENT
- Argument must be a quantity of the type specified by the ELEMENTTYPE argument, and is used to initalize all the entries of the array. Default is NIL for pointer type, and zero for numeric types.

INITIALCONTENTS
- If the array is zero-dimensional, the this specifies its contents; otherwise, it must a sequence whose length is equal to the first dimension, and each element thereof must be a nested structure for an array whose dimensions are the remaining dimensions, and so on.

DISPLACEDTO
- Argument will be an array whose linearized data segment will be shared with the one being "made"; see also the DISPLACEDOFFSET argument also.

DISPLACEDINDEXOFFSET
- When the data vector is being "shared", this specifies the offset from the origin of the data vector at which the new array will have its zero-origin.

The implementational constraints, as specified by the symbolic constants ARRAY-RANK-LIMIT, ARRAY-DIMENSION-LIMIT, and ARRAY-TOTAL-SIZE-LIMIT are: Ranks up to 63 are supported; each individual dimension of an array is constrained only to be a FIXP; in Interlisp-D, the total storage for the data block of an array may not exceed (2^16-5)*2^5 bits, and since a pointer requires 32 bits, then an array may contain up to 2^16-5 pointer elements, or 2^18-20 byte elements; Interlisp/VAX and Interlisp-10 may have other constraints.

[[The following limitations exist as of 9-MAY-84:
(1) the :FILL-POINTER keyword argument is not implemented at all;
(2) all arrays automatically have the :ADJUSTABLE property;
(3) the "nested structures" used to specify the :INITIAL-CONTENTS may be either another array of the same dimensionality, or else just lists of lists.
(4) the symbolic constants ARRAY-RANK-LIMIT, ARRAY-DIMENSION-LIMIT, and ARRAY-TOTAL-SIZE-LIMIT are not actually in the code. Just be aware of these limits by reading the documentation presented above.]]

Examples:

```
(MAKEARRAY 5)  -  Create a one-dimensional array of five elements
(MAKEARRAY '(3 4) 'ELEMENTTYPE '(MOD 256))  -- Create a two-
    dimensional array, 3 by 4, with 8-bit elements;
(SETQ A (MAKEARRAY '(4 2 3) 'INITIALCONTENTS
                   '(((a b c) (1 2 3))
                     ((d e f) (3 1 2))
```

```
                        ((g h i) (2 3 1))
                        ((j k l) (0 0 0)))))
    (MAKEARRAY '(4 2 3) 'INITIALCONTENTS A)
    (SETQ B (MAKEARRAY 107 'ELEMENTTYPE 'BIT))
    (SETQ C (MAKEARRAY   8 'ELEMENTTYPE 'BYTE
                           'DISPLACEDTO  A
                           'DISPLACEDINDEXOFFSET 51)
```

Thus B and C share their data portion; the second element of C, with index 1, is the byte obtained by concatening the bits with indices 67 through 74 of B. Note that element type specification of BYTE is equivalent to that in the first example of (MOD 256).

Compatibililty note: For multi-dimension arrays, both LispMachineLisp and FORTRAN store in column-major order; MacLisp stores arrays in row-major order.
A "row" is the collection of all elements obtained by holding all but the last index constant, and cycling through that index in order from 0 to its limit. [This package generally stores in row-major order, but there may be ocasional "gaps" in the index-sequencing if the ALIGNMENT options is used. See below for a description of the non-CommonLisp option ALIGNMENT.]

Accessing and Changing the Elements of an Array

The main primitive to access the elements of an array is called AREF (for "Array REFerence of element").

(AREF <array> ... {subscripts} ...)
    This returns the element of <array> specified by the subscripts (which are all the remaining arguments after <array>), and each subscript must be a non-negative integer less than the corresponding array dimension.

[The main primitive to change the contents of an array is called ASET (for "Array SET value of element"), modeled after the MIT Lisp Machine name. Ultimately, ASET will operate so as to translate into the same code that SETF would have produced thus Interlisp users need not fear that such code won't run on a "by-the-book" CommonLisp implementation.
    (ASET <newvalue> <array> ... {subscripts} ...)
Changes the contents of the element of <array> accessed by
    (AREF <array> ... {subscripts} ...)
to be <newvalue>.]

[[CommonLisp does not require separate names for the update functions; updating may always be specified by the SETF construct, which takes an access expression and a "new value" much the way Interlisp's CHANGE does. As of 28-Sep-83, there is no changetran entry for AREF, but it is expected to be added someday.]]

Informational Functions

(ARRAYELEMENTTYPE <array>)
    Returns a type specifier for the set of objects that can be stored in <array>. This set may be larger than the set requested when <array> was created; that is,
    (ARRAYELEMENTTYPE (MAKEARRAY 5 'ELEMENTTYPE '(MOD 8)))
could be (MOD 8), or BYTE, or (MOD 256), etc., or even POINTER.
[[as of 28-Sep-83, only the types enumerated above under MAKEARRAY are supported; there is no "coercion upwards" in order to find a type-specifier which could hold the elements of some non-enumerated type. Remember also that the Interlisp-D version supports (MOD n) for all integral n.]]

(ARRAYRANK <array>)
    Returns the number of dimensions (axes) of <array>; to parallel the indexing range, this is a zero-origin number and thus will be a non-negative

integer.

(ARRAYDIMENSION <array> <axis-number>)
    Returns the length of dimension number <axis-number> of <array> [which may be any kind of array, i.e., any instance of the \CMLARRAY datatype]; <axis-number> should be a non-negative integer less than the rank of <array>.

(ARRAYDIMENSIONS <array> <optionslst>)
    Returns a list whose elements are the dimensions of <array>. [The second argument, <optionslst>, is not a CommonLisp argument -- it is provided
in Interlisp so that one may specify the NOCOPY option. Default action is to return a copy of the internal dimensions list.]

(ARRAYTOTALSIZE <array> <in-bits-p>)
    Returns the total number of elements in <array>, calculated as the product
of all the dimensions. Roughly equivalent to
    (APPLY 'TIMES (ARRAYDIMENSIONS <array>))
Note that the total size of a zero-dimensional array is 1. [The second argument, <in-bits-p>, is not a CommonLisp argument -- it is provided in Interlisp in order to find out how much space is actually occupied by the array, including any gaps caused by the ALIGNMENT option.]

(ARRAYINBOUNDSP <array> ... {subscripts} ...)
    This predicate checks whether the subscripts are all legal subscripts for <array> and is true if they are; otherwise it is false. The subscripts
must be integers. The number of subscripts supplied must equal the rank of the array. E.g., if HA is a three-dimensional array, then
    (ARRAYINBOUNDSP HA 4 25 62)
makes the check such that (AREF HA 4 25 62) will not cause an illegal subscript error.

(ARRAYROWMAJORINDEX <array> ... {subscripts} ...)
    This function takes an array and valid subscripts for the array, and returns a single non-negative integer less than the total size of the array that identifies the accessed element in the row-major ordering of the elements.
The number of subscripts supplied must equal the rank of the arry. Each subscript must be a non-negative integer less than the corresponding array dimension. For a one-dimensional array, the result of this function always equals the supplied subscript. [However, if the ALIGNMENT option is used for
a multi-dimensional array in Interlisp-D, then the maximum linearized index will exceed the value returned by ARRAYTOTALSIZE. In such a case, the value
returned by (ARRAYTOTALSIZE <array> T) will be the field size per element times one plus the maximum linearized index. The quantity
    (fetch (CMLARRAY CMLIMAX) of <array>)
will always be this maximum linearized index.]

Changing the Dimensions of an Array

(ADJUSTARRAY <array> <dimensionslst>)
    Takes an array, and a list of dimensions just as with MAKEARRAY; the number of dimensions specified by <dimensionslst> must equal the rank of <array>. Returns <array>, whose components have been updated to conform to the new specifications (but if the new dimensions require more space in the block data area, this will cause a copying into a newly allocated block,
and <array> will be DISPLACEDTO to this new block.)
[[None of the keyword options mentioned in the CommonLisp manual are supported

as of 28-Sep-83.]]

Extensions to the Interlisp CommonLisp Array Functions

Although the following facilities aren't specified by the "Excelsior Edition",
they may be quite useful in Interlisp-D systems programming:

*) Filepkg com: CMLARRAYS is similar to the BITMAPS com -- namely
(CMLARRAYS <var1> <var2> <var3> . . .)
will save and restore the value of each global variable <vari>, assuming
it holds a CMLARRAY.

*) Additional ELEMENTTYPE values for MAKEARRAY:
- XPOINTER is like POINTER, except that the entries aren't reference
counted for the garbage collector; beware, beware!
- DISPLACEDTOBASE is like DISPLACEDTO except its value is
just a random pointer/address, rather than another CMLARRAY.
This way, one can use the CommonLisp array functions to access
parts of Interlisp-D's memory such as the screen bitmap.
- ALIGNMENT is for multi-dimensional arrays; each row may be
required to begin a multiple of some integer, with nulls or zeros
filling any space between "rows". For example, it might be that
a bitmap array must have the raster scans beginning on a word
boundary; since there are 16 bits/word, then an alignment of 16
would be used.

*) Functions Interfaceing to LISTPs:
(LISTARRAY <array> <startindex> <endindex>)
The second and third arguments are optional, and have meaning similar
to the corresponding arguments to SUBSTRING, but negative indicies
aren't allowed. Elements are selected from the <array> in row-major
order, and CONS's up into a list.
(FILLARRAY <array> <list> <startindex> <endindex>)
Similar to LISTARRAY, except that the elements of the <list> are
stored into the corresponding parts of the <array>. As a convenience,
if <list> isn't a LISTP, then it is converted into (LIST <list>);
furthermore, if there aren't enough elements in <list> to fill out the
range specified by <startindex> and <no.of.elements>, then the last
element of <list> is repeated until finished. Thus, for example, one
could fill an array with a single value by a construct like
(FILLARRAY SOMEARRAY (LIST THISVAL))

*) "Fast" accessing functions:
(PAREF <array> ... {subscripts} ...)
(NAREF <array> ... {subscripts} ...)
(LAREF <array> ... {subscripts} ...)
(16AREF <array> ... {subscripts} ...)
(8AREF <array> ... {subscripts} ...)
(4AREF <array> ... {subscripts} ...)
(1AREF <array> ... {subscripts} ...)
These are essentially the same as AREF, but have a consequence that, for
PAREF, <array> must be a POINTER or T array; and correspondingly, for
NAREF a FIXNUM array, for LAREF a FLONUM array, for 16AREF a
DOUBLEBYTE or (MOD 65536) array, for 8AREF a BYTE or (MOD 256)

array, for 4AREF a NIBBLE or (MOD 16) array, and for 1AREF a BIT or (MOD 2) array.

Furthermore, when compiled, these functions will be compiled "open", with little or no error checking, realizing orders of magnitude speed-up over AREF; however, *** there is no certification as to what kind of value will be returned should the subscripts be "out of range". To aid in debugging, the run-time code actually toggles on the global variable AREFSissyFLG, and when the flg is non-NIL, will call the function AREF instead of the fast-but-unchecked "in-line" accessing.

However, one may omit even these simple checks for "all out, no holds barred" code by prefixing a \ to these names (e.g. \PAREF ...)

*) "Fast" setting functions:

(PASET <newvalue> <array> ... {subscripts} ...)
(NASET <newvalue> <array> ... {subscripts} ...)
(LASET <newvalue> <array> ... {subscripts} ...)
(16ASET <newvalue> <array> ... {subscripts} ...)
(8ASET <newvalue> <array> ... {subscripts} ...)
(4ASET <newvalue> <array> ... {subscripts} ...)
(1ASET <newvalue> <array> ... {subscripts} ...)

These are essentially the same as ASET, but have a consequence that, for PASET, <array> must be a POINTER or T array; and correspondingly, for NASET a FIXNUM array, for LAREF a FLONUM array, for 16ASET a DOUBLEBYTE or (MOD 65536) array, for 8ASET a BYTE or (MOD 256) array, for 4ASET a NIBBLE or (MOD 16) array, and for 1ASET a BIT or (MOD 2) array.

Furthermore, when compiled, these functions will be compiled "open", with little or no error checking, realizing orders of magnitude speed-up over ASET; although some checking is performed to insure memory system integrity (i.e. that the word modified will actually be within the data block of the specified array), **** there is no certification as to which word will be clobbered should the subscripts be "out of range". As with AREF, the run-time code toggles on the global variable AREFSissyFLG, and will call the function ASET when the flg is non-NIL.

However, one may omit even these simple checks for "all out, no holds barred" code by prefixing a \ to these names (e.g. \PASET ...). If you use this option, there is no guarantee of memory integrity, and likely no one will want to listen to reports of any "system" bugs encountered while such "unsafe" options were being exercised.