15. Arrays

15.1 Array Concepts

15.1.1 Array Elements

An *array* contains a set of *objects* called *elements* that can be referenced individually according to a rectilinear coordinate system.

15.1.1.1 Array Indices

An *array element* is referred to by a (possibly empty) series of indices. The length of the series must equal the *rank* of the *array*. Each index must be a non-negative *fixnum* less than the corresponding *array dimension*. *Array* indexing is zero-origin.

15.1.1.2 Array Dimensions

An axis of an *array* is called a *dimension*.

Each *dimension* is a non-negative *fixnum*; if any dimension of an *array* is zero, the *array* has no elements. It is permissible for a *dimension* to be zero, in which case the *array* has no elements, and any attempt to *access* an *element* is an error. However, other properties of the *array*, such as the *dimensions* themselves, may be used.

15.1.1.2.1 Implementation Limits on Individual Array Dimensions

An *implementation* may impose a limit on *dimensions* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-dimension-limit**.

15.1.1.3 Array Rank

An *array* can have any number of *dimensions* (including zero). The number of *dimensions* is called the *rank*.

If the rank of an *array* is zero then the *array* is said to have no *dimensions*, and the product of the dimensions (see **array-total-size**) is then 1; a zero-rank *array* therefore has a single element.

15.1.1.3.1 Vectors

An *array* of *rank* one (i.e., a one-dimensional *array*) is called a *vector*.

15.1.1.3.1.1 Fill Pointers

A *fill pointer* is a non-negative *integer* no larger than the total number of *elements* in a *vector*. Not all *vectors* have *fill pointers*. See the *functions* **make-array** and **adjust-array**.

An *element* of a *vector* is said to be *active* if it has an index that is greater than or equal to zero, but less than the *fill pointer* (if any). For an *array* that has no *fill pointer*, all *elements* are considered *active*.

Only *vectors* may have *fill pointers*; multidimensional *arrays* may not. A multidimensional *array* that is displaced to a *vector* that has a *fill pointer* can be created.

15.1.1.3.2 Multidimensional Arrays

15.1.1.3.2.1 Storage Layout for Multidimensional Arrays

Multidimensional *arrays* store their components in row-major order; that is, internally a multidimensional *array* is stored as a one-dimensional *array*, with the multidimensional index sets ordered lexicographically, last index varying fastest.

15.1.1.3.2.2 Implementation Limits on Array Rank

An *implementation* may impose a limit on the *rank* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-rank-limit**.

15.1.2 Specialized Arrays

An *array* can be a *general array*, meaning each *element* may be any *object*, or it may be a *specialized array*, meaning that each *element* must be of a restricted *type*.

The phrasing "an *array specialized* to *type* <<*type*>>" is sometimes used to emphasize the *element type* of an *array*. This phrasing is tolerated even when the <<*type*>> is **t**, even though an *array specialized* to *type t* is a *general array*, not a *specialized array*.

The next figure lists some *defined names* that are applicable to *array* creation, *access*, and information operations.

```
adjust-array           array-has-fill-pointer-p  make-array
adjustable-array-p     array-in-bounds-p         svref
aref                   array-rank                upgraded-array-element-type
array-dimension        array-rank-limit          upgraded-complex-part-type
array-dimension-limit  array-row-major-index     vector
array-dimensions       array-total-size          vector-pop
array-displacement     array-total-size-limit    vector-push
array-element-type     fill-pointer              vector-push-extend
```

Figure 15-1. General Purpose Array-Related Defined Names

15.1.2.1 Array Upgrading

The *upgraded array element type* of a *type* T1 is a *type* T2 that is a *supertype* of T1 and that is used instead of T1 whenever T1 is used as an *array element type* for object creation or type discrimination.

During creation of an *array*, the *element type* that was requested is called the *expressed array element type*. The *upgraded array element type* of the *expressed array element type* becomes the *actual array element type* of the *array* that is created.

Type upgrading implies a movement upwards in the type hierarchy lattice. A *type* is always a *subtype* of its *upgraded array element type*. Also, if a *type* Tx is a *subtype* of another *type* Ty, then the *upgraded array element type* of Tx must be a *subtype* of the *upgraded array element type* of Ty. Two *disjoint types* can be *upgraded* to the same *type*.

The *upgraded array element type* T2 of a *type* T1 is a function only of T1 itself; that is, it is independent of any other property of the *array* for which T2 will be used, such as *rank*, *adjustability*, *fill pointers*, or displacement. The *function* **upgraded-array-element-type** can be used by *conforming programs* to predict how the *implementation* will *upgrade* a given *type*.

15.1.2.2 Required Kinds of Specialized Arrays

Vectors whose *elements* are restricted to *type* **character** or a *subtype* of **character** are called *strings*. *Strings* are of *type* **string**. The next figure lists some *defined names* related to *strings*.

Strings are *specialized arrays* and might logically have been included in this chapter. However, for purposes of readability most information about *strings* does not appear in this chapter; see instead Section 16 (Strings).

```
char                string-equal         string-upcase
make-string         string-greaterp      string/=
nstring-capitalize  string-left-trim     string<
nstring-downcase    string-lessp         string<=
nstring-upcase      string-not-equal     string=
schar               string-not-greaterp  string>
string              string-not-lessp     string>=
string-capitalize   string-right-trim
string-downcase     string-trim
```

Figure 15-2. Operators that Manipulate Strings

Vectors whose *elements* are restricted to *type* **bit** are called *bit vectors*. *Bit vectors* are of *type* **bit-vector**. The next figure lists some *defined names* for operations on *bit arrays*.

```
bit         bit-ior   bit-orc2
bit-and     bit-nand  bit-xor
bit-andc1   bit-nor   sbit
bit-andc2   bit-not
bit-eqv     bit-orc1
```

Figure 15-3. Operators that Manipulate Bit Arrays