CHAPTER 17 ARRAYS

17.1. make-array

Additional &key Arguments to `make-array`:

`:fatp`

(`t` or `nil`, defaults to `nil`). Affects storage allocation for arrays of element type string-char (strings). If `t`, storage is allocated to accomodate "fat" 16-bit (NS) characters. The default behavior is to allocate space for "thin" 8-bit characters. Fat characters can still be stored into a thin string (the string is automatically fattened), but it is more efficient to allocate it fat in the first place if it is known in advance that fat characters will be used.

`:extendable`

(`t` or `nil`, defaults to `nil`) similar to :adjustable but the only aspect of the array you can change is its size. This restriction allows for a more speed efficient implementation of extendable arrays, which is especially useful for those who make frequent use of vector-push-extend. `adjust-array` may be passed an extendable array. The predicate `extendable-array-p` is true for both adjustable and extendable arrays, but `adjustable-array-p` is true only for adjustable arrays.

`:read-only-p`

(`t` or `nil`, defaults to `nil`). If `t`, defines the array to be read-only, which is especially useful for displaced and displaced-to-base arrays. Read-only arrays may not be adjustable or extendable. An attempt to write into a read-only array does not cause an error, rather the array's storage block is copied before the write operation, so that the original storage block is unchanged.

`:displaced-to-base` *pointer*

pointer is a bare pointer or memory address. Allows you to displace an array directly to a memory storage block (like the screen bitmap). Should usually be used with `:read-only-p t` to prevent unintended changes to the original storage block.

In sum, Xerox Common Lisp `make-array` looks like (extensions are in bold):

```
make-array dimensions &key :element-type                [Function]
                           :initial-element
                           :initial-contents
                           :adjustable
                           :fill-pointer
                           :displaced-to
                           :displaced-index-offset
                           :fatp
                           :extendable
                           :read-only-p
                           :displaced-to-base
```

Limitations

Limits on rank and total size of arrays (i.e., values of the constants) are:

`array-rank-limit ⇒ 128`

`array-dimension-limit ⇒ 65534`

`array-total-size-limit ⇒ 65534`

Xerox Common Lisp Examples

Degenerate Arrays

There are two "degenerate" cases in making arrays, exemplified by the two following uses of make-array

1. `(setq a (make-array nil))`

   The variable a is now bound to a zero dimensional (non-empty) array. Others might call it a scalar. Note that dimensions is a required argument to make-array, so the NIL is given explicitly. The following relations hold:

   `(array-dimensions a)` returns `nil`

   `(array-total-size a)` returns 1

   `(array-rank a)` returns 0

The array a has storage for a single element (of the default element-type, t), which may be accessed by:

`(aref a)`

Note: There are *no* indices, since a is zero dimensional.

2. `(setq b (make-array '(0)))`

   The variable b is now bound to a one dimensional array, which has *no* elements. Others might call it an empty vector. The following relations hold:

   `(array-dimensions b)` returns '(0)

   `(array-dimension b 0)` returns 0

   `(array-total-size b)` returns 0

   `(array-rank b)` returns 1

   The array has *no* associated storage. `(aref b i)` is always an error regardless of the value of i, not because b is not one dimensional, but because the index i is always out of bounds.

It is possible to make empty arrays of higher dimensions as well; for example,

`(setq c (make-array '(2 3 4 0)))`

also creates an empty array, with no associated storage.

In summary, there are two sorts of degenerate arrays—zero dimensional or scalar arrays, for which aref is not an error, and empty arrays, arrays with at least one zero in their dimensions lists, for which `aref` is always an error, because there is no associated storage.

17.2. Array Access

Limitations

On page 291 in *Common Lisp: the Language* it states: "In some implementations of Common Lisp `svref` may be faster than `aref` in situations where it is applicable."

In Xerox Common Lisp there is no speed advantage in using `svref`.

17.3. Array Information

There are two additional predicates:

`(xcl:extendable-array-p` *array*`)` for &key `:extendable`

`(xcl:read-only-array-p` *array*`)` for &key `:read-only-p`

17.4. Functions on Arrays of Bits

On page 293 of *Common Lisp: the Language* it states: "In some implementations of Common Lisp, `bit` may be faster than `aref` in situations where it is applicable, and `sbit` may be similarly faster than `bit`."

In Xerox Common Lisp, there is no speed advantage in using `bit` or `sbit`.

17.5. Fill Pointers

The default value of extension is the value of the special variable
`xcl:*default-push-extension-size*`

which is initially 20.

17.6. Changing the Dimensions of an Array

Adjust-array—Additional &key Arguments

`:fatp`

(t or `nil`, defaults to `nil`). Affects storage allocation for arrays of element type string-char (strings). If `t`, storage is allocated to accomodate "fat" 16-bit (NS) characters. The default behavior is to allocate space for "thin" 8-bit characters. Fat characters can still be stored into a thin string (the string is automatically fattened), but it is more efficient to allocate it fat in the first place if it is known in advance that fat characters will be used.

`:displaced-to-base` *pointer*

pointer is a bare pointer or memory address. Allows you to displace an array directly to a memory storage block (like the screen bit map). Should usually be used with `:read-only-p t` to prevent unintended changes to the original storage block.

`adjust-array` looks like (XCL extensions are in bold):

```
adjust-array   array new-dimensions &key :element-type          [Function]
                                         :initial-element
                                         :initial-contents
                                         :fill-pointer
                                         :displaced-to
                                         :displaced-index-offset
                                         :fatp
                                         :displaced-to-base
```

Interpretation of adjust-array

Common Lisp: the Language is obscure on exactly what `adjust-array` does. Careful reading and discussions with Common Lisp implementors outside Xerox has led to the following interpretation.

The `adjust-array` function encounters three basic cases. These are listed in order of precedence, highest to lowest:

- Change size

  The array's total number of elements grows or shrinks and is copied to a new block of storage that the array is :displaced-to.

- New displacement

  If a displacement is provided, none of the original array contents appear in the resulting array.

- Undisplace an array

  If the original array was displaced to another array, then the original contents of the array disappear (since they are owned by the other array), and new storage of the appropriate size is created.

Interpretation of Standard &key Arguments to `adjust-array`

`:element-type` *type-specifier*

Does not change the type of elements in the array. Rather, it signals an error if the array could not hold elements of this type.

`:initial-element` *object*

Causes the newly adjusted array to have this element in all positions not otherwise filled.

`:fill-pointer` *integer-or-t*

The array being adjusted must be one-dimensional and have a fill pointer. If the value is `t` the fill pointer is set to the length of the vector, otherwise it must be an integer between zero and the size of the vector, inclusive.

`:initial-contents` *nested-sequences*

Causes the newly adjusted array to have this as its contents.

`:displaced-to array`

Causes the adjusted array to be a displaced array, one whose storage is shared with the given array.

`:displaced-index-offset` *integer*

Is a positive integer specifying the linear offset from the beginning of the "displaced to" array's elements, where this array will begin its addressing.

An Extension to Common Lisp—The Array Inspector

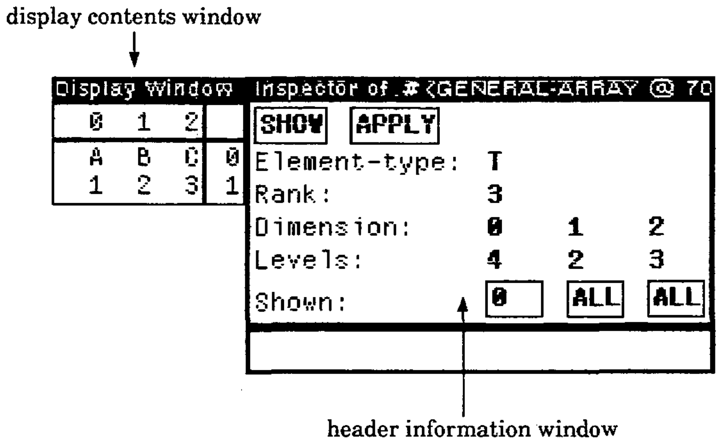
Inspecting Arrays

Xerox Common Lisp provides you with a way to examine the contents of arrays—the Array Inspector.

For example if you define an array a as follows:

```
(setq a (make-array '(4 2 3) :initial-contents
                    '(((a b c) (1 2 3))
                    ((d e f) (3 1 2))
                     ((g h i) (2 3 1))
                     ((j k l) (0 0 0))))
```

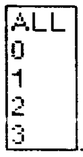
You can call the array inspector, with inspect a, to examine the contents of the array. The array inspector has two windows: a header information window and a content display window attached on the left. These two windows work in conjuction to display a slice of an array.



```
Display Window  Inspector of #<GENERAL-ARRAY @ 70
 0  1  2
 A  B  C   0   Element-type:  T
 1  2  3   1   Rank:          3
               Dimension:     0    1    2
               Levels:        4    2    3
               Shown:         0    ALL  ALL
```

The header information window displays the element type, total size, rank, and dimensionality of the array and controls which slice of the array's contents is shown in the contents display window. An array slice is determined by a set of restrictions on all the dimensions of the array. Selecting SHOW will display, in the header information window, the set of restrictions that describe the array slice being displayed in the contents window.

The restriction can be ALL (meaning "show every element of that dimension"), or some integer less than the value of that dimension of the array. If you want to change the slice being displayed you must change the restrictions that define which slice is displayed. To do this, move the cursor into one of the boxes on the line labeled "Shown:" and press the left mouse button. A small menu will pop up with the choices available for that dimension. For dimension 0

In the example above this menu looks like: 
.

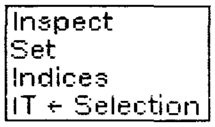
Select the new value for that dimension then, if you wish, you may change the values for the other dimensions in the same way.

After you have changed the restrictions for the dimensions selecting APPLY will cause the newly defined array slice to be displayed in the contents display window.

Imagining the three-dimensional array to be a series of planes, rows and columns, the above inspector shows a slice of the array created in the example above. To get this slice you would APPLY the restrictions:

plane	(dimension 0)	set to 0
rows	(dimension 1)	set to ALL
columns	(dimension 2)	set to ALL

The contents display window is capable of showing either a two- or one- or zero-dimensional slice of an array. The window is scrollable. A particular datum in the contents display may be selected with the left mouse button. After you select the datum, pressing the middle button in the contents display will pop up a menu that looks like:



This menu allows you to Inspect the datum, display the indices of the datum (in the box attached to the bottom of the header information window), set this position's value, or bind `il:it` to the selected datum.

CHAPTER 19 STRUCTURES

19.1. Introduction to Structures

Empty structures (those with no slots) are supported.

Common Lisp: the Language explicitly states that accessors, constructors, etc. are added to the current package, not the package containing the name of the structure. Xerox Common Lisp follows this standard. Also, accessors, constructors, etc., are defined as inline functions. If you don't want this behavior, Xerox Common Lisp provides an extension to the language that allows `(:inline nil)` in the argument list.

19.4. Defstruct Slot-Options

`:type`

No type-checking is done on typed slots when the slots contents are replaced.

9.5. Defstruct Options

The default structure type is unspecified in *Common Lisp: the Language*. Xerox Common Lisp uses the system datatype facilities and its microcode support.

`:conc-name`

While it is not made explicitly legal *Common Lisp: The Language* suggests that conc-names can be strings. Xerox Common Lisp supports that interpretation.

`:print-function`

In Xerox Common Lisp print functions are inherited. You can override this by specifying a print-function of `nil` for the subtype.

`:include`

Xerox Common Lisp does check for the incorrect use of access functions, by checking the type of the argument. Moreover, slot descriptions that are specified using `:include` are type-checked to ensure that the "shadowed" slot is a supertype of the new slot type. This is done in such a way that the resulting error is continuable, so that the user can disagree with `subtypep`.

An error is signaled if you "shadow" a slot name other than by using the `:include` option. For instance,

```
(defstruct super a)

(defstruct (sub (:include super)) (a 3) b)
```

is not legal, but the following is:

```
(defstruct (sub (:include super (a 3)) b)
```

Non-Standard Options

Our version of `defstruct` accepts a non-standard option, the `:inline` option, with the following syntax:

`(:inline` *categories*`)`

categories can be any of the following:

`nil` don't make optimizers for any `defstruct`-generated functions

`t` make optimizers for the default set of categories, `(:accessor :predicate)`

a list should contain only items from the following:

```
:accessor

:copier

:predicate

:boa-constructor

:constructor
```

and means to make optimizers for just the set of `defstruct`-generated functions in the categories given.

The default is `t`, or the list `(:accessor :predicate)`.

`xcl:*print-structure*` [*Variable*]

Structures of types without a user-specified `:print-function` normally print using the `#S` syntax described in *Common Lisp, the Language*. For example, a structure of type `foo` with slots `a` and `b` would print as follows:

```
#S(foo a nil b nil)
```

It is sometimes desirable, especially for structures with a large number of slots or with slot names in another package, to be able to use a more concise printing syntax, such as the following:

```
#<foo @ 52,14306>
```

In Xerox Lisp, the variable `xcl:*print-structure*` provides this flexibility. If `xcl:*print-structure*` is non-`nil`, structures of types without a user-specified `:print-function` will print using the `#S` syntax. Otherwise, those structures print using the more concise syntax shown above.

Note that `xcl:*print-structure*` is normally only examined by the default `:print-function`, though, of course, users writing their own `:print-functions` may choose also to assign some similar semantics to it.

[This page intentionally left blank]

CHAPTER 21 STREAMS

Xerox Lisp Extensions

The following functions have been added to Xerox Common Lisp.

Predicates

`xcl:synonym-stream-p stream` [*Function*]

Returns `t` if *stream* is a synonym stream.

`xcl:broadcast-stream-p` *stream* [*Function*]

Returns `t` if *stream* is a broadcast stream.

`xcl:concatenated-stream-p` *stream* [*Function*]

Returns `t` if *stream* is a concatenated stream.

`xcl:two-way-stream-p` *stream* [*Function*]

Returns `t` if *stream* is a two-way stream.

`xcl:echo-stream-p` *stream* [*Function*]

Returns t if *stream* is an echo stream.

`xcl:open-stream-p` *stream* [*Function*]

Returns `t` if *stream* is an open stream.

Accessors

`xcl:synonym-stream-symbol` *stream* [*Function*]

Returns the symbol for which *stream* is a synonym stream.

`xcl:broadcast-stream-streams` *stream* [*Function*]

Returns the streams (if any) that the broadcast stream *stream* broadcasts to.

`xcl:concatenated-stream-streams` *stream* [*Function*]

If *stream* is a concatenated stream, returns its remaining input streams.

`xcl:two-way-stream-input-stream` *stream* [*Function*]

Returns the two-way stream *stream*'s input side.

`xcl:two-way-stream-output-stream` *stream* [*Function*]

Returns the two-way stream *stream*'s output side.

`xcl:echo-stream-input-stream` *stream* [*Function*]

Returns the echo stream *stream*'s input side.

`xcl:echo-stream-output-stream` *stream* [*Function*]

Returns the echo stream *stream*'s output side.

Ambiguities

`close` specifies that the `:abort` option attempts to clean things up, to whatever extent is possible. In XCL, if the stream was open for output to a file, the file is deleted.

`close` of a synonym or broadcast stream has no effect on the underlying stream(s).

`streamp` returns its argument rather than `t`.

Cautions

If you pass a string containing fat NS characters to `make-string-input-stream`, the value of `file-position` for the stream will be wrong (off by a factor of 2). Similarly, if you read from a fat string using `with-input-from-string` with the `:index` option, the index variable will be off by a factor of 2.

CHAPTER 22 INPUT/OUTPUT

Ambiguities

The reader interprets the "potential number" of the form <octal digits>Q as an octal integer (same as #o<octal digits>), for compatibility with Interlisp. A potential number that is entirely numeric digits but illegal (e.g., "89" when `*read-base*` is 8) signals an error. All other potential numbers are taken to be symbols, without signalling an error.

Section 22.1.3. Macro Characters

The back-quote facility does not go to any trouble to create fresh list-structures unless it is necessary to do so. Thus, for example,

```
`(1 2 3)
```

is equivalent to

```
'(1 2 3)
```

not

```
(list 1 2 3)
```

Users needing to avoid sharing structure should use explicit calls to `list` or `copy-tree`.

Cautions

In this release, comma does not signal an error if used outside a backquote expression.

Section 22.1.4. Standard Dispatching Macro Character Syntax

In #+ and #- reader macros, the default package of symbols in the features expression is `keyword`. You can, of course, override the default by explicitly specifying package prefixes.

Section 22.1.6. What the Print Function Produces

Cautions

`*print-circle*` [*Variable*]

`*print-circle*` cannot be used to print large data structures containing more than 32K pointers.

Section 22.3.1. Output to Character Streams

`finish-output` is equivalent to `force-output` for some kinds of network stream (it merely empties the stream's buffers, without assuring secure arrival at its destination).

`clear-output` is a no-op.

Section 22.3.3 . Formatted Output to Character Streams

Cautions

The method to be used to distribute justification pad characters in the ~< format directive is not defined. XCL uses a random distribution function. Note that this makes text look good, but any tables that happen to be justified will not line up.

CHAPTER 23 FILE SYSTEM INTERFACE

Ambiguities

`load` and `compile-file` require that Common Lisp plain text files, which must begin with a semi-colon to distinguish themselves from Interlisp source files. Plain text files are to be read in the package `user` (but see `:package` keyword below).

Namestring is defined to be of the form:

```
{HOST}DEVICE:<DIR>SUBDIR>SUBDIR>NAME.EXT;
VERSION
```

Additional Features/Improvements

The default value of `*load-verbose*` is `t`, meaning the file name and possibly other information is printed when the file is loaded.

`load` accepts an additional keyword, `:package`, whose value must be a package. `load` binds `*package*` to this value while reading the file. In the case of files produced by the Xerox Lisp File Manager, the value overrides the package specified in the file's makefile environment.

`:wild` is defined to be the same as a `*` in a namestring.

Limitations

The following optional values have been implemented for version numbers: `:oldest`. No other keywords are allowed out of the list shown in Steele's *Common Lisp: the Language*.

`load` makes no attempt to fill in default extensions on files.

XCL only supports 8-bit files. `open` accepts as values for the `:element-type` keyword only the following values `:string-char`, `character`, `:default`, and `unsigned-byte`. The first three are all equivalent and produce files of type "text", while `unsigned-byte` produces a file of type "binary".

[This page intentionally left blank]