8. Arrays

An *array* is a Lisp object that consists of a group of cells, each of which may contain an object. The individual cells are selected by numerical *subscripts*. The type predicate **arrayp** (page 12) can be used to test whether an object is an array.

The *rank* of an array (the number of dimensions which the array has) is the number of subscripts used to refer to one of the elements of the array. The rank may be any integer from zero to seven, inclusively. An array of rank zero has a single element which is addressed using no subscripts. An array of rank one is called a *vector*; the predicate **vectorp** (see page 12) tests whether an object is a vector. A series of functions called the generic sequence functions accept either a vector or a list as argument indiscriminantly (see chapter 9, page 188).

array-rank-limit *Constant*

A constant giving the upper limit on the rank of an array. It is 8, indicating that 7 is the highest possible rank.

The lowest value for any subscript is zero; the highest value is a property of the array. Each dimension has a size, which is the lowest number which is too great to be used as a subscript. For example, in a one-dimensional array of five elements, the size of the one and only dimension is five, and the acceptable values of the subscript are zero, one, two, three, and four.

array-dimension-limit *Constant*

Any one dimension of an array must be smaller than this constant.

The *total size* of an array is the number of elements in it. It is the product of the sizes of the dimensions of the array.

array-total-size-limit *Constant*

The total number of elements of any array must be smaller than this constant.

A vector can have a *fill pointer* which is a number saying how many elements of the vector are *active*. For many purposes, only that many elements (starting with element zero) are used.

The most basic primitive functions for handling arrays are: **make-array**, which is used for the creation of arrays, **aref**, which is used for examining the contents of arrays, and **aset**, which is used for storing into arrays.

An array is a regular Lisp object, and it is common for an array to be the binding of a symbol, or the car or cdr of a cons, or, in fact, an element of an array. There are many functions, described in this chapter, which take arrays as arguments and perform useful operations on them.

Another way of handling arrays, inherited from Maclisp, is to treat them as functions. In this case each array has a name, which is a symbol whose function definition is the array. Zetalisp supports this style by allowing an array to be *applied* to arguments, as if it were a function. The arguments are treated as subscripts and the array is referenced appropriately. The **store** special form (see page 187) is also supported. This kind of array referencing is considered to be obsolete and is slower than the usual kind. It should not be used in new programs.

8.1 Array Types

There are several types of arrays, which differ primarily in which kinds of elements they are allowed to hold. Some types of arrays can hold Lisp objects of any type; such arrays are called *general* arrays. The other types of array restrict the possible elements to a certain type, usually a numeric type. Arrays of these types are called *specialized* arrays, or *numeric* arrays if the elements must be numbers. For example, one array type permits only complex numbers with floating components to be stored in the array. Another permits only the numbers zero and one; Common Lisp calls these *bit arrays*. The contents of a black-and-white screen are stored in a bit array. Several predicates exist for finding out which of these classifications an array belongs to: **simple-vector-p** (page 13), **bit-vector-p**, **simple-bit-vector-p**, **stringp** (page 12), and **simple-string-p**.

The array types are known by a set of symbols whose names begin with **art-** (for 'ARray Type').

The most commonly used type is called **art-q**. An **art-q** array simply holds Lisp objects of any type.

Similar to the **art-q** type is the **art-q-list**. Like the **art-q**, its elements may be any Lisp object. The difference is that the **art-q-list** array doubles as a list; the function **g-l-p** takes an **art-q-list** array and returns a list whose elements are those of the array, and whose actual substance is that of the array. If you **rplaca** elements of the list, the corresponding element of the array is changed, and if you store into the array, the corresponding element of the list changes the same way. An attempt to **rplacd** the list causes a **sys:rplacd-wrong-representation-type** error, since arrays cannot implement that operation.

The most important type of specialized array is the *string*, which is a vector of character objects. Character strings are implemented by the **art-string** array type. Many important system functions, including **read**, **print**, and **eval**, treat **art-string** arrays very differently from the other kinds of arrays. There are also many functions specifically for operating on strings, described in chapter 10.

As viewed by Common Lisp programs, the elements of a string are character objects. As viewed by traditional programs, the elements are integers in the range 0 to 255. While most code still accesses strings in the traditional manner and gets integers out, the Common Lisp viewpoint is considered the correct one. See page 203 for a discussion of this conflict of conventions and its effect on programs.

An **art-fat-string** array is a character string with wider characters, containing 16 bits rather than 8 bits. The extra bits are ignored by many string operations, such as comparison, on these strings; typically they are used to hold font information.

There is a set of types called **art-1b**, **art-2b**, **art-4b**, **art-8b**, and **art-16b**; these names are short for '1 bit', '2 bits', and so on. Each element of an **art-*n*b** array is a non-negative fixnum, and only the least significant *n* bits are remembered in the array; all of the others are discarded.

Thus **art-1b** arrays store only 0 and 1, and if you store a 5 into an **art-2b** array and look at it later, you will find a 1 rather than a 5.

These arrays are used when it is known beforehand that the fixnums which will be stored are non-negative and limited in size to a certain number of bits. Their advantage over the **art-q** array is that they occupy less storage, because more than one element of the array is kept in a single machine word. (For example, 32 elements of an **art-1b** array or 2 elements of an **art-16b** array fit into one word).

There are also **art-32b** arrays which have 32 bits per element. Since fixnums only have 24 bits anyway, these are the same as **art-q** arrays except that they only hold fixnums. They are not compatible with the other "bit" array types and generally should not be used.

An **art-half-fix** array contains half-size fixnums. Each element of the array is a signed 16-bit integer; the range is from -32768 to 32767 inclusive.

The **art-float** array type is a special-purpose type whose elements are floats. When storing into such an array the value (any kind of number) is converted to a float, using the **float** function (see page 149). The advantage of storing floats in an **art-float** array rather than an **art-q** array is that the numbers in an **art-float** array are not true Lisp objects. Instead the array remembers the numerical value, and when it is **aref**'ed creates a Lisp object (a float) to hold the value. Because the system does special storage management for bignums and floats that are intermediate results, the use of **art-float** arrays can save a lot of work for the garbage collector and hence greatly increase performance. An intermediate result is a Lisp object passed as an argument, stored in a local variable, or returned as the value of a function, but not stored into a special variable, a non-**art-float** array, or list structure. **art-float** arrays also provide a locality of reference advantage over **art-q** arrays containing floats, since the floats are contained in the array rather than being separate objects probably on different pages of memory.

The **art-fps-float** array type is another special-purpose type whose elements are floats. The internal format of this array is compatible with the PDP-11/VAX single-precision floating-point format. The primary purpose of this array type is to interface with the FPS array processor, which can transfer data directly in and out of such an array.

Any type of number may be stored into an **art-fps-float** array, but it is, in effect, converted to a float, and then rounded off to the 24-bit precision of the PDP-11. If the magnitude of the number is too large, the largest valid floating-point number is stored. If the magnitude is too small, zero is stored.

When an element of an **art-fps-float** array is read, a new float is created containing the value, just as with an **art-float** array.

The **art-complex** array type is a special purpose type whose elements are arbitrary numbers, which may be complex numbers. (Most of the numeric array types can only hold real numbers.) As compared with an ordinary **art-q** array, **art-complex** provides an advantage in garbage collection similar to what **art-float** provides for floating point numbers.

The **art-complex-float** array type is a special purpose type whose elements are numbers (real or complex) whose real and imaginary parts are both floating point numbers. (If you store a non-floating-point number into the array, its real and imaginary parts are converted to floating point.) This provides maximum advantage in garbage collection if all the elements you wish to store in the array are numbers with floating point real and imaginary parts.

The **art-complex-fps-float** array type is similar to **art-complex-float** but each real or imaginary part is stored in the form used by the FPS array processor. Each element occupies two words, the first being the real part and the second being the imaginary part.

There are three types of arrays which exist only for the implementation of *stack groups*; these types are called **art-stack-group-head**, **art-special-pdl**, and **art-reg-pdl**. Their elements may be any Lisp object; their use is explained in the section on stack groups (see chapter 13, page 256).

array-types *Constant*
The value of **array-types** is a list of all of the array type symbols such as **art-q**, **art-4b**, **art-string** and so on. The values of these symbols are internal array type code numbers for the corresponding type.

array-types *array-type-code*
Given an internal numeric array-type code, returns the symbolic name of that type.

array-elements-per-q *Constant*
array-elements-per-q is an association list (see page 110) which associates each array type symbol with the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-elements-per-q *array-type-code*
Given the internal array-type code number, returns the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-bits-per-element *Constant*
The value of **array-bits-per-element** is an association list (see page 110) which associates each array type symbol with the number of bits of unsigned number it can hold, or **nil** if it can hold Lisp objects. This can be used to tell whether an array can hold Lisp objects or not.

array-bits-per-element *array-type-code*
Given the internal array-type code numbers, returns the number of bits per cell for unsigned numeric arrays, or **nil** for a type of array that can contain Lisp objects.

array-element-size *array*
Given an array, returns the number of bits that fit in an element of that array. For arrays that can hold general Lisp objects, the result is 25., based on the assumption that you will be storing fixnums in the array.

8.2 Extra Features of Arrays

Any array may have an *array leader*. An array leader is like a one-dimensional art-q array which is attached to the main array. So an array which has a leader acts like two arrays joined together. The leader can be stored into and examined by a special set of functions, different from those used for the main array: array-leader and store-array-leader. The leader is always one-dimensional, and always can hold any kind of Lisp object, regardless of the type or rank of the main part of the array.

Very often the main part of an array is used as a homogeneous set of objects, while the leader is used to remember a few associated non-homogeneous pieces of data. In this case the leader is not used like an array; each slot is used differently from the others. Explicit numeric subscripts should not be used for the leader elements of such an array; instead the leader should be described by a defstruct (see page 374).

By convention, element 0 of the array leader of an array is used to hold the number of elements in the array that are "active". When the zeroth element is used this way, it is called a *fill pointer*. Many array-processing functions recognize the fill pointer. For instance, if a string (an array of type art-string) has seven elements, but its fill pointer contains the value five, then only elements zero through four of the string are considered to be active; the string's printed representation is five characters long, string-searching functions stop after the fifth element, etc. Fill pointers are a Common Lisp standard, but the array leader which is the Lisp Machine's way of implementing them is not standard.

fill-pointer *array*

Returns the fill pointer of *array*, or nil if it does not have one. This function can be used with setf to set the array's fill pointer.

The system does not provide a way to turn off the fill-pointer convention; any array that has a leader must reserve element 0 for the fill pointer or avoid using many of the array functions.

Leader element 1 is used in conjunction with the "named structure" feature to associate a user-defined data type with the array; see page 390. Element 1 is treated specially only if the array is flagged as a named structure.

8.2.1 Displaced Arrays

The following explanation of *displaced arrays* is probably not of interest to a beginner; the section may be passed over without losing the continuity of the manual.

Normally, an array is represented as a small amount of header information, followed by the contents of the array. However, sometimes it is desirable to have the header information removed from the actual contents. One such occasion is when the contents of the array must be located in a special part of the Lisp Machine's address space, such as the area used for the control of input/output devices, or the bitmap memory which generates the TV image. Displaced arrays are also used to reference certain special system tables, which are at fixed addresses so the microcode can access them easily.

If you give **make-array** a fixnum or a locative as the value of the **:displaced-to** option, it creates a displaced array referring to that location of virtual memory and its successors. References to elements of the displaced array will access that part of storage, and return the contents; the regular **aref** and **aset** functions are used. If the array is one whose elements are Lisp objects, caution should be used: if the region of address space does not contain typed Lisp objects, the integrity of the storage system and the garbage collector could be damaged. If the array is one whose elements are bytes (such as an **art-4b** type), then there is no problem. It is important to know, in this case, that the elements of such arrays are allocated from the right to the left within the 32-bit words.

It is also possible to have an array whose contents, instead of being located at a fixed place in virtual memory, are defined to be those of another array. Such an array is called an *indirect array*, and is created by giving **make-array** an array as the value of the **:displaced-to** option. The effects of this are simple if both arrays have the same type; the two arrays share all elements. An object stored in a certain element of one can be retrieved from the corresponding element of the other. This, by itself, is not very useful. However, if the arrays have different rank, the manner of accessing the elements differs. Thus, creating a one-dimensional array of nine elements, indirected to a second, two-dimensional array of three elements by three, allows access to the elements in either a one-dimensional or a two-dimensional manner. Weird effects can be produced if the new array is of a different type than the old array; this is not generally recommended. Indirecting an **art-*m*b** array to an **art-*n*b** array does the obvious thing. For instance, if *m* is 4 and *n* is 1, each element of the first array contains four bits from the second array, in right-to-left order.

It is also possible to create an indirect array in such a way that when an attempt is made to reference it or store into it, a constant number is added to the subscript given. This number is called the *index-offset*. It is specified at the time the indirect array is created, by giving a fixnum to **make-array** as the value of the **:displaced-index-offset** option. The length of the indirect array need not be the full length of the array it indirects to; it can be smaller. Thus the indirect array can cover just a subrange of the original array. The **nsubstring** function (see page 216) creates such arrays. When using index offsets with multi-dimensional arrays, there is only one index offset; it is added in to the linearized subscript which is the result of multiplying each subscript by an appropriate coefficient and adding them together.

8.3 Constructing Arrays

vector &rest *elements*

Constructs and returns a vector (one-dimensional array) whose elements are the arguments given.

make-array *dimensions* &rest *options.*

This is the primitive function for making arrays. *dimensions* should be a list of fixnums which are the dimensions of the array; the length of the list is the rank of the array. For convenience you can specify a single fixnum rather than a list of one fixnum, when making a one-dimensional array.

options are alternating keywords and values. The keywords may be any of the following:

:area
: The value specifies in which area (see chapter 16, page 296) the array should be created. It should be either an area number (a fixnum), or **nil** to mean the default area.

:type
: The value should be a symbolic name of an array type; the most common of these is **art-q**, which is the default. The elements of the array are initialized according to the type: if the array is of a type whose elements may only be fixnums or floats, then every element of the array is initially 0 or 0.0; otherwise, every element is initially **nil**. See the description of array types on page 163. The value of the option may also be the value of a symbol which is an array type name (that is, an internal numeric array type code).

:element-type
: *element-type* is the Common Lisp way to control the type of array made. Its value is a Common Lisp type specifier (see section 2.3, page 14). The array type used is the most specialized which can allow as an element anything which fits the type specifier. For example, if *element-type* is **(mod 4)**, you get an **art-2b** array. If *element-type* is **(mod 3)**, you still get an **art-2b** array, that being the most restrictive which can store the numbers 0, 1 and 2. If **element-type** is **string-char**, you get a string.

:initial-value
:initial-element
: Specifies the value to be stored in each element of the new array. If it is not specified, it is **nil** for arrays that can hold arbitrary objects, or 0 or 0.0 for numeric arrays. **:initial-value** is obsolete.

:initial-contents
: Specifies the entire contents for the new array, as a sequence of sequences of sequences... Array element 1 3 4 of a three-dimensional array would be **(elt (elt (elt** *initial-contents* **1) 3) 4)**. Recall that a sequence is either a list or a vector, and vectors include strings.

:displaced-to
: If this is not **nil**, a *displaced* array is constructed. If the value is a fixnum or a locative, **make-array** creates a regular displaced array which refers to the specified section of virtual address space. If the value is an array, **make-array** creates an indirect array (see page 167).

:leader-length
: The value should be a fixnum. The array is made with a leader containing that many elements. The elements of the leader are initialized to **nil** unless the **:leader-list** option is given (see below).

:leader-list
: The value should be a list. Call the number of elements in the list *n*. The first *n* elements of the leader are initialized from successive elements of this list. If the **:leader-length** option is not specified, then the length of the leader is *n*. If the **:leader-length** option is given, and its value is greater than *n*, then the *n*th and following leader elements are initialized to **nil**. If its value is less than *n*, an error is signaled. The leader elements are filled in forward order; that is, the car of the list is stored in leader element 0, the cadr in element 1, and so on.

:fill-pointer The value should be a fixnum. The array is made with a leader containing at least one element, and this fixnum is used to initialize that first element.

Using the :fill-pointer option is equivalent to using :leader-list with a list one element long. It avoids consing the list, and is also compatible with Common Lisp.

:displaced-index-offset
If this is present, the value of the :displaced-to option should be an array, and the value should be a non-negative fixnum; it is made to be the index-offset of the created indirect array. (See page 167.)

:named-structure-symbol
If this is not nil, it is a symbol to be stored in the named-structure cell of the array. The array made is tagged as a named structure (see page 390.) If the array has a leader, then this symbol is stored in leader element 1 regardless of the value of the :leader-list option. If the array does not have a leader, then this symbol is stored in array element zero. Array leader slot 1, or array element 0, cannot be used for anything else in a named structure.

:adjustable-p In strict Common Lisp, a non-nil value for this keyword makes the array *adjustable*, which means that it is permissible to change the array's size with adjust-array (page 176). This is because other Lisp systems have multiple representations for arrays, one which is simple and fast to access, and another which can be adjusted. The Lisp Machine does not require two representations: any array's size may be changed, and this keyword is ignored.

Examples:

```
        ;; Create a one-dimensional array of five elements.
        (make-array 5)
        ;; Create a two-dimensional array,
        ;; three by four, with four-bit elements.
        (make-array '(3 4) :type 'art-4b)
        ;; Create an array with a three-element leader.
        (make-array 5 :leader-length 3)
        ;; Create an array containing 5 t's,
        ;; and a fill pointer saying the array is full.
        (make-array 5 :initial-value t :fill-pointer 5)
        ;; Create a named-structure with five leader
        ;; elements, initializing some of them.
        (setq b (make-array 20 :leader-length 5
                               :leader-list '(0 nil foo)
                               :named-structure-symbol 'bar))
        (array-leader b 0) => 0
        (array-leader b 1) => bar
        (array-leader b 2) => foo
        (array-leader b 3) => nil
        (array-leader b 4) => nil
```

make-array returns the newly-created array, and also returns, as a second value, the number of words allocated in the process of creating the array, i.e. the **%structure-total-size** of the array.

When make-array was originally implemented, it took its arguments in the following fixed pattern:

(make-array *area type dimensions*
 &optional *displaced-to leader*
 displaced-index-offset
 named-structure-symbol)

leader was a combination of the :leader-length and :leader-list options, and the list was in reverse order. This obsolete form is still supported so that old programs will continue to work, but the new keyword-argument form is preferred.

8.4 Accessing Array Elements

aref *array* &rest *subscripts*

Returns the element of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the rank of *array*.

cli:aref *array* &rest *subscripts*

The Common Lisp version of aref differs from the traditional one in that it returns a character object rather than an integer when *array* is a string. See chapter 10 for a discussion of the data type of string elements.

aset *x array* &rest *subscripts*

Stores *x* into the element of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the rank of *array*. The returned value is *x*.

aset is equivalent to
```
        (setf (aref array subscripts...) x)
```

aloc *array* &rest *subscripts*

Returns a locative pointer to the element-cell of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the rank of *array*. The array must not be a numeric array, since locatives to the middle of a numeric array are not allowed. See the explanation of locatives in chapter 14, page 267.

It is equivalent, and preferable, to write
```
        (locf (aref array subscripts...))
```

ar-1-force *array i*
as-1-force *value array i*
ap-1-force *array i*

These functions access an array with a single subscript regardless of how many dimensions the array has. They may be useful for manipulating arrays of varying rank, as an alternative to maintaining and updating lists of subscripts or to creating one-dimensional indirect arrays. ar-1-force refers to an element, as-1-force sets an element, and ap-1-force returns a locative to the element's cell.

In using these functions, you must pay attention to the order in which the array elements are actually stored. See section 8.11, page 182.

array-row-major-index *array* &rest *indices*

Calculates the cumulative index in *array* of the element at indices *indices*.
```
        (ar-1-force array
           (array-row-major-index array indices...))
```
is equivalent to (aref *array indices...*).

array-leader *array i*

array should be an array with a leader, and *i* should be a fixnum. This returns the *i* th element of *array*'s leader. This is analogous to **aref**.

store-array-leader *x array i*

array should be an array with a leader, and *i* should be a fixnum. *x* may be any object. *x* is stored in the *i* th element of *array*'s leader. store-array-leader returns *x*. This is analogous to **aset**.

It is equivalent, and preferable, to write
```
        (setf (array-leader array i) x)
```

ap-leader *array i*
Is equivalent to
`(locf (array-leader` *array i*`))`

The following array accessing functions generally need not be used by users.

ar-1 *array i*
ar-2 *array i j*
ar-3 *array i j k*
as-1 *x array i*
as-2 *x array i j*
as-3 *x array i j k*
ap-1 *array i*
ap-2 *array i j*
ap-3 *array i j k*
These are obsolete versions of aref, aset and aloc that only work for one-, two-, or three-dimensional arrays, respectively.

The compiler turns aref into ar-1, ar-2, etc. according to the number of subscripts specified, turns aset into as-1, as-2, etc., and turns aloc into ap-1, ap-2, etc. For arrays with more than three dimensions the compiler uses the slightly less efficient form since the special routines only exist for one, two and three dimensions. There is no reason for any program to call ar-1, as-1, ar-2, etc. explicitly; they are documented because there used to be such a reason, and many old programs use these functions. New programs should use aref, aset, and aloc.

A related function, provided only for Maclisp compatibility, is arraycall (page 187).

svref *vector index*
A special accessing function defined by Common Lisp to work only on simple general vectors: vectors with no fill pointer, not displaced, and not adjustable (see page 169). Some other Lisp systems open code svref so that it is faster than aref, but on the Lisp Machine svref is a synonym for cli:aref.

bit *bit-vector index*
sbit *bit-vector index*
char *bit-vector index*
schar *bit-vector index*
Special accessing functions defined to work only on bit vectors, only on simple bit vectors, only on strings, and only on simple strings, respectively. On the Lisp Machine they are all synonyms for cli:aref.

Here are the conditions signaled for various errors in accessing arrays.

sys:array-has-no-leader (sys:bad-array-mixin error) *Condition*
This is signaled on a reference to the leader of an array that doesn't have one. The condition instance supports the :array operation, which returns the array that was used.

The :new-array proceed-type is provided.

sys:bad-array-mixin *Condition Flavor*

This mixin is used in the conditions signaled by several kinds of problems pertaining to arrays. It defines prompting for the :new-array proceed type.

sys:array-wrong-number-of-dimensions (sys:bad-array-mixin error) *Condition*

This is signaled when an array is referenced (either reading or writing) with the wrong number of subscripts; for example, (aref "foo" 1 2).

The :array operation on the condition instance returns the array that was used. The :subscripts-used operation returns the list of subscripts used.

The :new-array proceed type is provided. It expects one argument, an array to use instead of the original one.

sys:subscript-out-of-bounds (error) *Condition*

This is signaled when there are the right number of subscripts but their values specify an element that falls outside the bounds of the array. The same condition is used by sys:%instance-ref, etc., when the index is out of bounds in the instance.

The condition instance supports the operations :object and :subscripts-used, which return the array or instance and the list of subscripts.

The :new-subscript proceed type is provided. It takes an appropriate number of subscripts as arguments. You should provide as many subscripts as there originally were.

sys:number-array-not-allowed (sys:bad-array-mixin error) *Condition*

This is signaled by an attempt to use aloc on a numeric array such as an art-1b array or a string. The :array operation and the :new-array proceed type are available.

8.5 Getting Information About an Array

array-type *array*

Returns the symbolic type of *array*.
Example:

```
        (setq a (make-array '(3 5)))
        (array-type a) => art-q
```

array-element-type *array*

Returns a type specifier which describes what elements could be stored in *array* (see section 2.3, page 14 for more about type specifiers). Thus, if *array* is a string, the value is string-char. If *array* is an art-1b array, the value is bit. If *array* is an art-2b array, the value is (mod 4). If array is an art-q array, the value is t (the type which all objects belong to).

array-length *array*
array-total-size *array*

array may be any array. This returns the total number of elements in *array*. For a one-dimensional array, this is one greater than the maximum allowable subscript. (But if fill pointers are being used, you may want to use array-active-length.)
Example:

```
        (array-length (make-array 3)) => 3
        (array-length (make-array '(3 5))) => 15
```

array-total-size is the Common Lisp name of this function.

array-active-length *array*

If *array* does not have a fill pointer, then this returns whatever (array-length *array*) would have. If *array* does have a fill pointer, array-active-length returns it. See the general explanation of the use of fill pointers on page 166.

array-rank *array*

Returns the number of dimensions of *array*.
Example:

```
        (array-rank (make-array '(3 5))) => 2
```

array-dimension *array n*

Returns the length of dimension *n* of *array*. Examples:

```
        (setq a (make-array '(2 3)))
        (array-dimension a 0) => 2
        (array-dimension a 1) => 3
```

array-dimension-n *n array*

array may be any kind of array, and *n* should be a fixnum. If *n* is between 1 and the rank of *array*, this returns the *n* th dimension of *array*. If *n* is 0, this returns the length of the leader of *array*; if *array* has no leader it returns nil. If *n* is any other value, this returns nil.

This function is obsolete; use array-dimension-n, whose calling sequence is cleaner.
Examples:

```
        (setq a (make-array '(3 5) :leader-length 7))
        (array-dimension-n 1 a) => 3
        (array-dimension-n 2 a) => 5
        (array-dimension-n 3 a) => nil
        (array-dimension-n 0 a) => 7
```

array-dimensions *array*

Returns a list whose elements are the dimensions of *array*.
Example:

```
        (setq a (make-array '(3 5)))
        (array-dimensions a) => (3 5)
```

Note: the list returned by (array-dimensions *x*) is equal to the cdr of the list returned by (arraydims *x*).

arraydims *array*

Returns a list whose first element is the symbolic name of the type of *array*, and whose remaining elements are its dimensions. *array* may be any array; it also may be a symbol whose function cell contains an array, for Maclisp compatibility (see section 8.14, page 186).

Example:

```
        (setq a (make-array '(3 5)))
        (arraydims a) => (art-q 3 5)
```

arraydims is for Maclisp compatibility only.

array-in-bounds-p *array* &rest *subscripts*

t if *subscripts* is a legal set of subscripts for *array*, otherwise nil.

array-displaced-p *array*

t if *array* is any kind of displaced array (including an indirect array), otherwise nil. *array* may be any kind of array.

array-indirect-p *array*

t if *array* is an indirect array, otherwise nil. *array* may be any kind of array.

array-indexed-p *array*

t if *array* is an indirect array with an index-offset, otherwise nil. *array* may be any kind of array.

array-index-offset *array*

Returns the index offset of *array* if it is an indirect array which has an index offset. Otherwise it returns nil. *array* may be any kind of array.

array-has-fill-pointer-p *array*

t if array has a fill pointer. It must have a leader and leader element 0 must be an integer. While array leaders are not standard Common Lisp, fill pointers are, and so is this function.

array-has-leader-p *array*

t if *array* has a leader, otherwise nil.

array-leader-length *array*

Returns the length of *array*'s leader if it has one, or nil if it does not.

adjustable-array-p *array*

According to Common Lisp, returns t if *array*'s size may be adjusted with adjust-array (see below). On the Lisp Machine, this function always returns t.

8.6 Changing the Size of an Array

adjust-array *array new-dimensions* &key *element-type initial-element initial-contents fill-pointer displaced-to displaced-index-offset*

Modifies various aspects of an array. *array* is modified in place if that is possible; otherwise, a new array is created and *array* is forwarded to it. In either case, *array* is returned. The arguments have the same names as arguments to make-array, and signify approximately the same thing. However:

element-type is just an error check. adjust-array cannot change the array type. If the array type of *array* is not what *element-type* would imply, you get an error.

If *displaced-to* is specified, the new array is displaced as specified by *displaced-to* and *displaced-index-offset*. If *array* itself was already displaced, it is modified in place provided that either *array* used to have an index offset and is supposed to continue to have one, or *array* had no index offset and is not supposed to have one.

Otherwise, if *initial-contents* was specified, it is used to set all the contents of the array. The old contents of *array* are irrelevant.

Otherwise, each element of *array* is copied forward into the new array to the slot with the same indices, if there is one. Any new slots whose indices were out of range in *array* are initialized to *initial-element*, or to nil or 0 if *initial-element* was not specified.

fill-pointer, if specified, is used to set the fill pointer of the array. Aside from this, the result has a leader with the same contents as the original *array*.

adjust-array is the only function in this section which is standard Common Lisp. According to Common Lisp, an array's dimensions can be adjusted only if the :adjustable option was specified to make-array with a non-nil value when the array was created (see page 169). The Lisp Machine does not distinguish adjustable and nonadjustable arrays; any array may be adjusted.

adjust-array-size *array new-size*

If *array* is a one-dimensional array, its size is changed to be *new-size*. If *array* has more than one dimension, its size (array-length) is changed to *new-size* by changing only the last dimension.

If *array* is made smaller, the extra elements are lost; if *array* is made bigger, the new elements are initialized in the same fashion as make-array (see page 167) would initialize them: either to nil or 0, depending on the type of array.

Example:

```
(setq a (make-array 5))
(aset 'foo a 4)
(aref a 4) => foo
(adjust-array-size a 2)
(aref a 4) => an error occurs
```

If the size of the array is being increased, adjust-array-size may have to allocate a new array somewhere. In that case, it alters *array* so that references to it will be made to the new array instead, by means of invisible pointers (see structure-forward, page 273). adjust-array-size returns the new array if it creates one, and otherwise it returns *array*. Be careful to be consistent about using the returned result of adjust-array-size, because you may end up holding two arrays which are not the same (i.e. not eq), but which share the same contents.

array-grow *array* &rest *dimensions*
Equivalent to (adjust-array *array dimensions*). This name is obsolete.

si:change-indirect-array *array type dimlist displaced-p index-offset*
Changes an indirect array *array*'s type, size, or target pointed at. *type* specifies the new array type, *dimlist* its new dimensions, *displaced-p* the target it should point to (an array, locative or fixnum), and *index-offset* the new offset in the new target.

array is returned.

8.7 Arrays Overlaid With Lists

These functions manipulate art-q-list arrays, which were introduced on page 163.

g-l-p *array*
array should be an art-q-list array. This returns a list which shares the storage of *array*.
Example:

```
(setq a (make-array 4 :type 'art-q-list))
(aref a 0) => nil
(setq b (g-l-p a)) => (nil nil nil nil)
(rplaca b t)
b => (t nil nil nil)
(aref a 0) => t
(aset 30 a 2)
b => (t nil 30 nil)
```

g-l-p stands for 'get list pointer'.

The following two functions work strangely, in the same way that store does, and should not be used in new programs.

get-list-pointer-into-array *array-ref*
The argument *array-ref* is ignored, but should be a reference to an art-q-list array by applying the array to subscripts (rather than by aref). This returns a list object which is a portion of the "list" of the array, beginning with the last element of the last array which has been called as a function.

get-locative-pointer-into-array *array-ref*

get-locative-pointer-into-array is similar to get-list-pointer-into-array, except that it returns a locative, and doesn't require the array to be art-q-list. Use **locf** of **aref** in new programs.

8.8 Adding to the End of an Array

vector-push *x array*
array-push *array x*

array must be a one-dimensional array which has a fill pointer and *x* may be any object. vector-push attempts to store *x* in the element of the array designated by the fill pointer, and increase the fill pointer by one. If the fill pointer does not designate an element of the array (specifically, when it gets too big). it is unaffected and vector-push returns **nil**; otherwise, the two actions (storing and incrementing) happen uninterruptibly, and vector-push returns the *former* value of the fill pointer, i.e. the array index in which it stored *x*. If the array is of type art-q-list, an operation similar to **nconc** has taken place, in that the element has been added to the list by changing the cdr of the formerly last element. The cdr-coding is updated to ensure this.

array-push is an old name for this function. vector-push is preferable because it takes arguments in an order like **push**.

vector-push-extend *x array* &optional *extension*
array-push-extend *array x* &optional *extension*

vector-push-extend is just like vector-push except that if the fill pointer gets too large, the array grows to fit the new element; it never "fails" the way vector-push does, and so never returns **nil**. *extension* is the number of elements to be added to the array if it needs to grow. It defaults to something reasonable, based on the size of the array.

array-push-extend differs only in the order of arguments,

vector-pop *array*
array-pop *array*

array must be a one-dimensional array which has a fill pointer. The fill pointer is decreased by one and the array element designated by the new value of the fill pointer is returned. If the new value does not designate any element of the array (specifically, if it had already reached zero), an error is caused. The two operations (decrementing and array referencing) happen uninterruptibly. If the array is of type art-q-list, an operation similar to **nbutlast** has taken place. The cdr-coding is updated to ensure this.

The two names are synonymous.

sys:fill-pointer-not-fixnum (sys:bad-array-mixin error) *Condition*

This is signaled when one of the functions in this section is used with an array whose leader element zero is not a fixnum. Most other array accessing operations simply assume that the array has no fill pointer in such a case, but these cannot be performed without a fill pointer.

The :array operation on the condition instance returns the array that was used. The :new-array proceed type is supported, with one argument, an array.

8.9 Copying an Array

The new functions **replace** (page 189) and **fill** (page 190) are useful ways to copy parts of arrays.

array-initialize *array value* &optional *start end*

Stores *value* into all or part of *array*. *start* and *end* are optional indices which delimit the part of *array* to be initialized. They default to the beginning and end of the array.

This function is by far the fastest way to do the job.

fillarray *array x*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. It can also be **nil**, in which case an array of type **art-q** is created. There are two forms of this function, depending on the type of *x*.

If *x* is a list, then **fillarray** fills up *array* with the elements of *list*. If *x* is too short to fill up all of *array*, then the last element of *x* is used to fill the remaining elements of *array*. If *x* is too long, the extra elements are ignored. If *x* is **nil** (the empty list), *array* is filled with the default initial value for its array type (**nil** or **0**).

If *x* is an array (or, for Maclisp compatibility, a symbol whose function cell contains an array), then the elements of *array* are filled up from the elements of *x*. If *x* is too small, then the extra elements of *array* are not affected.

If *array* is multi-dimensional, the elements are accessed in row-major order: the last subscript varies the most quickly. The same is true of *x* if it is an array.

fillarray returns *array*; or, if *array* was **nil**, the newly created array.

listarray *array* &optional *limit*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. **listarray** creates and returns a list whose elements are those of *array*. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array* are used, and so the maximum length of the returned list is *limit*.

If *array* is multi-dimensional, the elements are accessed in row-major order: the last subscript varies the most quickly.

list-array-leader *array* &optional *limit*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. **list-array-leader** creates and returns a list whose elements are those of *array*'s leader. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array*'s leader are used, and so the maximum length of the returned list is *limit*. If *array* has no leader, **nil** is returned.

copy-array-contents *from to*

from and *to* must be arrays. The contents of *from* is copied into the contents of *to*, element by element. If *to* is shorter than *from*, the rest of *from* is ignored. If *from* is shorter than *to*, the rest of *to* is filled with nil, 0 or 0.0 according to the type of array. This function always returns t.

The entire length of *from* or *to* is used, ignoring the fill pointers if any. The leader itself is not copied.

copy-array-contents works on multi-dimensional arrays. *from* and *to* are linearized subscripts, and elements are taken in row-major order.

copy-array-contents-and-leader *from to*

Like copy-array-contents, but also copies the leader of *from* (if any) into *to*.

copy-array-portion *from-array from-start from-end to-array to-start to-end*

The portion of the array *from-array* with indices greater than or equal to *from-start* and less than *from-end* is copied into the portion of the array *to-array* with indices greater than or equal to *to-start* and less than *to-end*, element by element. If there are more elements in the selected portion of *to-array* than in the selected portion of *from-array*, the extra elements are filled with the default value as by copy-array-contents. If there are more elements in the selected portion of *from-array*, the extra ones are ignored. Multi-dimensional arrays are treated the same way as copy-array-contents treats them. This function always returns t.

%blt and %blt-typed (page 280) are often useful for copying parts of arrays. They can be used to shift a part of an array either up or down.

8.10 Bit Array Functions

These functions perform bitwise boolean operations on the elements of arrays.

bit-and *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-ior *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-xor *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-eqv *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-nand *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-nor *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-andc1 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-andc2 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-orc1 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-orc2 *bit-array-1 bit-array-2* &optional *result-bit-array*

Perform boolean operations element by element on bit arrays. The arguments must match in their size and shape, and all of their elements must be integers. Corresponding elements of *bit-array-1* and *bit-array-2* are taken and passed to one of **logand**, **logior**, etc. to get an element of the result array.

If the third argument is non-nil, the result bits are stored into it, modifying it destructively. If it is t, the results are stored in *bit-array-1*. Otherwise a new array of the same type as *bit-array-1* is created and used for the result. In any case, the value returned is the array where the results are stored.

These functions were introduced for the sake of Common Lisp, which defines them only when all arguments are specialized arrays that hold only zero or one. In the Lisp machine, they accept not only such arrays (art-1b arrays) but any arrays whose elements are integers.

bit-not *bit-array* &optional *result-bit-array*

Performs lognot on each element of *bit-array* to get an element of the result. If *result-bit-array* is non-nil, the result elements are stored in that; it must match *bit-array* in size and shape. Otherwise, a new array of the same type as *bit-array* is created and used to hold the result. The value of bit-not is the array where the results are stored.

bitblt *alu width height from-array from-x from-y to-array to-x to-y*

from-array and *to-array* must be two-dimensional arrays of bits or bytes (art-1b, art-2b, art-4b, art-8b, art-16b, or art-32b). bitblt copies a rectangular portion of *from-array* into a rectangular portion of *to-array*. The value stored can be a Boolean function of the new value and the value already there, under the control of *alu* (see below). This function is most commonly used in connection with raster images for TV displays.

The top-left corner of the source rectangle is (ar-2-reverse *from-array from-x from-y*). The top-left corner of the destination rectangle is (ar-2-reverse *to-array to-x to-y*). *width* and *height* are the dimensions of both rectangles. If *width* or *height* is zero, bitblt does nothing. The *x* coordinates and *width* are used as the second dimension of the array, since the horizontal index is the one which varies fastest in the screen buffer memory and the array's last index varies fastest in row-major order.

from-array and *to-array* are allowed to be the same array. bitblt normally traverses the arrays in increasing order of *x* and *y* subscripts. If *width* is negative, then (abs *width*) is used as the width, but the processing of the *x* direction is done backwards, starting with the highest value of *x* and working down. If *height* is negative it is treated analogously. When bitblt'ing an array to itself, when the two rectangles overlap, it may be necessary to work backwards to achieve effects such as shifting the entire array downwards by a certain number of rows. Note that negativity of *width* or *height* does not affect the (*x*, *y*) coordinates specified by the arguments, which are still the top-left corner even if bitblt starts at some other corner.

If the two arrays are of different types, bitblt works bit-wise and not element-wise. That is, if you bitblt from an art-2b array into an art-4b array, then two elements of the *from-array* correspond to one element of the *to-array*.

If bitblt goes outside the bounds of the source array, it wraps around. This allows such operations as the replication of a small stipple pattern through a large array. If bitblt goes outside the bounds of the destination array, it signals an error.

If *src* is an element of the source rectangle, and *dst* is the corresponding element of the destination rectangle, then **bitblt** changes the value of *dst* to **(boole** *alu src dst***)**. See the **boole** function (page 152). There are symbolic names for some of the most useful *alu* functions; they are **tv:alu-seta** (plain copy), **tv:alu-ior** (inclusive or), **tv:alu-xor** (exclusive or), and **tv:alu-andca** (and with complement of source).

bitblt is written in highly-optimized microcode and goes very much faster than the same thing written with ordinary **aref** and **aset** operations would. Unfortunately this causes **bitblt** to have a couple of strange restrictions. Wrap-around does not work correctly if *from-array* is an indirect array with an index-offset. **bitblt** signals an error if the second dimensions of *from-array* and *to-array* are not both integral multiples of the machine word length. For **art-1b** arrays, the second dimension must be a multiple of 32., for **art-2b** arrays it must be a multiple of 16., etc.

8.11 Order of Array Elements

Currently, multi-dimensional arrays are stored in row-major order, as in Maclisp., and as specified by Common Lisp. This means that successive memory locations differ in the last subscript. In older versions of the system, arrays were stored in column-major order.

Most user code has no need to know about which order array elements are stored in. There are three known reasons to care: use of multidimensional indirect arrays; paging efficiency (if you want to reference every element in a multi-dimensional array and move linearly through memory to improve locality of reference, you must vary the last subscript fastest in row-major order); and access to the TV screen or to arrays of pixels copied to or from the screen with **bitblt**. The latter is the most important one.

The bits on the screen are actually stored in rows, which means that the dimension that varies fastest has to be the horizontal position. As a result, if arrays are stored in row-major order, the horizontal position must be the second subscript, but if arrays are stored in column-major order, the horizontal position must be the first subscript. To ease the conversion of code that uses arrays of pixels, several bridging functions are provided:

make-pixel-array *width height* &rest *options*
: This is like **make-array** except that the dimensions of the array are *width* and *height*, in whichever order is correct. *width* is used as the dimension in the subscript that varies fastest in memory, and *height* as the other dimension. *options* are passed along to **make-array** to specify everything but the size of the array.

pixel-array-width *array*
: Returns the extent of *array*, a two-dimensional array, in the dimension that varies faster through memory. For a screen array, this is always the width.

pixel-array-height *array*
: Returns the extent of *array*, a two-dimensional array, in the dimension that varies slower through memory. For a screen array, this is always the height.

ar-2-reverse *array horizontal-index vertical-index*

Returns the element of *array* at *horizontal-index* and *vertical-index*. *horizontal-index* is used as the subscript in whichever dimension varies faster through memory.

as-2-reverse *newvalue array horizontal-index vertical-index*

Stores *newvalue* into the element of *array* at *horizontal-index* and *vertical-index*. *horizontal-index* is used as the subscript in whichever dimension varies faster through memory.

Code that was written before the change in order of array indices can be converted by replacing calls to make-array, array-dimension, aref and aset with these functions. It can then work either in old systems or in new ones. In more complicated circumstances, you can facilitate conversion by writing code which tests this variable.

sys:array-index-order *Constant*

This is t in more recent system versions which store arrays in row-major order (last subscript varies fastest). It is nil in older system versions which store arrays in column-major order.

8.12 Matrices and Systems of Linear Equations

The functions in this section perform some useful matrix operations. The matrices are represented as two-dimensional Lisp arrays. These functions are part of the mathematics package rather than the kernel array system, hence the 'math:' in the names.

math:multiply-matrices *matrix-1 matrix-2* &optional *matrix-3*

Multiplies *matrix-1* by *matrix-2*. If *matrix-3* is supplied, multiply-matrices stores the results into *matrix-3* and returns *matrix-3*, which should be of exactly the right dimensions for containing the result of the multiplication; otherwise it creates an array to contain the answer and returns that. All matrices must be either one- or two-dimensional arrays, and the first dimension of *matrix-2* must equal the second dimension of *matrix-1*.

math:invert-matrix *matrix* &optional *into-matrix*

Computes the inverse of *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result and returns that. *matrix* must be two-dimensional and square. The Gauss-Jordan algorithm with partial pivoting is used. Note: if you want to solve a set of simultaneous equations, you should not use this function; use math:decompose and math:solve (see below).

math:transpose-matrix *matrix* &optional *into-matrix*

Transposes *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result and returns that. *matrix* must be a two-dimensional array. *into-matrix*, if provided, must be two-dimensional and have exactly the right dimensions to hold the transpose of *matrix*.

math:determinant *matrix*

Returns the determinant of *matrix*. *matrix* must be a two-dimensional square matrix.

The next two functions are used to solve sets of simultaneous linear equations. math:decompose takes a matrix holding the coefficients of the equations and produces the LU decomposition; this decomposition can then be passed to math:solve along with a vector of right-hand sides to get the values of the variables. If you want to solve the same equations for many different sets of right-hand side values, you only need to call math:decompose once. In terms of the argument names used below, these two functions exist to solve the vector equation $A\ x = b$ for x. A is a matrix. b and x are vectors.

math:decompose *a* &optional *lu ps*

Computes the LU decomposition of matrix *a*. If *lu* is non-nil, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. The lower triangle of *lu*, with ones added along the diagonal, is L, and the upper triangle of *lu* is U, such that the product of L and U is *a*. Gaussian elimination with partial pivoting is used. The *lu* array is permuted by rows according to the permutation array *ps*, which is also produced by this function; if the argument *ps* is supplied, the permutation array is stored into it; otherwise, an array is created to hold it. This function returns two values, the LU decomposition and the permutation array.

math:solve *lu ps b* &optional *x*

This function takes the LU decomposition and associated permutation array produced by math:decompose and solves the set of simultaneous equations defined by the original matrix *a* given to math:decompose and the right-hand sides in the vector *b*. If *x* is supplied, the solutions are stored into it and it is returned; otherwise an array is created to hold the solutions and that is returned. *b* must be a one-dimensional array.

math:list-2d-array *array*

Returns a list of lists containing the values in *array*, which must be a two-dimensional array. There is one element for each row; each element is a list of the values in that row.

math:fill-2d-array *array list*

This is the opposite of math:list-2d-array. *list* should be a list of lists, with each element being a list corresponding to a row. *array*'s elements are stored from the list. Unlike fillarray (see page 179), if *list* is not long enough, math:fill-2d-array "wraps around", starting over at the beginning. The lists which are elements of *list* also work this way.

math:singular-matrix (sys:arithmetic-error error) *Condition*

This is signaled when any of the matrix manipulation functions in this section has trouble because of a singular matrix. (In some functions, such as math:determinant, a singular matrix is not a problem.)

The :matrix operation on the condition instance returns the matrix which is singular.

8.13 Planes

A *plane* is effectively an array whose bounds, in each dimension, are plus-infinity and minus-infinity; all integers are legal as indices. Planes may be of any rank. When you create a plane, you do not need to specify any size, just the rank. You also specify a default value. At that moment, every component of the plane has that value. As you can't ever change more than a finite number of components, only a finite region of the plane need actually be stored. When you refer to an element for which space has not actually been allocated, you just get the default value.

The regular array accessing functions don't work on planes. You can use **make-plane** to create a plane, **plane-aref** or **plane-ref** to get the value of a component, and **plane-aset** or **plane-store** to store into a component. **array-rank** works on planes.

A plane is actually stored as an array with a leader. The array corresponds to a rectangular, aligned region of the plane, containing all the components in which a **plane-store** has been done (and, usually, others which have never been altered). The lowest-coordinate corner of that rectangular region is given by the **plane-origin** in the array leader. The highest-coordinate corner can be found by adding the **plane-origin** to the **array-dimensions** of the array. The **plane-default** is the contents of all the elements of the plane that are not actually stored in the array. The **plane-extension** is the amount to extend a plane by in any direction when the plane needs to be extended. The default is 32.

If you never use any negative indices, then the **plane-origin** remains all zeroes and you can use regular array functions, such as **aref** and **aset**, to access the portion of the plane that is actually stored. This can be useful to speed up certain algorithms. In this case you can even use the **bitblt** function on a two-dimensional plane of bits or bytes, provided you don't change the **plane-extension** to a number that is not a multiple of 32.

make-plane *rank* &key *type default-value extension initial-dimensions initial-origins*

Creates and returns a plane. *rank* is the number of dimensions. The keyword arguments are

type	The array type symbol (e.g. **art-1b**) specifying the type of the array out of which the plane is made.
default-value	The default component value as explained above.
extension	The amount by which to extend the plane, as explained above.
initial-dimensions	**nil** or a list of integers whose length is *rank*. If not **nil**, each element corresponds to one dimension, specifying the width to allocate the array initially in that dimension.
initial-origins	**nil** or a list of integers whose length is *rank*. If not **nil**, each element corresponds to one dimension, specifying the smallest index in that dimension for which storage should initially be allocated.

Example:

```
        (make-plane 2 :type 'art-4b :default-value 3)
```

creates a two-dimensional plane of type **art-4b**, with default value 3.

plane-origin *plane*
A list of numbers, giving the lowest coordinate values actually stored.

plane-default *plane*
This is the contents of the infinite number of plane elements that are not actually stored.

plane-extension *plane*
The amount to extend the plane by, in any direction, when plane-store is done outside of the currently-stored portion.

plane-aref *plane* &rest *subscripts*
plane-ref *plane subscripts*
These two functions return the contents of a specified element of a plane. They differ only in the way they take their arguments; plane-aref wants the subscripts as arguments, while plane-ref wants a list of subscripts.

plane-aset *datum plane* &rest *subscripts*
plane-store *datum plane subscripts*
These two functions store *datum* into the specified element of a plane, extending it if necessary, and return *datum*. They differ only in the way they take their arguments; plane-aset wants the subscripts as arguments, while plane-store wants a list of subscripts.

8.14 Maclisp Array Compatibility

The functions in this section are provided only for Maclisp compatibility and should not be used in new programs.

Fixnum arrays do not exist (however, see Zetalisp's small-positive-integer arrays). Float arrays exist but you do not use them in the same way; no declarations are required or allowed. Un-garbage-collected arrays do not exist. Readtables and obarrays are represented as arrays, but Zetalisp does not use special array types for them. See the descriptions of read (page 531) and intern (page 645) for information about readtables and obarrays (packages). There are no 'dead" arrays, nor are Multics "external" arrays provided.

The arraycall function exists for compatibility but should not be used (see aref, page 170.)

Subscripts are always checked for validity, regardless of the value of *rset and whether the code is compiled or not. However, in a multi-dimensional array, an error is caused only if the subscripts would have resulted in a reference to storage outside of the array. For example, if you have a 2 by 7 array and refer to an element with subscripts 3 and 1, no error occurs despite the fact that the reference is invalid; but if you refer to element 1 by 100, an error occurs. In other words, subscript errors are caught if and only if they refer to storage outside the array; some errors are undetected, but they can only clobber (alter randomly) some other element of the same array, not something completely unpredictable.

loadarrays and dumparrays are not provided. However, arrays can be put into QFASL files; see section 17.8, page 317.

The *rearray function is not provided, since not all of its functionality is available in Zetalisp. Its most common uses are implemented by adjust-array-size.

In Maclisp, arrays are usually kept on the array property of symbols, and the symbols are used instead of the arrays. In order to provide some degree of compatibility for this manner of using arrays, the array, *array, and store functions are provided, and when arrays are applied to arguments, the arguments are treated as subscripts and apply returns the corresponding element of the array.

array &quote *symbol type* &eval &rest *dims*

Creates an art-q type array in default-array-area with the given dimensions. (That is, *dims* is given to make-array as its first argument.) *type* is ignored. If *symbol* is nil, the array is returned; otherwise, the array is put in the function cell of *symbol*, and *symbol* is returned.

***array** *symbol type* &rest *dims*

Is like array, except that all of the arguments are evaluated.

store *array-ref x* *Special form*

Stores *x* into the specified array element. *array-ref* should be a form which references an array by calling it as a function (aref forms are not acceptable). First *x* is evaluated, then *array-ref* is evaluated, and then the value of *x* is stored into the array cell last referenced by a function call, presumably the one in *array-ref*.

xstore *x array-ref*

This is just like store, but it is not a special form; this is because the arguments are in the other order. This function only exists for the compiler to compile the store special form into, and should never be used by programs.

arraycall *ignored array* &rest *subscripts*

(arraycall t *array sub1 sub2...*) is the same as (aref *array sub1 sub2...*). It exists for Maclisp compatibility.