IMPLEMENTATION NOTES

XEROX

3102464
Lyric Release
June 1987

XEROX COMMON LISP IMPLEMENTATION NOTES

3102464

Lyric Release

June 1987

This manual is set in Modern typeface with text written and formatted on Xerox Artificial Intelligence workstations. Xerox laser printers were used to produce text masters.

PREFACE

The Xerox Common Lisp Implementation Notes cover several aspects of the Lyric release. In these notes you will find:

- An explanation of how Xerox Common Lisp extends the Common Lisp standard. For example, in Xerox Common Lisp the Common Lisp array-constructing function `make-array` has additional keyword arguments that enhance its functionality.
- An explanation of how several ambiguities in Steele's *Common Lisp: the Language* were resolved.
- A description of additional features that provide far more than extensions to Common Lisp.

How the Implementation Notes are Organized

These notes are intended to accompany the Guy L. Steele book, *Common Lisp: the Language* which represents the current standard for Common Lisp.

The implementation notes are organized to coincide with the chapter and section arrangement of the Steele book. Not every section in the book has a corresponding section in the implementation notes because most of *Common Lisp: the Language* is implemented as presented.

How to Use the Implementation Notes

We recommend that when you consult Guy Steele's book, you also use these implementation notes to see if any extensions have been added or ambiguities resolved for that portion of the text.

TABLE OF CONTENTS

Only those chapters with additions, ambiguities, or extensions to Steele's *Common Lisp: the Language* are provided in this document.

[This page intentionally left blank]

CHAPTER 4 TYPE SPECIFIERS

4.2. Type Specifier Lists

The special form "the" operates as an assertion in interpreted code, but has no effect on compiled code.

Although it is not an error to ask (`typep x 'foo`) when `foo` is not yet a defined type at compile time, the compiler will produce much more efficient code if type `foo` is known at compile time.

4.8. Type Conversion Function

The function `coerce` operates only on the types explicitly listed in the book.

[This page intentionally left blank]

CHAPTER 5 PROGRAM STRUCTURE

5.1.2. Variables

Unbound special variables have `il:nobind` in their value cell. If you try to access an unbound variable in interpreted code, an error is signaled. If you try to use one in compiled code, `il:nobind` is returned as its value.

5.2. Functions

5.2.2. Lambda-Expressions

In this release, argument number checking is performed in the interpreter, but not in all compiled code. For those compiled functions that do not check, if the function is called with fewer arguments than the function requires, the remaining required arguments will have value `nil`; if called with more arguments than the function permits (required plus optionals), the extra arguments are ignored.

Compiled functions also do not check for unexpected keywords, or malformed keyword/value pairs, though the interpreter does.

`lambda-parameters-limit` ⇒ 512.

5.3.1. Defining Named Functions

`xcl:definline` *name arg-list* &body *body* [*Macro*]

`xcl:definline` is exactly like `defun` except that it also arranges for the compiler to expand inline any calls to the named function. In future releases, this will be accomplished via the inline declaration mechanism. In Lyric, however, the `xcl:defoptimizer` facility is used. As a result, users should take care not to use `xcl:definline` for recursive functions, as this will cause the compiler to loop indefinitely, expanding the recursive calls.

5.3.2. Declaring Global Variables and Named Constants

`xcl:defglobalvar` *name* `&optional` *initial-value doc-string* [*Macro*]

`xcl:defglobalvar` is exactly like `defvar` except that it declares the variable name to be *global* instead of special. Note that if you change a variable from a *global* to a *special*, all functions using that variable must be recompiled. See section 9.1 for more information on global declarations.

`xcl:defglobalparameter` *name initial-value* `&optional` *doc-string* [*Macro*]

`xcl:defglobalparameter` is analogous to `defparameter` except that it declares the parameter as global instead of special.

CHAPTER 6 PREDICATES

Predicates are required to return `nil` for false and non-`nil` for true. There are some types such that `typep` for the type and the specific predicate for the type are equivalent in truth value only (some predicates return `t`, some return the object itself).

The function `subtypep` is defined to return two values. The first value is the value of the predicate, and the second is the certainty of the result. `subtypep` could always return (`values nil nil`) and be legal; however, that wouldn't be a very useful implementation of `subtypep`. Xerox Common Lisp's `subtypep` is guaranteed to handle the following cases:

1. Any two datatypes (including structures defined with `defstruct` with no `:type` option) will return a definite answer.
2. "Built-in" Common Lisp types will return a definite answer.
3. `nil` is `subtypep` of everything.
4. Everything is `subtypep` of `t`.
5. No non-nil type is `subtypep` of `nil`.
6. `and` and `or` of any of the previous expressions are handled properly.

Though `equalp` is required to work for objects "with components", it is not specified if this includes structures. However, there is no other interesting way to test equality of structures. So our implementation defines `equalp` to mean all components are `equalp`. (This is analagous to Interlisp-D's `il:equalall`). Of course, this means `equalp` may not terminate if comparing circular structures, just as `equal` may not terminate given circular lists.

[This page intentionally left blank]

CHAPTER 7 CONTROL STRUCTURE

7.3. Function Invocation

`Call-arguments-limit` ⇒ 512.

7.5. Establishing New Variable Bindings

`xcl:destructuring-bind` *pattern form* `&body` *body* [*Macro*]

Executes *body* with the variables in the s-expression *pattern* bound to elements of the list structure returned by *form*. It is analogous to `multiple-value-bind`. *pattern* can be any arglist acceptable to `defmacro` except that the `&environment` keyword may not be used. For example:

```
(xcl:destructuring-bind
      ((vl  v2) &key a b)
       '((1 2) :b 3 :a 4)
      ...body...)
```

is equivalent to:

```
(let ((vl 1)
      (v2 2)
      (a  4)
      (b  3))
   ...body...)
```

Note: `xcl:destructuring-bind` currently does no error checking for too many or too few elements being returned by *form*.

7.6. Conditionals

`case` *keyform* {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]

Be careful about using `nil` as a keylist in the case macro. `nil` is interpreted as the list of no keys, not as the single key `nil`. Thus, any clause whose `car` is `nil` will never be selected. To use `nil` as a key, use the keylist `(nil)` instead.

Wrong:

```
(case expression
 ...
(nil ... code for expression being nil...)
  ...
  )
```

Right:

```
(case expression
  ...
  ((nil) ... code for expression being nil...)
  ...
  )
```

7.9.1. Constructs for Handling Multiple Values

```
multiple-values-limit ⇒ 512.
```

CHAPTER 8 MACROS

While the Common Lisp construct `defmacro` does remove any function definition the given symbol may have, it does not remove any Interlisp macro definition that might exist on the `il:macro`, `il:bytemacro`, or `il:dmacro` properties of the symbol. If a given symbol has both a Common Lisp and Interlisp macro definition, the one to be used depends upon the compiler or interpreter in use. The Common Lisp interpreter and the new XCL compiler will both use the Common Lisp macro. The Interlisp interpreter and compiler will use the Interlisp macro. Because of this potential for confusion, it is strongly recommended that, when providing a Common Lisp `defmacro` definition for a symbol, any existing Interlisp macro definition for that symbol should be removed.

Xerox Common Lisp diverges from *Common Lisp: the Language* on the issue of destructuring in `&body` parameters. We implement an extension to that syntax to allow for easy parsing of the bodies of certain macros, such as `defun` and `defmacro`. Some uses of `&body` are interpreted as implicit calls to the XCL function `parse-body`:

`parse-body` body *environment* `&optional` *doc-string-allowed-p* [*Macro*]

The given *body* should be a list of forms in the syntax of a standard Common Lisp `lambda` body, described in *Common Lisp: the Language* as

{*declaration* | *doc-string*}* {*form*}*

The *environment* should be a lexical environment such as those acquired through the use of the `&environment` keyword. *doc-string-allowed-p* defaults to `t`. `parse-body` returns three values:

1. A list of the non-declaration, non-doc-string *form*'s found in the *body*
2. A list of the `declare` forms found in the *body*
3. The documentation string found, if any, or `nil`.

`parse-body` works by examining the forms in *body* one by one, macroexpanding them if necessary, trying to find the first non-declaration, non-string form. The tail of *body* beginning with that form is returned as the first value, a list of all of the declarations found is

the second value and, if any documentation string is found, it is returned as the third value. If *doc-string-allowed-p* is `nil`, any strings found will be assumed to be the first form in the tail and the search will end. Note that Common Lisp allows macros to expand into either documentation strings or declarations. Because of this, `parse-body` will always have macroexpanded the first form in the tail. The original, unexpanded form is returned as the `car` of the first value, though.

Because of the usefulness of `parse-body`, and the frequency with which constructs like the following are used:

```
(defmacro define-foo (arg-list &body body
                              &environment env)
          (multiple-value-bind (code decls doc)
                               (parse-body body env)
              ...))
```

the syntax of the `&body` keyword was changed to allow the following code, with the same meaning:

```
(defmacro define-foo (arg-list &body (code decls doc))
      ...)
```

This frees the programmer from having to specify an `&environment` parameter when it will only be used in a call to `parse-body`.

The full syntax of the XCL `&body` keyword is as follows:

`&body` *symbol*

This is treated exactly like `&rest` *symbol*.

`&body` (*symbol-or-list*)

When `&body` precedes a list of length one, it is treated exactly like `&rest` *symbol-or-list*.

`&body` (*symbol-or-list symbol-or-list* [*symbol-or-list*])

When followed by a list of length two or three, it is treated as an implicit call to the function `parse-body`, described above. The *body* argument is the list that would have been bound to a simple `&rest` parameter, the *environment* argument is given by what would have been supplied to an `&environment` parameter, and *doc-string-allowed-p* is passed as `t` if, and only if, the third element of the `&body` list is provided. Each of the *symbol-or-list*'s is matched against the corresponding returned value of

`parse-body`. This allows full destructuring on each of those three values, even though it is only likely to be useful at all for the first one.

When this third, so-called "parsing version" of `&body` is used, no `&key` parameters are allowed. Also, as described in *Common Lisp: the Language*, only one of `&body` and `&rest` may be used in a single argument list. Note also that this extension to Common Lisp contradicts the statement on page 145 of *Common Lisp: the Language* that `&body` "is identical in function to `&rest`."

Compatibility Note

All Interlisp `nlambda` functions appear to be macros from the point of view of the Common Lisp function `macro-function`. Those Interlisp `nlambda` functions that actually evaluate some of their arguments have also been defined as real Common Lisp macros. Thus, all calls to Interlisp `nlambda` functions are treated properly by the Common Lisp interpreter and the new XCL compiler.

[This page intentionally left blank]

CHAPTER 9 DECLARATIONS

9.1. Declaration Syntax

Inline declarations are ignored in this release. The macro `xcl:definline` creates a function, calls to which will be expanded inline.

Xerox Common Lisp supports an additional declaration `xcl:global`.

(`xcl:global` *var1 var2* ...) specifies that all of the variables named are to be considered *global*, i.e., it is a declaration that the variables are never dynamically bound. All references to such a variable are compiled to fetch the top level binding directly. This declaration for global variables is analogous to the special declaration for special variables.

9.2. Declaration Specifiers

Xerox Common Lisp supports an additional declaration, `xcl:global`.

(`xcl:global` *var1 var2* ...) specifies that all of the variables named are to be considered *global*. This specifier pervasively affects variable references. The affected references refer directly to the top-level value of the variable, bypassing a search for any intermediate bindings. This can in some cases lead to a significant performance improvement, especially if the references are in deeply-nested, frequently-executed code. Variables declared `xcl:global` may not be bound. The declaration `xcl:global` is analogous to the `special` declaration for special variables.

`inline` declarations are ignored in the Lyric release. Inline functions can be defined using the `xcl:definline` macro, described in Section 5.3.1 of Steele's *Common Lisp: the Language*.

[This page intentionally left blank]

CHAPTER 10 SYMBOLS

Symbol print names are limited to 255 characters.

`symbol-name` returns a string displaced to the symbol pname. Strings returned from `symbol-name` may be destructively modified without affecting the symbol pname.

Interlisp users should note that `cl:gensym` and `il:gensym` are not the same. `il:gensym` always creates a symbol in the Interlisp package, while `cl:gensym` creates uninterned symbols.

[This page intentionally left blank]

CHAPTER 11 PACKAGES

Standard Packages

The following standard packages are included in Xerox Lisp:

LISP contains all symbols (other than keywords) defined in *Common Lisp: the Language*. Some of these symbols are shared with the Interlisp package (by importing them into INTERLISP), in the cases where the semantics of the symbols are identical (e.g., CAR).

KEYWORD contains all Common Lisp keywords.

SYSTEM contains system internals.

USER is the default package in a standard Common Lisp Exec. It uses the LISP package.

INTERLISP contains (or imports) all Interlisp symbols. All symbols in this package are external, reflecting Interlisp-D's flat symbol name space.

XCL contains symbols of the Xerox Common Lisp extensions. Many symbols in this package are also shared with Interlisp.

XCL-USER is the default package in a Xerox Common Lisp Exec. It uses both LISP and XCL; thus, extensions to Common Lisp are accessible in this package. Most users will prefer this package to USER.

Extensions to Standard Packages

`*package*` [*Variable*]

This symbol is bound in each exec.

`xcl:*total-packages-limit*` [*Constant*]

An inclusive limit to the total number of packages in the system. Currently this is 255 but will increase in the next release.

`do-symbols` [*Macro*]

`do-external-symbols` [*Macro*]

`do-all-symbols` [*Macro*]

These macros are as specified in *Common Lisp: the Language*, except note that symbols may be iterated over more than once.

`xcl:do-internal-symbols` [*Macro*]

Maps over only the internal symbols of a package, not those that are external (exported).

`xcl:do-local-symbols` [*Macro*]

Maps over the symbols interned in a package, internal and external (exported) symbols, not bothering to map those that are merely accessible in it (as by inheritance).

`xcl:delete-package` *package* [*Function*]

Uninterns all of the symbols interned in *package* and then removes the package structure itself. All of *package*'s symbols become uninterned and will then print out preceded by "hash colon," e.g., `#:foo`. This should obviously be used with caution.

`make-package` *name* `&key :prefix-name :internal-symbols :external-symbols :external-only` [*Function*]

There are several additional keywords for make-package:

`:prefix-name` *name*

The symbol printer uses *name* to prefix symbols that need to be qualified, instead of the package's full name.

`:internal-symbols` *positive-integer*

The number of internal symbols this package should expect to accommodate.

`:external-symbols` *positive-integer*

The number of external symbols this package should expect to accommodate.

`:external-only` *truth-value*

If this keyword is present the package will have only external symbols; i.e., interning a symbol in this package implicitly exports it.

`rename-package` *package new-name* `&optional` *new-nicknames prefix-name* [*Function*]

The function `rename-package` has been extended with a second optional argument *prefix-name*, with is the name the symbol printer will use to qualify symbols of this package when needed.

`defpackage` *name* `&rest` *option-clauses* [*NLambda Function*]

Define a package named *name*. If no such package already exists, create it using *option-clauses*. If one does exist, try to make it match the description, or produce an error if that's not possible. Arguments are unevaluated (it is an Interlisp NLambda). Each of *option-clauses* is a list whose car is a keyword from those described below.

This function can be used in a file's `il:makefile-environment` property to define the package in which the file is to be read and written. It is somewhat similar to the Symbolics `defpackage`.

The following option clauses are implemented:

(`:use` *name1 name2 ...*)

Causes the package to use the named package(s).

(`:nicknames` *name1 name2 ...*)

Adds the nickname(s) to the package.

(`:prefix-name` *name*)

The symbol printer will use *name* to prefix symbols that need to be qualified, rather than the package's full name.

(`:internal-symbols` *positive-integer*)

The number of internal symbols this package should expect to accommodate (only noticed if the package needs to be created).

(`:external-symbols` *positive-integer*)

The number of external-symbols this package should expect to accommodate (only noticed if the package needs to be created).

(`:external-only` *truth-value*)

If this keyword is present, the package will have only external symbols.

(`:shadow` *symbol1 symbol2* ...)

Shadow the given symbol(s) in this package.

(`:export` *symbol1 symbol2* ...)

Export (make external) the given symbol (s) from this package. Note: This option can only be used in a `defpackage` for an already-defined package, because otherwise the arguments to this option clause (internal symbols in the package) can't exist yet. (Of course, you can still export inherited symbols this way, but this is not a very interesting case.)

(`:import` *symbol1 symbol2* ...)

Import (make internal and accessible) the given symbol(s) in this package.

(`:shadowing-import` *symbol1 symbol2* ...)

Import (make internal and accessible) the given symbol(s) in this package, shadowing any conflicts.

Modules

`require` *module-name* `&optional` *pathname* [*Function*]

The implementation-specific way in which Xerox Lisp searches for a *module-name* when no pathname is provided is to first merge *module-name* with `*default-pathname-defaults*` and then with each of the contents of the variable `il:directories`.

Error Conditions Raised by the Package System

There are a number of situations in which the package system will raise error conditions, which can be caught and handled by the user from within the debugger. These situations are described below. For details on how to handle these conditions and invoke these

proceed cases, see the error system documentation in Chapter 24 of this manual.

While in the reader:

The conditions listed in this section are all subtypes of the `xcl:read-error` condition.

`xcl:symbol-colon-error` *name* [*Condition*]

Indicates that the reader has found a name with too many colons in it. *name* is a string containing all of the characters of the invalid symbol.

`xcl:escape-colons-proceed` [*Proceed case*]

Returns a symbol made in the current package, with the colons quoted.

`xcl:missing-external-symbol` *name package* [*Condition*]

This indicates that a name, qualified as external in a package, has not been found in a package. *name* is a string. *package* is a package.

`xcl:make-external-proceed` [*Proceed case*]

Creates and returns an external symbol.

`xcl:make-internal-proceed` [*Proceed case*]

Creates and returns an internal symbol.

`xcl:missing-package` *package-name symbol-name* [*Condition*]

This indicates that a package named *package-name*, referred to in a qualified symbol name, has not been found. Both *package-name* and *symbol-name* are strings.

`xcl:new-package-proceed` [*Proceed case*]

Creates a new package named *package-name* and interns the symbol there. The package is created with default attributes. If the symbol was qualified external it is exported from the new package.

`xcl:ugly-symbol-proceed` [*Proceed case*]

Creates a new internal symbol, in the current package, with a name composed of the package name, an appropriate number of colons, and the symbol name.

That is, it creates the symbol that would have resulted had the colon(s) in the name been escaped. This is handy when an old Interlisp symbol like DECLARE: has been typed, which should (in a Common Lisp readtable) be typed DECLARE\:.

`xcl:read-conflict` *name packages* [*Condition*]

Indicates that the reader compatibility feature has found a name whose package it cannot determine. This is described in detail below under "Koto Reader Compatibility Feature".

Package System Supertype Conditions:

`xcl:package-error` *package* [*Condition*]

This indicates an error has occurred in a call to package code that attempts to alter *package*. It is a subtype of the `error` condition. All the conditions described in this chapter, except for the reader errors listed above, are a subtype of this one. This error will almost never be signaled directly; most package conditions are actually of the type or types described below. The slot *package* is inherited by all subtype conditions of `xcl:package-error`.

`xcl:symbol-conflict` *symbols* [*Condition*]

This condition is a subtype of the `xcl:package-error` condition. It indicates that, during a package system operation, a set of symbol names has been found to conflict (the list of symbols in *symbols*). This error will almost never be signaled directly; most package system conditions are subtypes of this type, since it is the most common error. The slot *symbols* is inherited by all subtype conditions of `xcl:symbol-conflict`.

While calling use-package:

`xcl:use-conflict` *used-package* [*Condition*]

This is a subtype of the `xcl:symbol-conflict` condition. It indicates that during a `use-package` operation the conflicting *symbols* exported by the *used-package* have names that conflict with symbols already accessible in the *package*. *symbols* (inherited from `xcl:symbol-conflict`) is a list of the symbols.

`xcl:shadow-use-conflicts-proceed` [*Proceed case*]

Shadow conflicting symbols in the "using" package (*package*). This is the the safest way to proceed from this condition, but remember that references to any of the shadowed names will now refer to a local symbol, not the one that you might have been expecting to inherit.

`xcl:unintern-user-proceed` [*Proceed case*]

Unintern conflicting symbols from the "using" package (*package*). This is useful if you have inadvertantly interned the conflicting symbols by typing them to an executive before calling `use-package`. However, unless you are very sure of the use of the symbols being uninterned this operation may make those symbols permanently unavailable. This is a dangerous option; use it with caution.

`xcl:unintern-usee-proceed` [*Proceed case*]

Unintern conflicting symbols from the package being used (*used-package*). Unless you are very sure of the use of the symbols being uninterned this operation may make those symbols permanently unavailable. This is a dangerous option; use it with caution.

`xcl:abort` [*Proceed case*]

Abort the `use-package` operation.

While calling export:

`xcl:export-conflict` *exported-symbols packages* [*Condition*]

A subtype of the `xcl:symbol-conflict` condition. This condition indicates that exporting *exported-symbols* from *package* results in name conflicts with *symbols* in *packages*.

`xcl:unintern-proceed` [*Proceed case*]

Unintern conflicting symbols in *package*. Unless you are very sure of the use of the symbols being uninterned this operation may make those symbols permanently unavailable. This is a dangerous option; use it with caution.

`xcl:abort` [*Proceed case*]

Abort exporting from *package*.

`xcl:export-missing` *symbols* [*Condition*]

A subtype of the `xcl:package-error` condition. This condition indicates that the *symbols* are not available in *package* to be exported.

`xcl:import-proceed` [*Proceed Case*]

Import these symbols into *package* before exporting them.

`xcl:abort` [*Proceed Case*]

Abort export from *package*.

While calling import:

`xcl:import-conflict` [*Condition*]

This is a subtype of the `xcl:symbol-conflict` condition. It indicates that importing the *symbols* into *package* causes a name conflict with symbols already accessible in *package*.

`xcl:shadowing-import-proceed` [*Proceed case*]

Import *symbols* with `shadowing-import`.

`xcl:abort` [*Proceed case*]

Abort import into *package*.

While calling unintern:

`xcl:unintern-conflict` *symbol* [*Condition*]

This is a subtype of the `xcl:symbol-conflict` condition. It indicates that uninterning *symbol* from *package* causes name conflicts among the symbols on *symbols*.

`xcl:shadowing-import-proceed` [*Proceed case*]

Shadowing-import a new symbol into *package* to hide *symbols*.

`xcl:abort` *[Proceed case]*

Abort unintern of *symbol* from *package*.

Koto Reader Compatibility Feature

For the benefit of Koto users of the CML Library module, the Lyric release contains a "reader compatibility feature" to aid in reading Koto CML files into Lyric. If you do not have any such files, you can ignore this section.

The Koto release did not have an implementation of packages, so the CML module used a syntactic convention in which symbols containing colons were used to denote keywords and those Common Lisp symbols whose names conflicted with Interlisp symbols. Of course, the vast majority of Common Lisp names do not conflict, and those symbols were written with no package prefix. Ordinarily, if you were to load such a file into Lyric, all the symbols would be read as Interlisp symbols, and the colons would be treated as any other alphabetic character, consistent with the syntactic conventions of Interlisp in releases prior to Lyric. For example, the character sequence "CL:UNLESS" would read as the symbol `il:cl\:unless`; the sequence "FIND-PACKAGE" would read as the symbol `il:find-package`.

Enabling the reader compatibility feature causes the reader to attempt to resolve all symbols into the appropriate package. The feature is enabled when

(a) `il:litatom-package-conversion-enabled` (a special variable) is true, and

(b) the read table being used (the value of `*readtable*`) is either `il:filerdtbl` or `il:coderdtbl`.

Condition (b) is met when loading files produced by the File Manager and the compiler prior to Lyric and is (usually) not true for files produced in Lyric. You should only enable the compatibility feature when loading Koto CML files, as it may cause other files to be read incorrectly.

The reader feature handles two cases: strings containing an explicit "package prefix", and unqualified strings that name a symbol in the LISP package. When enabled, the reader follows the following procedure when it encounters a string of characters to be interpreted as a symbol:

1. If the string contains an explicit package prefix, such as a leading colon, or "CL:", the string is interned in the package indicated by the prefix.

2. If the string does not name a symbol in the LISP package, then no conversion is needed—the string is interned in the INTERLISP package.

3. If the string names a symbol in the LISP package and there is not already a symbol by the same name in the INTERLISP package, the reader returns the LISP symbol.

4. At this point, the string names symbols in both LISP and INTERLISP. If the LISP symbol is not an external one, then the conflict is with a private LISP symbol and hence accidental; the reader returns the INTERLISP symbol.

5. If exactly one of the symbols is on the preferred reading list (see below), the reader returns that symbol.

6. Otherwise, there is a conflict that cannot be automatically resolved. This will in general happen for any symbol of Common Lisp for which there happens to already exist an Interlisp symbol that was not "shadowed" in CML.

In case 6, a debugger window appears with the message

Symbols named *name* exist in packages Lisp and Interlisp.

Several proceed cases are available under the "PROCEED" option in the debugger menu. These are:

Return Lisp symbol, make it preferred

This returns the symbol from the Lisp package and also puts it on the global list `xcl:*preferred-reading-symbols*`, removing the Interlisp symbol if it was there. This is useful in

cases where you have accidentally interned an uninteresting symbol in Interlisp by typing a name without a CL: qualifier. This usually results in an error, such as undefined function, but in the meantime you have created the symbol in the Interlisp package, making it difficult for the compatibility feature to decide what to do. From the moment this symbol is made preferred, you will no longer receive warnings and it will *always* be read as a Lisp symbol.

Just return Lisp symbol

This is a conservative version of the above choice.

Return Interlisp symbol, make it preferred

In some cases you may want to prefer the reading of an Interlisp symbol to that of a similarly named Common Lisp one. The symbol is made preferred by placing it on the global list `xcl:*preferred-reading-symbols*`, removing the Lisp symbol if it was there. The next time it is encountered, the reader feature will use it instead of the Lisp symbol.

Just return Interlisp symbol

Again, this is a conservative version of the above choice, one which does not make the Interlisp symbol preferred.

Reader Compatibility Feature: Making Symbols Preferred

`xcl:*preferred-reading-symbols*` [*Global variable*]

This global list contains symbols (not namestrings) whose reading is preferred. If both Interlisp and Lisp symbols appear on the list the name will still be considered ambiguous. Be careful about placing symbols on this list. You should be very sure that they will never be referred to on a file in such a way that the other meaning is desired. When a symbol from one package is marked preferred (via a proceed option in the debugger), the other one is removed, if it was present. This list initially contains a set of Interlisp symbols corresponding to Lisp symbols "shadowed" in CML (by symbols beginning with "CL:") or not implemented in CML.

Reader Compatibility Feature: Enabling and Disabling

`il:litatom-package-conversion-enabled` [*Variable*]

Set or bind this flag true to enable the compatibility feature. Note that reader performance drops considerably when the compatibility feature is enabled. This flag should only be turned on while reading files written using the Koto Common Lisp library module.

Reader Compatibility Feature: Format of the Conversion Table

`il:litatom-package-conversion-table` [*Global variable*]

This table is a list of clauses specifying the "package prefixes" to check when reading a symbol while the compatibility feature is enabled. The clauses are searched linearly. Each clause has the form:

(prefix-string exception-list package-name where-keyword)

The initial contents of this table are suitable for converting files produced using the Koto CML library module. Such clauses are:

```
(":" NIL "KEYWORD" :external)

("CL:" ("CL:FLG") "LISP" :external)
```

You need only alter the table if you are trying to convert files that contained additional user "pseudo-packages."

prefix-string is a string which is matched to the first characters of a name. If the name matches the **prefix-string** this clause is "activated."

exception-list is a list of strings. If the name, including its prefix, matches any of these strings it is not converted and the conversion is aborted.

package-name is a string containing the name of the package in which the symbol name (without its prefix) will be interned.

where-keyword is one of the keywords `:internal` or `:external`, indicating whether the symbol is to be interned or interned and immediately exported.

Note that since the clauses are tested sequentially, longer prefixes must go earlier in the list. If, for

example, you wanted to convert "CL::" prefixed names to be internal in "LISP" then you would have to place a clause before the one starting "CL:". This avoids the "CL:" clause being activated for symbols named "CL::FOO" and the like.

Reader Compatibility Feature: Conditions

The reader compatibility feature uses the following condition and proceed cases in its interaction:

`xcl:read-conflict` *name packages* [*Condition*]

This condition indicates that the reader compatibility feature (see below) has found a name whose package it cannot determine. It is a subtype of the `xcl:read-error` condition. *name* is a string. *packages* contains a list of the packages in which the name was found, and between whom the reader feature cannot decide.

`xcl:prefer-clsym-proceed` [*Proceed case*]

Return LISP symbol, make it preferred.

`xcl:return-clsym-proceed` [*Proceed case*]

Just return LISP symbol.

`xcl:prefer-ilsym-proceed` [*Proceed case*]

Return INTERLISP symbol, make it preferred.

`xcl:return-ilsym-proceed` [*Proceed case*]

Just return INTERLISP symbol.

Moving Existing Code into a New Package

Now that Xerox Lisp supports Common Lisp packages, users may wish to take advantage of package modularity by moving existing code modules into their own packages. For Common Lisp code being maintained in purely text form, *Common Lisp: the Language* tells you much of what you need to know. However, there are several additional considerations for code maintained by the Xerox Lisp File Manager; this section addresses some of these considerations.

The Lyric release contains no tools for completely automating the conversion to another package, nor does it supply tools for supporting very complex packages. The discussion that follows points out some of the mechanisms that may help for creating relatively simply user packages. It assumes you have a file or set of files produced by the File Manager in Lyric. Files written with the Koto CML module should first be converted to Lyric as described above.

How to specify the makefile-environment

In order to have a file written in your own package, it must have a makefile-environment property, which takes the form of a list (`:readtable` *tbl* `:package` *package*). The *Xerox Lisp Release Notes* on the File Manager discuss how this property is used. The discussion here is confined to the form in which the *package* is described.

If you want to write a file in one of the standard packages, such as XCL-USER or INTERLISP, you need only specify the package name, preferably as a string (make sure it is upper-case). If you want to use your own package, you *must* supply an expression whose evaluation will return the package, creating it if necessary. The expression must not assume that any package, other than the standard ones, already exists; in particular, the expression cannot contain any symbols that are in your new package. It should also be self-contained; e.g., if it calls `in-package`, it must be sure to bind `*package*`, in order not to side-effect whatever code is loading or otherwise using the expression to produce a reader environment. And finally, it should not assume that the file it is on is actually being loaded; makefile environments are examined and evaluated by various system utilities that manipulate files (e.g., `il:loadfns`), not just the loader.

For most packages, the simplest expression to use is `defpackage`. It creates the package if it does not yet exist, and returns the package's name in any case, so it is well suited as a *package* expression. For example, to specify a package that inherits both LISP and XCL (as the pre-supplied XCL-USER package does) and imports the Interlisp window system symbols `createw` and `windowprop`, you could write

```
(defpackage "MYHACK"
     (:use "LISP" "XCL")
     (:nicknames "MH")
     (:import il:createw il:windowprop))
```

The major complication arises if you want to export any symbols (any package that presents a programmer's interface surely does). You can't put the exported symbols in the `defpackage` expression, because the package doesn't yet exist in which to type them. There are two principal ways to do the exporting: export the symbols later (in the body of the file), or write a more complex expression.

In the former case, you write a minimal `defpackage` expression for the makefile-environment, then write a more complete one in the body of the file (e.g., in a P command, or in an initialization function). The minimal `defpackage` is responsible for creating enough of the package so that expressions on the file can be read. This means it has to specify inheritance, imported symbols and any shadows. For example, you might write

```
(defpackage "MYHACK"
     (:use "LISP" "XCL")
     (:import il:createw il:windowprop))
```

as the minimal expression, then in the body of the file write the "full" expression, which can rely on the package already having been created:

```
(defpackage "MYHACK"
     (:use "LISP" "XCL")
     (:nicknames "MH")
     (:export make-hack-window save-hack))
```

This method requires some discipline on the part of package users, since the package as created by the simple expression lacks external symbols. In this state, forms on the file can still be read correctly (though when printed from any other package context the not yet exported symbols will appear with two colons in their name). However, the package cannot be properly inherited by any other package, since references from such a package to the not yet exported symbols will instead create internal symbols in the other package, which will (a) be the wrong symbols and (b) create a package conflict when the full package definition is evaluated. Thus, users of the package *must* ensure that the file containing the full

package definition is loaded before attempting to use-package it or refer to its symbols.

The alternative is to write a single very careful expression to define the whole package, e.g.,

```
(let (*package*)
   (in-package "MYHACK" "MH" '("LISP" "XCL"))
   (import '(il:createw il:windowprop))
   (export (mapcar #'intern
           '("MAKE-HACK-WINDOW" "SAVE-HACK"))))
```

If you have a very complex package, or one that is used on many files, it may be preferable just to create a file whose sole purpose is to define the package, then require that file. For example, the package expression might simply be

```
(progn (require "MYHACKDEFS") "MYHACK")
```

Changing the Package of Existing Code

Once you've decided how to define the package, you still have to arrange for the symbols currently in some old package to be moved into your new package. Much of this task can be done by specifying an explicit package to the loader, either as the `:package` keyword to `cl:load`, or the fourth argument to `il:load`. The package you specify overrides whatever package is specified in the file's makefile environment.

In order for this to work, the new package must have fundamentally the same inheritance structure as the package in which the file was written. For example, if the file was written in the XCL-USER package, your new package must inherit LISP and XCL. If the file was written in the INTERLISP package, your new package must inherit INTERLISP (but read the cautions below). When you load the file, the reader will then do the "right" thing whenever it encounters a symbol with no package qualifier—if the symbol was inherited by the old package, it will also be inherited by the new package, so the exact same symbol is read; if the symbol was local to the old package, it will not be inherited, but will be read as a local symbol in the new package.

After loading the file and doing whatever touchups seem appropriate (e.g., there may be local symbols in the old package that really should have been references to the old package), give the file a new

makefile-environment property as described above, then call `il:makefile` to write out the new file.

Changing Package Inheritance

It may be the case that you want your new package to have a different inheritance structure than the old. For example, you have a file written in the INTERLISP package that you want to move into a Common Lisp package. In this case, you should temporarily define your new package to have the same inheritance as the old package, and load the file as above. At this point, all the important symbols have been read correctly. Then change the new package's inheritance structure to be as desired, for example:

```
(cl:in-package "RAPT")
(cl:unuse-package "INTERLISP")
(cl:use-package '("LISP" "XCL"))
```

At this point, your new package is defined the way you want, but you may still have unexpected references to other packages, typically in the form of lexical variables in functions. For example, in the case of moving from INTERLISP to a non-INTERLISP package, many of your module's lexical variables happened to coincide with symbols already extant in INTERLISP, so were still read as INTERLISP symbols. If you view the definition of a function in your new module, you may see such things as

```
(let ((il:x (car il:top))
      il:a il:b)
      ...)
```

You'll likely want to rename those variables to be locals in your new package. It is fairly easy to write an SEdit mutator function that searches for symbols not accessible in the current package and replaces them (with user approval) with symbols of the same name interned in the current package.

Caution about referring to other packages

If you use symbols from other packages (other than the standard ones), you must, of course, make sure that the other package is defined. There are some subtleties here when the File Manager is involved.

If you want your package to inherit another non-standard package, you must ensure that the module defining the other package is loaded before

defining your package. For example, your package expression might look like:

```
(progn (require "MYHACKDEFS")
       (defpackage "MOREHAX"
           (:use "LISP" "MYHACK")))
```

Similarly, if you simply want to refer to symbols of another package, you must ensure that its module is loaded first. The same procedure is recommended, even if you are not defining your own package. For example, on an Interlisp file, you might give as package expression:

```
(progn (require "MYHACKDEFS")
       "INTERLISP")
```

You might be tempted to do this instead by including one or more `files` commands in the file's coms, e.g., the command

```
(il:files 'myhackdefs).
```

However, there are two problems with this method. You can't refer to any of the variables in the coms, for example,

```
(vars (myhack:default-size 37)
      (myhack:default-speed :fast))
```

because at the time the coms expression itself is read, the `files` command contained in it has not yet been executed, so you can get a missing package error when the reader encounters `myhack:default-size`.

Even if you solve that problem, for example by hiding all the variable settings in an initialization function, system utilities that attempt to read only part of the file (e.g., `il:loadfns`) may fail when they encounter the other symbols. The utility will have read the makefile environment (which is why the `require` works there), but will not necessarily have read, much less evaluated, the commands in the body of the file that cause the other file to be loaded.

Special considerations for the IL package

It is perfectly permissible to define a package that inherits from the INTERLISP package. However, when you do so, you must keep in mind a couple of things that are special about it: INTERLISP is external only, and the INTERLISP package's external symbols (i.e., the entire package) are not all defined in the standard

sysout—loading Library and LispUsers modules generally adds new symbols to the INTERLISP package. Following are some of the pitfalls to watch for.

The File Manager currently requires that certain symbols be in the INTERLISP package, regardless of the package of the file's contents: the symbol naming the file's coms (whose value is a list describing the file's contents), and the file's rootname (whose property list includes the file's makefile environment, file type and other File Manager properties). However, INTERLISP is external only, so those symbols are automatically inherited by any package using INTERLISP. Thus, for example, when the File Manager writes the symbol `il:foocoms` on the file `foo`, whose environment is the package `bar` inheriting `il`, the printer does not qualify the symbol with its package name, printing simply `"foocoms"`. If you subsequently load `foo` into a standard sysout, the reader encounters the token `"foocoms"` and, since `il:foocoms` does not exist, reads it as `bar::foocoms`.

To avoid this problem, you must ensure that the symbols `il:foocoms` and `il:foo` exist when the file is loaded. The simplest way is to include them in the makefile environment's package expression:

```
(progn '(il:foocoms il:foo)
       (defpackage "BAR"
            (:use "INTERLISP")))
```

For essentially the same reason, you must be careful that your file loads in advance any modules that define INTERLISP symbols it needs to refer to. Otherwise, since those symbols are external in INTERLISP, references to them are not qualified, and thus will be read instead as new internal symbols in your package. See the discussion above about referring to symbols from other packages.

When you follow the procedure outlined above for loading an Interlisp file into an Interlisp-inheriting package, be sure to load the file into a pristine sysout, or at least one in which your file has never been read. Otherwise, all the symbols on the file will already have been interned in INTERLISP, and thus would be read as the same INTERLISP symbols, rather than symbols of your new package.

[This page intentionally left blank]

CHAPTER 12 NUMBERS

As stated in *Common Lisp: the Language*, the values of named constants are implementation-dependent. The section that follows lists the limits for the Xerox Common Lisp implementation of the constants described by Steele.

12.10. Implementation Parameters

```
most-positive-fixnum ⇒ 65535
most-negative-fixnum ⇒ -65536
most-positive-short-float ⇒ 3.4028235E+38
least-positive-short-float ⇒ 1.40129847E-45
least-negative-short-float ⇒ -1.1754945E-38
most-negative-short-float ⇒ -3.4028235E+38
most-positive-single-float ⇒ 3.4028235E+38
least-positive-single-float ⇒ 1.40129847E-45
least-negative-single-float ⇒ -1.1754945E-38
most-negative-single-float ⇒ -3.4028235E+38
most-positive-double-float ⇒ 3.4028235E+38
least-positive-double-float ⇒ 1.40129847E-45
least-negative-double-float ⇒ -1.1754945E-38
most-negative-double-float ⇒ -3.4028235E+38
most-positive-long-float ⇒ 3.4028235E+38
least-positive-long-float ⇒ 1.40129847E-45
least-negative-long-float ⇒ -1.1754945E-38
most-negative-long-float ⇒ -3.4028235E+38
short-float-epsilon ⇒ 1.1920929E-7
single-float-epsilon ⇒ 1.1920929E-7
double-float-epsilon ⇒ 1.1920929E-7
long-float-epsilon ⇒ 1.1920929E-7
```

```
short-float-negative-epsilon ⇒ 5.9604645E-8
single-float-negative-epsilon ⇒ 5.9604645E-8
double-float-negative-epsilon ⇒ 5.9604645E-8
long-float-negative-epsilon ⇒ 5.9604645E-8
```

CHAPTER 13 CHARACTERS

13.1. Character Attributes

Characters in Xerox Common Lisp follow the Xerox NS Character Code Standard. Character codes are 16-bit quantities, partitioned into 8 bits of character set and 8 bits of character within the set. The value of the constant `char-code-limit` is 65536. Characters are an immediate data type; i.e., they consume no storage.

Xerox Common Lisp supports neither font nor bits attributes. Hence, the values of the constants `char-font-limit` and `char-bits-limit` are both one . The functionality of font and bits attributes are achieved instead through graphics programming conventions and the use of a larger character space.

Characters do not themselves have "font" attributes; however, streams have a notion of a current font, and some programs attach fonts to larger entities, such as strings or subranges of a file. In addition, the Xerox Character Standard encodes in the character itself some information that other implementations associate with a font. For example, a lower-case beta (β) is a distinct character (in the Greek character set), rather than being a lower-case b with a Greek font attribute. This distinct character can be rendered in an assortment of fonts.

The use for which "bits" attributes were originally intended was to represent different keyboard keystrokes (e.g., meta-hyper-A). The equivalent functionality is achieved in Xerox Common Lisp by assigning codes from other character sets to keystrokes using the key action table. Xerox Common Lisp follows the convention that characters typed with the Meta key depressed are in character set 1. Most of the extra function keys on the keyboard are in character set 2.

```
char-code-limit ⇒ 65536

char-font-limit ⇒ 1

char-bits-limit ⇒ 1
```

13.2. Predicates on Characters

XCL considers `graphic-char-p` to be true for exactly those characters in the space that the Xerox Character Code Standard calls "graphic" or "rendering". This space consists of characters whose character set component is zero or in one of the octal ranges [41, 176] or [241, 376], and whose character byte is in one of the octal ranges [40, 176] or [241, 376]. In particular, all of the normal ASCII printing characters are in the range [40, 176] in character set 0, and hence are graphic. Not all graphic characters are necessarily defined or have a rendering in any particular font.

The character sets 1 thru 40 and 177 thru 240 (octal) are in the range that the Xerox Character Code Standard calls "control characters". Of these character sets, only sets 1 and 2 have any assigned meaning in this release of Xerox Common Lisp.

13.4. Character Conversions

The names of most non-graphic characters are of the form *cset-char*, where *cset* is a character set name (e.g., Greek) or its octal representation (0 to 376), and *char* is the octal representation of the character within the set (0 to 376). The ASCII control characters (codes 1 through 32 octal) have names of the form "↑letter", e.g., `(cl:char-name (cl:code-char 2))` ⇒ "↑B". Some well-known characters, including those documented in *Common Lisp: the Language*, have more interesting names, which are registered in the association list `il:characternames`. Users are free to add to this list, but should beware of reading characters with new names in a system that has only the default `il:characternames`. Names of well-known character sets are registered on the list `il:charactersetnames`.

Graphic characters have no name (`char-name` returns `nil`) and print as themselves.

You can input characters using the same syntax, or you can give the character within the character set a name. For example, `#\1-A` and `#\1-101` are the same character (capital A in character set 1, or the character obtained by typing Meta-A). Lowercase beta can be

typed #\Greek-142 or #\46-142; being a graphic character, it always prints as #\β. However, you cannot use the "names" of lowercase letters, because Common Lisp reads case-insensitively. Thus, it could not distinguish #\Greek-B from #\Greek-b.

13.5. Character Control-Bit Functions

The constants char-control-bit, char-meta-bit, char-super-bit, and char-hyper-bit have the value zero (0), since non-zero bits attributes are not supported in XCL. In addition, the functions char-bit and set-char-bit exist but signal an error when called.

[This page intentionally left blank]

CHAPTER 15 LISTS

15.1. Conses

Xerox Common Lisp assumes trees are non-circular. Therefore, passing circular lists to these functions results in undefined actions (likely to be stack overflows or infinite loops).

15.2. Lists

`pushnew` uses the same keywords as `adjoin`, not the same ones as the standard sequence operands.

15.5. Using Lists as Sets

It is an error to hand these functions lists which are not true sets.

[This page intentionally left blank]

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

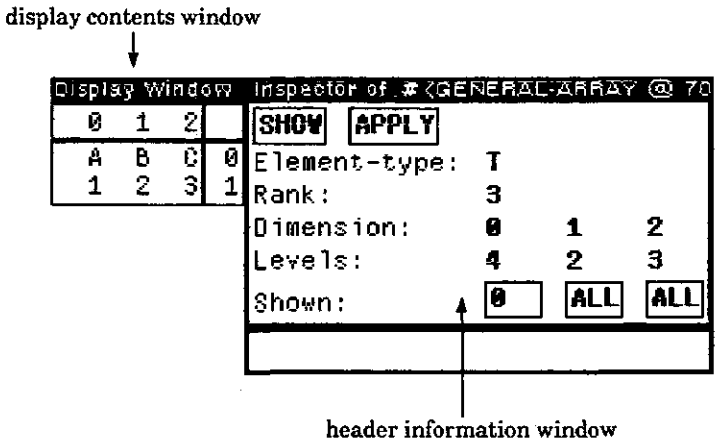
Inspecting Arrays

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:

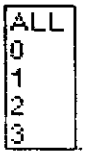
```
(setq a (make-array '(4 2 3) :initial-contents
                    '(((a b c) (1 2 3))
                      ((d e f) (3 1 2))
                      ((g h i) (2 3 1))
                      ((j k l) (0 0 0)))))
```

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.



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]

CHAPTER 24 ERROR SYSTEM

This chapter replaces most of Chapter 24, Errors, of *Common Lisp, the Language*.

The Xerox Common Lisp error system is based on proposal number 8 for the Common Lisp error system. Deviations from this proposal are noted. In particular, proceeding and proceed functions are more like those in an earlier proposal. Since the Common Lisp error system has not yet been standardized, this system may change in future releases to accommodate the final version of the Common Lisp error system.

Introduction to Error System Terminology

condition A *condition* is a kind of object which is created when an exceptional situation arises in order to represent the relevant features of that situation.

signal, handlers Once a condition is created, it is common to *signal* it. When a condition is signaled, a set of *handlers* are tried in some pre-defined order until one decides to *handle* the condition or until no more handlers are found. A condition is said to have been handled if a handler performs a non-local transfer of control to exit the signalling process.

proceed Although such transfers of control may be done directly using traditional Lisp mechanisms such as `catch` and `throw`, `block` and `return`, or `tagbody` and `go`, the condition system also provides a more structured way to *proceed* from a condition. Among other things, the use of these structured primitives for proceeding allow a better and more integrated relationship between the user program and the interactive debugger.

serious conditions It is not necessary that all conditions be handled. Some conditions are trivial enough that a failure to handle them may be disregarded. Others, which we

will call *serious conditions* must be handled in order to assure correct program behavior. If a serious condition is signalled but no handler is found, the debugger will be entered so that the user may interactively specify how to proceed.

errors Serious conditions which result from incorrect programs or data are called *errors*. Not all serious conditions are errors, however. Storage conditions are examples of serious conditions that are not errors. For example, the control stack may legitimately overflow without a program being in error. Even though a stack overflow is not necessarily a program error, it is serious enough to warrant entry to the debugger if the condition goes unhandled.

Some types of conditions are predefined by the system. All types of conditions are subtypes of `xcl:condition`. That is,

```
(typep c 'xcl:condition)
```

is true if c is a condition.

creating conditions The only standard way to define a new condition type is `xcl:define-condition`. The only standard way to instantiate a condition is `xcl:make-condition`.

When a condition object is created, the most common operation to be performed upon it is to *signal* it (although there may be applications in which this does not happen, or does not happen immediately).

When a condition is signaled, the system tries to locate the most appropriate handler for the condition and invoke that handler. Handlers are located according to the following rules:

bound

- Check for locally defined (ie, *bound*) handlers.
- If no appropriate bound handler is found, check first for the default handler of the signalled type and then of each of its superiors.

decline If an appropriate handler is found, the handler may *decline* by simply returning without performing a non-local transfer of control. In such cases, the search for an appropriate handler is picked up where it left off, as if the called handler had never been present. When a handler is running, the "handler binding stack" is popped back to just below the binding that caused that handler to be invoked. This is done to

avoid infinite recursion in the case that a handler also signals a condition.

`xcl:handler-bind` When a condition is signaled, handlers are searched for in the dynamic environment of the signaller. Handlers can be established within a dynamic context by use of `xcl:handler-bind`.

handler A *handler* is a function of one argument, the condition to be handled. The handler may inspect the object (using primitives described in another section) to be sure it is interested in handling the condition. After inspecting the condition, the handler must take one of the following actions:

- It may decline to handle the condition, by simply returning. When this happened, the returned values are ignored and the effect is the same as if the handler had been invisible to the mechanism seeking to find a handler. The next handler in line will be tried, or if no such handler exists, the default action for the given condition will be taken. A default handler may also decline, in which case the condition will go unhandled. What happens then depends on which function was used to signal the condition (`xcl:signal`, `error`, `cerror`, `warn`).
- It may perform some non-local transfer of control using `go`, `return`, `throw`, `abort`, or `xcl:invoke-proceed-case`.
- It may signal another condition.
- It may invoke the interactive debugger.

`xcl:proceed-case` When a condition is signaled, a facility is available for use by handlers to non-locally transfer control to an outer dynamic contour of the program. The form which creates contours that may be returned to is called `xcl:proceed-case`. Each contour is set up by an `xcl:proceed-case` clause, and is called a *proceed case*. The function that transfers control to a proceed case is called `xcl:invoke-proceed-case`.

proceed function Also, control may be transferred along with parameters to a named `xcl:proceed-case` clause by invoking a *proceed function* of that name.

Proceed functions are created with the macro `xcl:define-proceed-function`.

proceed type A proceed case with a particular name, or a particular set of proceed cases that share an interface defined by a *proceed function*, are sometimes called a *proceed type*.

report In some cases, it may be useful to *report* a condition or a proceed case to a user or a log file of some sort. When the printer is invoked on a condition or proceed case and `*print-escape*` is `nil`, the report function for that object is invoked. In particular, this means that an expression like

```
(princ condition)
```

will invoke `condition`'s report function. Because of this, no special function is provided for invoking the report function of a condition or a proceed case.

Program Interface to the Condition System

Defining and Creating Conditions

`xcl:define-condition` *name parent-type* [*Macro*]
*{keyword value}**
*{slots}**

Defines a new condition type with the given *name*, making it a subtype of the given *parent-type*.

Except as otherwise noted, the arguments are not evaluated.

The valid *keyword/value* pairs are:

`:conc-name` *symbol-or-string*

As in `defstruct`, this sets up automatic prefixing of the names of slot accessors. Also as in `defstruct` if no prefix is specified the default behavior for automatic prefixing is to use the name of the new type followed by a hyphen.

`:report-function` *expression*

expression should be a suitable argument to the `function` special form, e.g., a symbol or a lambda expression. It designates a function of two arguments, a condition and a stream, which prints the condition to the stream when `*print-escape*` is `nil`.

The `:report-function` describes the condition in a human-sensible form. This item is somewhat different than a structure's `:print-function` in that it is only used if `*print-escape*` is `nil`.

`:report` *form*

A short form of `:report-function` to cover two common cases.

If *form* is a constant string, this is the same as

```
:report-function
   (lambda (ignore stream)
      (write-string form stream))
```

Otherwise, this is the same as

```
:report-function
   (lambda (condition
            *standard-output*)
      form)
```

In the latter case, the form describes how to print objects of the type being defined. The form should do output to standard output. The condition being printed will be the value of the variable `condition` (the symbol `condition` in this usage is in the same package as the name of the new condition type). The condition's slots are accessible as simple variables within the report form.

`:handler-function` *expression*

expression should be a suitable argument to the `function` special form. It designates a function of one argument, a condition, which may handle that condition if no dynamically-bound handler did.

`:handle` *form*

An expression to be used as the body of a default handler for this condition type. While executing *form*, the variable `condition` will be bound to the condition being handled (as with `:report` above, the symbol `condition` in this usage is in the same package as the name of the new condition type). That is, this defines a function

```
(lambda (condition)
   form)
```

as the default handler for that type.

It is an error to specify both `:report-function` and `:report` in the same `xcl:define-condition` form. It is also an error to specify both `:handler-function` and `:handle`. If neither `:report-function` nor `:report` is specified, information about how to print this type of condition will be inherited from the `parent-type`. If neither `:handler-function` nor `:handle` was specified, there will be no default handler for the new condition type.

slots is a list of *slot-descriptions*, and specifies slots to be used by the given type. In addition to those specified, the slots of the *parent-type* are also available. A *slot-description* is exactly the same as for `defstruct` except that no slot-options are allowed, only an optional default-value expression. Condition objects are immutable, i.e., all of their slots are declared to be `:read-only`.

`xcl:make-condition` will accept keywords with the same name as any of the slots, and will initialize the corresponding slots in conditions it creates.

Accessors are created according to the same rules as used by `defstruct`. For example:

```
(xcl:define-condition bad-food-color food-lossage
      :report (format t "The food ~A was ~A"
                        food color)
   food
   color)
```

defines an error of type `bad-food-color` which inherits from the `food-lossage` condition type. The new type has slots `food` and `color` so that `xcl:make-condition` will accept `:food` and `:color` keywords and accessors `bad-food-color-food` and `bad-food-color-color` will apply to objects of this type.

The report function for a condition will be implicitly called any time a condition is printed with `*print-escape*` being `nil`. Hence,

```
(princ condition)
```

is a way to invoke the condition's report function.

Here are some examples of defining condition types. This form defines a condition called `machine-error` which inherits from `error`:

```
(xcl:define-condition machine-error error
      :report (format t
                      "There is a problem with ~A."
                      machine-name)
   machine-name)
```

The following defines a new error condition (a subtype of `machine-error`) for use when machines are not available:

```
(xcl:define-condition machine-not-available-error
                      machine-error
    :report (format t
                    "The machine ~A is not available."
                    machine-name)
   machine-name)
```

The following defines a still more specific condition, built upon `machine-not-available-error`, which provides a default for `machine-name` but which does not provide any new slots:

```
(xcl:define-condition
          my-favorite-machine-not-available-error
          machine-not-available-error
   (machine-name "Tesuji:AISDev:Xerox"))
```

This gives the `machine-name` slot a default initialization. Since no `:report` clause was given, the information supplied in the definition of `machine-not-available-error` will be used if a condition of this type is printed while `*print-escape*` is `nil`.

`xcl:condition-reporter` *type* [*Macro*]

Returns the object used to report conditions of the given *type*. This will be either a string, a function of two arguments (condition and stream) or `nil` if the report function is inherited. `setf` may be used with this form to change the report function for a condition type.

`xcl:condition-handler` *type* [*Macro*]

Returns the default handler for conditions of the given *type*. This will be a function of one argument or `nil` if the default handler for that type is inherited. `setf` may be used with this form to change the default handler for a condition type.

`xcl:make-condition` *type* `&rest` *slot-initializations* [*Function*]

Calls the appropriate constructor function for the given type, passing along the given slot initializations

to the constructor, and returning an instantiated condition.

The *slot-initializations* are given in alternating keyword/value pairs. eg,

```
(xcl:make-condition 'bad-food-color
   :food my-food
   :color my-color)
```

This function is provided mainly for writing subroutines that manufacture a condition to be signaled. Since all of the condition-signalling functions can take a *type* and *slot-initializations*, it is usually easier to call them directly.

Signalling Conditions

`xcl:*current-condition*` [*Variable*]

This variable is bound by condition-signalling forms (`xcl:signal`, `error`, `cerror`, and `warn`) to the condition being signaled. This is especially useful in proceed case filters. The top-level value of `xcl:*current-condition*` is `nil`.

`xcl:signal` *datum* `&rest` *arguments* [*Function*]

Invokes the signal facility on a condition. If the condition is not handled, `xcl:signal` returns the condition object that was signaled.

If *datum* is a condition then that condition is used directly. In this case, it is an error for `xcl:arguments` to be non-`nil`.

If *datum* is a condition type, then the condition used is the result of doing

```
(apply #'xcl:make-condition
       datum arguments)
```

If *datum* is a string, then the condition used is the result of doing

```
(xcl:make-condition
   'xcl:simple-condition
   :format-string datum
   :format-arguments arguments).
```

If the condition is of type `xcl:serious-condition`, then `xcl:signal` will behave exactly like `error`, i.e., it will call `xcl:debug` if the condition isn't handled, and will never return to its caller.

`error` *datum* `&rest` *arguments* [*Function*]

Like `xcl:signal` except if the condition is not handled, the debugger is called with the given condition, and `error` never returns.

datum is treated as in `xcl:signal`. If *datum* is a string, a conditon of type `xcl:simple-error` is made. This form is compatible with that described in Steele's *Common Lisp, the Language*.

`cerror` *proceed-format-string datum* `&rest` *arguments* [*Function*]

Like `error`, if the condition is not handled the debugger is called with the given condition. However, `cerror` enables the proceed type `xcl:proceed`, which will simply return the condition being signalled from `cerror`.

`cerror` is used to signal continuable errors. Like `error`, it signals an error and enters the debugger. However, `cerror` allows the program to be continued from the debugger after resolving the error.

datum is treated as in `error`. If *datum* is a condition, then that condition is used directly. In this case, *arguments* will be used only with the *proceed-format-string* and will not be used to initialize *datum*.

The *proceed-format-string* must be a string. Note that if *datum* is not a string, then the format arguments used by the *proceed-format-string* will still be the *arguments* (in the keyword format as specified). In this case, some care may be necessary to set up the *proceed-format-string* correctly. The `format` directive `~*` may be particularly useful in this situation.

The value returned by `cerror` is the condition which was signaled.

See Steele's *Common Lisp, the Language,* page 430 for examples of the use of `cerror`.

`warn` *datum* `&rest` *arguments* [*Function*]

Invokes the signal facility on a condition. If the condition is not handled, then the text of the warning is output to `*error-output*`. If the variable `*break-on-warnings*` is true, then in addition to printing the warning, the debugger is entered using the function `break`. The value returned by `warn` is the condition that was signaled.

datum the same as for signal except that if *datum* is a string, a condition of type `xcl:simple-warning` is made.

The eventual condition type resulting from *datum* must be a subtype of `xcl:warning`.

`*break-on-warnings*` [*Variable*]

`check-type` [*Macro*]

`ecase` [*Macro*]

`ccase` [*Macro*]

`etypecase` [*Macro*]

`ctypecase` [*Macro*]

`assert` [*Macro*]

All of the above behave as described in *Common Lisp: the Language*. The default clauses of `ecase` and `ccase` forms signal `xcl:simple-error` conditions. The default clauses of `etypecase` and `ctypecase` forms signal `xcl:type-mismatch` conditions. `assert` signals the `xcl:assertion-failed` condition. `ccase` and `ctypecase` set up a `xcl:store-value` proceed case.

Handling Conditions

`xcl:handler-bind` *bindings* `&rest` *forms* [*Macro*]

Executes the forms in a dynamic context where the given local handler *bindings* are in effect. The *bindings* must take the form (*type handler*). The handlers are bound in the order they are given, i.e., when searching for a handler, the error system will consider the leftmost binding in a particular `xcl:handler-bind` form first.

type may be the name of a condition type or a list of condition types.

handler should evaluate to a function of one argument, a condition, to be used to handle a signalled condition during execution of the *forms*.

An example of the use of `xcl:handler-bind` appears at the end of the `xcl:proceed-case` macro description.

`xcl:condition-case` *form* `&rest` *cases* [*Macro*]

Executes the given *form*. Each *case* has the form

(*type* ([*var*]) . *body*)

If a condition is signalled (and not handled by an intervening handler) during the execution of the form, and there is an appropriate clause—i.e., one for which

(typep *condition* '*type*)

is true—then control is transferred to the body of the relevant clause, binding *var*, if present, to the condition that was signaled. If no condition is signaled, then the values resulting from the *form* are returned by the `xcl:condition-case`. If the condition is not needed, `var` may be omitted.

Earlier clauses will be considered first by the error system. I.e.,

```
(xcl:condition-case form
   (cond1 ...)
   (cond2 ...))
```

is equivalent to

```
(xcl:condition-case
        (xcl:condition-case form
           (cond1 ...))
   (cond2 ...))
```

type may also be a list of types, in which case it will catch conditions of any of the specified types.

Examples:

```
(xcl:condition-case (/ x y)
   (division-by-zero () nil))

(xcl:condition-case (open *the-file*
                          :direction :input)
   (file-error (condition)
      (format t "~&Open failed: ~A~%" condition)))

(xcl:condition-case (some-user-function)
   (file-error (condition) condition)
   (division-by-zero () 0)
   ((xcl:unbound-variable xcl:undefined-function) ()
      'unbound))
```

Note the difference between `xcl:condition-case` and `xcl:handler-bind`. In `xcl:handler-bind`, you are specifying functions that will be called in the dynamic context of the condition-signalling form. In `xcl:condition-case`, you are specifying continuations to be used instead of the original form if a condition of a particular type is signaled. These

continuations will be executed in the same dynamic context as the original form.

`xcl:ignore-errors` `&body` *forms* [*Macro*]

Executes the forms in a context that handles errors of type `error` by returning control to this form. If no error is signaled, all values returned by the last form are returned by `xcl:ignore-errors`. Otherwise, the form returns `nil` and the condition that was signaled. Synonym for

```
(xcl:condition-case (progn . forms)
   (error (condition)
      (values nil condition)).
```

`xcl:debug` `&optional` *datum* `&rest` *arguments* [*Function*]

Enters the debugger with a given condition without signalling that condition. When the debugger is entered, it will announce the condition by invoking the condition's report function.

datum is treated the same as for `xcl:signal` except if *datum* is not specified, it defaults to `"Call to DEBUG"`.

This function will never directly return to its caller. Return can occur only by a special transfer of control, such as to a `catch`, `block`, `tagbody`, `xcl:proceed-case` or `xcl:catch-abort`.

`break` `&optional` *datum* `&rest` *arguments* [*Function*]

Like `xcl:debug` except sets up a proceed case like `cerror`.

If *datum* is not specified, it defaults to `"Break"`.

If the `break` is proceeded, the value returned is the condition that was used.

`break` is approximately:

```
(defun break (&optional (datum "Break")
                &rest arguments)
   (xcl:proceed-case (apply #'xcl:debug datum
                                        arguments)
      (xcl:proceed (condition)
           :report "Return from BREAK."
       condition)))
```

Proceed Cases

`xcl:proceed-case` *form* `&rest` *clauses* [*Macro*]

The *form* is evaluated in a dynamic context where the clauses have special meanings as points to which control may be transferred. If *form* runs to completion, all values returned by the form are simply returned by the `xcl:proceed-case` form. On the other hand, the computation of forms may choose to transfer control to one of the proceed case clauses. If a transfer to a clause occurs, the forms in the body of that clause will be evaluated in the same dynamic context as the `xcl:proceed-case` form, and any values returned by the last such form will be returned by the `xcl:proceed-case` form.

A proceed case clause has the form:

(*proceed-function-name arglist* {*keyword value*}* {*body-form*}*)

The *proceed-function-name* may be `nil` or any symbol, usually the name of a defined proceed function. `xcl:define-proceed-function` will be described later.

The *arglist* is a list of optional argument specifications that will be bound and evaluated in the dynamic context of the `xcl:proceed-case` form. They will use whatever values were provided by `xcl:invoke-proceed-case`.

The valid *keyword/value* pairs are:

`:filter-function` *expression*

expression should be suitable as an argument to the `function` special form. It defines a predicate of no arguments that determines if this clause is visible to `xcl:find-proceed-function`.

`:filter` *form*

A shorthand form of `:filter-function` that is equivalent to

```
:filter-function (lambda () form)
```

`:condition` *type*

Shorthand for the common special case of `:filter`. The following two *key/value* pairs are equivalent:

```
:condition foo
:filter
   (lambda ()
      (typep xcl:*current-condition*
             'foo))
```

`:report-function` *expression*

The *expression* must be an appropriate argument to the `function` special form, and should designate a function of two arguments, a proceed case and a stream, that writes to the stream a summary of the action that this proceed case will take if invoked..

`:report` *form*

This is a shorthand for two important special cases of `:report-function`. If *form* is a constant string, then this is the same as:

```
:report-function
   (lambda (ignore stream)
      (write-string form stream))
```

Otherwise, this is the same as

```
:report-function
   (lambda (xcl:proceed-case
            *standard-output*)
      form)
```

In the latter case, form must do output to `*standard-output*`, summarizing the action that this proceed case will take if invoked. The proceed-case will be bound to the variable `xcl:proceed-case`.

Only one of `:condition`, `:filter` or `:filter-function` may be specified. Only one of `:report` or `:report-function` may be specified.

If a named proceed function has a default filter and the proceed case specifies a filter, then the information supplied in the proceed case takes precedence. Similarly, if `:report` or `:report-function` is specified in the proceed case, then only that information is considered, and any `:report` or `:report-function` specified as a default for the named proceed function is not used.

If a named proceed function is used but no report information is supplied, the name of the proceed function is used to generate the default help information. It is an error if no named proceed case is

used and no report information is provided; this means that you must always have a way of describing to the user how to proceed. If you don't specify report methods, make sure that the name of the proceed type is something sensible.

When `*print-escape*` is `nil`, the printer will use the report information for a proceed case.

Examples:

```
(xcl:proceed-case (a-random-computation)
   (new-function (new-function)
      (setq function new-function)))

(xcl:proceed-case (a-random-computation)
   (nil ((new-function (read-typed-object
                                  'function
                                  "Function: ")))
        :report "Use a different function."
        :condition undefined-function
      (setq function new-function)))

(xcl:proceed-case (a-command-loop)
   (return-from-command-level ()
         :report
            (format t
                 "Return from command level ~D."
                 level)
      nil))

(loop
   (xcl:proceed-case (another-computation)
      (xcl:proceed ())))
```

Assuming that `new-function` is defined as a proceed function with defaults:

```
:report "Use a different function."
:condition xcl:undefined-function
```

then the first and second examples are equivalent from the point of view of someone using the interactive debugger, but differ in one important aspect for non-interactive handling. If a handler "knows about" proceed function names, as in:

```
(when (xcl:find-proceed-case 'new-function
                             condition)
   (new-function condition the-replacement))
```

then only the first example, and not the second, will have control transferred to its correction clause.

Here's a more complete example:

```
(let ((my-food 'milk)
      (my-color 'greenish-blue))
   (do ()
       ((not (bad-food-color-p food
                               color)))
     (xcl:proceed-case (error 'bad-food-color
                              :food my-food
                              :color my-color)
        (use-food (new-food)
           (setf my-food new-food))
        (use-color (new-color)
           (setf my-color new-color))))
   ;; We won't get to here until my-food
   ;; and my-color are compatible.
   (list my-food my-color))
```

A handler can then proceed the error in either of two ways. It may correct the color or correct the food. For example:

```
#'(lambda (condition) ...
     ;; Corrects color
     (use-color 'white)  ...)
```

or

```
#'(lambda (condition) ...
     ;; Corrects food
     (use-food 'cheese) ...)
```

Here is an example using `xcl:handler-bind` and `xcl:proceed-case`.

```
(xcl:handler-bind ((foo-error
                          #'(lambda (condition)
                              (xcl:use-value 7))))
   (xcl:proceed-case (error 'foo-error)
      (xcl:use-value (x) (* x x))))
```

The above form returns 49.

`xcl:define-proceed-function` *name* [*Macro*]
{*keyword value*}*
{*variable*}*

Valid *keyword/value* pairs are the same as those which are defined for the `xcl:proceed-case` special form. That is, `:filter`, `:filter-function`, `:condition`, `:report`, and `:report-function`. The filter and report functions specified in a `xcl:define-proceed-function` form will be used for `xcl:proceed-case` clauses with the same name that do not specify their own filter or report functions, respectively.

This form defines a function called `name` which will invoke a proceed case with the same name. The proceed function takes optional arguments which are given by the *variables* specification. The parameter list for the proceed function will look like

`(&optional . variables)`

The only thing that a proceed function really does is collect values to be passed on to a proceed case clause.

Each element of *variables* has the form *variable-name* or (*variable-name initial-value*). If *initial-value* is not supplied, it defaults to `nil`.

For example, here are some possible proceed functions which might be useful in conjunction with the `bad-food-color` error we used as an example earlier:

```
(xcl:define-proceed-function use-food
      :report "Use another food."
   (food (read-typed-object 'food
                    "Food to use instead: ")))

(xcl:define-proceed-function use-color
      :report "Change the food's color."
   (color
      (read-typed-object 'food
           "Color to make the food: ")))

(defun maybe-use-water (condition)
   ;; A sample handler
   (when (eq (bad-food-color-food condition)
             'milk)
      (use-food 'water)))

(xcl:handler-bind ((bad-food-color
                             #'maybe-use-water))
   ...)
```

If a named proceed function is invoked in a context in which there is no active proceed case by that name, the proceed function simply returns `nil`. So, for example, in each of the following pairs of handlers, the first is equivalent to the second but less efficient:

```
#'(lambda (condition)            ; OK, but slow
     (when (xcl:find-proceed-case 'use-food)
        (use-food 'milk)))
#'(lambda (condition)            ; Preferred
     (use-food 'milk))

#'(lambda (condition)
```

```
                (cond ((xcl:find-proceed-case 'use-food)
                       (use-food 'chocolate))
                      ((xcl:find-proceed-case 'use-color)
                       (use-color 'orange))))
            #'(lambda (condition)
                (use-food 'chocolate)
                (use-color 'orange))
```

`xcl:compute-proceed-cases` [*Function*]

Uses the dynamic state of the program to compute a list of *proceed cases*.

Each proceed case object represents a point in the current dynamic state of the program to which control may be transferred. The only operations that Xerox Lisp defines for such objects are

`xcl:proceed-case-name`,
`xcl:find-proceed-case`,
`xcl:invoke-proceed-case`,
`princ`, and
`print`,

the identification of an object as a proceed case using `(typep x 'proceed-case)`, and standard Lisp operations that work for all objects, such as `eq`, `eql`, `describe`, etc.

The list which results from a call to `xcl:compute-proceed-cases` is ordered so that the innermost (ie, more-recently established) proceed cases are nearer the head of the list.

Note also that `xcl:compute-proceed-cases` returns *all* valid proceed cases, even if some of them have the same name as others and therefore would not be found by `xcl:find-proceed-case`.

`xcl:proceed-case-name` *proceed-case* [*Function*]

Returns the name of the given *proceed-case*, or `nil` if it is not named.

`xcl:default-proceed-test` *proceed-case-name* [*Macro*]

Returns the default filter function for proceed cases with the given *proceed-case-name*. May be used with `setf` to change it.

`xcl:default-proceed-report` *proceed-case-name* [*Macro*]

Returns the default report function for proceed cases with the given *proceed-case-name*. This may be a

string or a function just as for condition types. May be used with `setf` to change it.

`xcl:find-proceed-case` *name* [*Function*]

Searches for a proceed case by the given *name* which is in the current dynamic contour. This is determined by calling the proceed case's filter function.

If *name* is a proceed function name, then the innermost (ie, most recently established) proceed case with that function name that is active is returned. `nil` is returned if no such proceed case is found.

If *name* is a proceed case object, then it is simply returned unless it is not currently valid for use. In that case, `nil` is returned.

`xcl:invoke-proceed-case` *proceed-case* `&rest` *values* [*Function*]

Transfers control to the given *proceed-case*, passing it the given *values*. The *proceed-case* must be a proceed case object or the name of a proceed case which is valid in the current dynamic context. If the argument is not valid, the error `xcl:bad-proceed-case` will be signaled. If the argument is a named proceed case that has a corresponding proceed function, `xcl:invoke-proceed-case` will do the optional argument resolution specified by that function before transferring control to the proceed case.

`xcl:catch-abort` *print-form* `&body` *forms* [*Macro*]

Sets up a proceed case named `xcl:abort`.

If no call to the proceed function `xcl:abort` is made while executing *forms* and they return normally, all values returned by the last form in *forms* are returned. If an `xcl:abort` transfers control to this `xcl:catch-abort`, two values are returned: `nil` and the condition that was given to `xcl:abort` (or `nil` if none was given).

`xcl:catch-abort` could be defined by:

```
(defmacro xcl:catch-abort (print-form
                           &body forms)
  `(xcl:proceed-case (progn ,@forms)
     (xcl:abort (condition)
          :report ,print-form
       (values nil condition))))
```

Example:

```
(defun read-eval-print-loop (level)
   (xcl:catch-abort
         (format t "Exit command level ~D."
                   level)
      (loop
         (xcl:catch-abort
               (format t
                     "Return to command level ~D."
                     level)
            (print (eval (read)))))))
```

`xcl:abort &optional` *condition* [*Function*]

This is a predefined proceed function that transfers control to the innermost (dynamic) visible proceed case named `xcl:abort`.

`xcl:abort` could be defined by:

```
(define-proceed-function xcl:abort
   :report "Abort")
```

`xcl:proceed &optional` *condition* [*Function*]

This is a predefined proceed function. It is used by such functions as `break`, `cerror`, etc.

`xcl:use-value &optional` *new-value* [*Function*]

This is a predefined proceed function. It is intended to be used for supplying an alternate value to be used in a computation. If *new-value* is not provided, `xcl:use-value` will prompt the user for one.

`xcl:store-value &optional` *new-value* [*Function*]

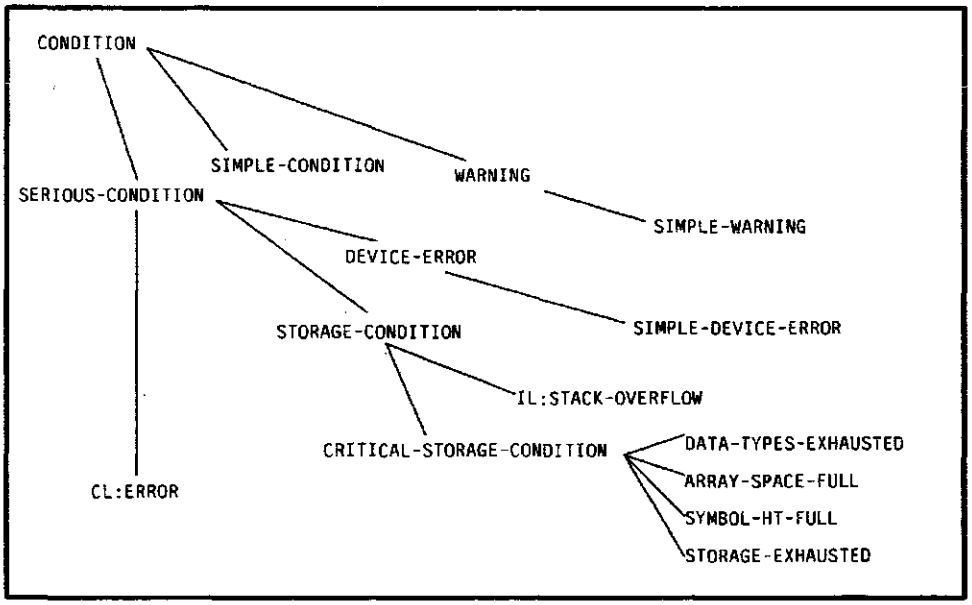
This is a predefined proceed function. It is intended to be used for supplying an alternate value to store in some location as a way of proceeding from an error. The proceed function `xcl:store-value` does not actually store the new value anywhere: it is up to proceed case to take care of that. If *new-value* is not provided, `xcl:store-value` will prompt the user for one. `xcl:store-value` is used by such forms as `check-type` and `cerror`.

Predefined Types

`xcl:proceed-case` *[Type]*

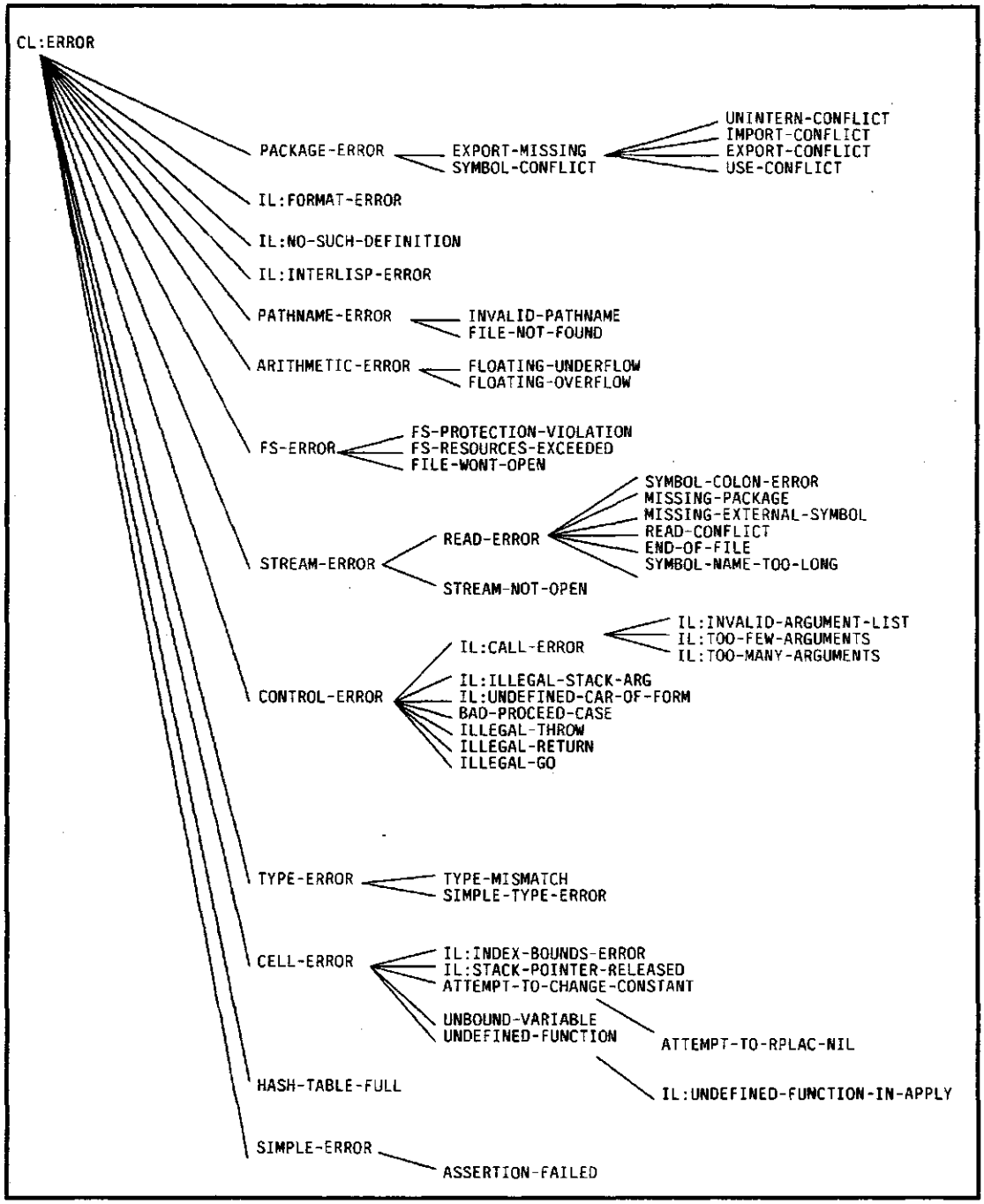
This is the data type used to represent a proceed case.

The condition type hierarchy looks like this:



All condition types shown in the graph above, and in the one that follows, are in the XCL package, unless otherwise qualified.

The hierarchy continues on the next page.


CL:ERROR
PACKAGE-ERROR
EXPORT-MISSING
SYMBOL-CONFLICT
UNINTERN-CONFLICT
IMPORT-CONFLICT
EXPORT-CONFLICT
USE-CONFLICT
IL:FORMAT-ERROR
IL:NO-SUCH-DEFINITION
IL:INTERLISP-ERROR
PATHNAME-ERROR
INVALID-PATHNAME
FILE-NOT-FOUND
ARITHMETIC-ERROR
FLOATING-UNDERFLOW
FLOATING-OVERFLOW
FS-ERROR
FS-PROTECTION-VIOLATION
FS-RESOURCES-EXCEEDED
FILE-WONT-OPEN
STREAM-ERROR
READ-ERROR
SYMBOL-COLON-ERROR
MISSING-PACKAGE
MISSING-EXTERNAL-SYMBOL
READ-CONFLICT
END-OF-FILE
SYMBOL-NAME-TOO-LONG
STREAM-NOT-OPEN
CONTROL-ERROR
IL:CALL-ERROR
IL:INVALID-ARGUMENT-LIST
IL:TOO-FEW-ARGUMENTS
IL:TOO-MANY-ARGUMENTS
IL:ILLEGAL-STACK-ARG
IL:UNDEFINED-CAR-OF-FORM
BAD-PROCEED-CASE
ILLEGAL-THROW
ILLEGAL-RETURN
ILLEGAL-GO
TYPE-ERROR
TYPE-MISMATCH
SIMPLE-TYPE-ERROR
CELL-ERROR
IL:INDEX-BOUNDS-ERROR
IL:STACK-POINTER-RELEASED
ATTEMPT-TO-CHANGE-CONSTANT
UNBOUND-VARIABLE
UNDEFINED-FUNCTION
ATTEMPT-TO-RPLAC-NIL
IL:UNDEFINED-FUNCTION-IN-APPLY
HASH-TABLE-FULL
SIMPLE-ERROR
ASSERTION-FAILED

The types that are non-terminals in the above tree:
`xcl:condition`,
`xcl:warning`,
`xcl:serious-condition`,
`xcl:storage-condition`,
`error`,
`xcl:control-error`, etc.
are provided primarily for type inclusion purposes. Normally, they would not be directly instantiated.

In the descriptions of condition types below, the names in italics on the first line of each description are the names of the slots defined for that condition type.

`xcl:condition` [*Condition*]

All types of conditions, whether `error` or non-`error`, must inherit from this type.

`xcl:warning` [*Condition*]

All types of warnings should inherit from this type. This is a subtype of `condition`.

`xcl:serious-condition` [*Condition*]

Any condition, whether error or non-error, which should enter the debugger when signalled but not handled should inherit from this type. This is a subtype of `xcl:condition`.

Note: `ignore-errors` will ignore conditions of type `error`, not of type `xcl:serious-condition`. Conditions which are serious conditions but not errors are typically those that may require more sophisticated handling than simply being ignored. For example, `xcl:ignore-errors` will not ignore an `xcl:storage-condition`, which is a serious condition but is not generally a program error.

Compatibility Note: `serious-condition` is similar to Zetalisp's `dbg:debugger-condition`.

`error` [*Condition*]

All types of error conditions inherit from this condition. This is a subtype of `xcl:serious-condition`.

`xcl:simple-condition` *format-string format-arguments* [*Condition*]

Conditions signalled by `xcl:signal` when given a format string as a first argument are of this type. This is a subtype of `xcl:condition`.

`xcl:simple-warning` *format-string format-arguments* [*Condition*]

Conditions signalled by `warn` when given a format string as a first argument are of this type. This is a subtype of `xcl:warning`.

`xcl:simple-error` *format-string format-arguments* [*Condition*]

Conditions signalled by `error` and `cerror` when given a format string as a first argument are of this type. This is a subtype of `error`.

`xcl:storage-condition` [*Condition*]

Conditions which relate to memory overflow conditions should inherit from this type. This is a subtype of `xcl:serious-condition`.

`xcl:stack-overflow` [*Condition*]

Conditions which relate to stack overflow should inherit from this type. This is a subtype of `xcl:storage-condition`.

`xcl:control-error` [*Condition*]

Errors in the transfer of control in a program should inherit from this type. This is a subtype of `error`.

`xcl:illegal-throw` *tag* [*Condition*]

The error which results when `throw` is given a tag which is not active should inherit from this. This is a subtype of `xcl:control-error`. *tag* is the offending tag.

`xcl:illegal-go` *tag* [*Condition*]

The error which results when `go` is given a tag which is no longer available should inherit from this. This is a subtype of `xcl:control-error`. *tag* is the offending tag.

`xcl:illegal-return` *tag* [*Condition*]

The error which results when `return-from` is given a block name which is no longer accessible should inherit from this. This is a subtype of `xcl:control-error`. *tag* is the offending block name.

`xcl:stream-error` *stream* [*Condition*]

Errors which occur during input from or output to a stream should inherit from this type. This is a subtype of `error`. The function `stream-error-stream` will access the offending stream.

`xcl:read-error` [*Condition*]

Errors which occur during an input operation on a stream should inherit from this type. This is a subtype of `xcl:stream-error`.

`xcl:end-of-file` [*Condition*]

The error which results when a read operation is done on a stream which has no more tokens should inherit from this type. This is a subtype of `read-error`.

`xcl:cell-error` *name* [*Condition*]

Errors which occur while accessing a location should inherit from this type. This is a subtype of `error`. *name* is the name of the offending cell.

`xcl:unbound-variable` [*Condition*]

The error which results from trying to access the value of an unbound variable should inherit from this type. This is a subtype of `xcl:cell-error`.

`xcl:undefined-function` [*Condition*]

The error which results from trying to access the value of an undefined function should inherit from this type. This is a subtype of `xcl:cell-error`.

[This page intentionally left blank]

CHAPTER 25
MISCELLANEOUS FEATURES

25.1. The Compiler

There are two entry points to the XCL Compiler, one for compiling a single function already in memory, and one for compiling a file full of code.

`compile` *name* &optional `definition &key :lap` [*Function*]

Works precisely as defined in *Common Lisp: the Language*. Xerox Common Lisp provides an additional &key parameter `:lap` which, if non-`nil`, causes the compiler to pretty-print the Lisp Assembly Program input to `*standard-output*`. This is the Xerox Lisp equivalent of the assembly-language code produced by compilers for other languages. This code is primarily only of use when debugging the compiler, but users may find it interesting on occasion.

Xerox Common Lisp Extensions to Section 25.1

`compile-file` *input-file* `&key :output-file` [*Function*]
`:error-file`
`:errors-to-terminal`
`:lap-file`
`:load`
`:file-manager-format`
`:process-entire-file`

Compiles the forms on *input-file* and produces a DFASL file containing the compiled code. The keyword arguments are as follows:

`:output-file`

The name of the file on which the DFASL output should be written. Defaults to *input-file* but with the extension "dfasl."

`:error-file`

The name of the file on which warnings and error messages from the compiler should be printed. If the value is `nil`, the messages will not be saved on a file. If the value is `t`, the file name is *input-file* but with the extension "log." The default value is `nil`.

`:errors-to-terminal`

If non-`nil`, warnings and error messages from the compiler will, in addition to being saved on any error file, be printed on `*error-output*`, usually bound to a display stream. Defaults to `t`.

`:lap-file`

If an explicit file name is given as the value for `:lap-file`, the compiler will pretty-print all Lisp Assembly Program (LAP) input to that file before sending the LAP input to the assembler. If the value is `t`, the file name is *input-file* but with the extension "dlap." If the value is `nil`, no LAP code will be printed. The default value is `nil`.

`:load`

If non-`nil`, all code will be loaded into the environment after it is compiled. If the value is `:save`, then any previous contents of changed function definition cells will be saved on the `il:expr` property of the symbol. This saving will not be done if the symbol has a File Manager `functions` definition.

`:file-manager-format`

If non-`nil`, the compiler will assume that *input-file* is produced by the File Manager and will process it accordingly. The default value is `t` if (and only if) the first non-blank character on the *input-file* is "(" (a left parenthesis).

If `nil`, the compiler will assume that the file is a standard Common Lisp source file produced by a text editor, either in Xerox Lisp or in another implementation.

`:process-entire-file`

If non-`nil`, the compiler will read through the entire file, looking for implicitly or explicitly `(eval-when (compile)...)` forms, evaluating them as it finds them. Afterwards, the forms on the file will be compiled. This behavior allows the user to put macros, proclamations, and other compile-time forms anywhere on the file, not necessarily before any uses

of them. This option defaults to the value of the `:file-manager-format` option.

Supported Features of the Interlisp Compiler

The XCL Compiler will compile programs written in Common Lisp, Interlisp, or a combination of the two. In particular, the following features of the old Interlisp Compiler are supported by the XCL Compiler (refer to the *Interlisp Reference Manual* for details on their use):

- `il:localvars`, `il:globalvars`, and `il:specvars` declarations
- The special forms `il:constant`, `il:deferredconstant` and `il:loadtimeconstant`
- The lists `il:nlama`, `il:nlaml` and `il:dontcompilefns`
- Block compilation
- Macros defined on the `il:macro` property of a symbol

Unsupported Features of the Interlisp Byte Compiler

The XCL Compiler does not support the following features of the old Interlisp Compiler:

- The XCL Compiler will not ask the user any of the questions asked by the Interlisp Compiler. The function `il:compset` is never called.
- The function `il:dassem.savelocalvars` is never called.
- The variable `il:compileuserfn` is never examined. The compilation of Interlisp's iterative statements and IF-THEN-ELSE statements is achieved through the normal macro-expansion process.
- Macros defined on the `il:dmacro` or `il:bytemacro` properties of symbols are ignored by the XCL Compiler. The new `xcl:defoptimizer` facility should be used instead (see below).
- The list `il:compiletypelst` is not consulted. Non-list, non-symbol data encountered during compilation are treated as though they had been quoted; that is, such data are considered self-evaluating.

- The variable `il:dwimifycompflg` is not consulted. The XCL Compiler does not call the `il:dwimify` function and thus does not properly treat code that requires such treatment.

Compiler Optimizers: The XCL:Defoptimizer Facility

Xerox Lisp provides a facility that allows you to advise the compiler about efficient compilation of certain functions and macros. This facility works both with the old Interlisp Compiler and with the new XCL Compiler.

An *optimizer* is, to a rough approximation, a macro that is only invoked at compile-time and which takes precedence over any normal macro definition that might exist for the form. Unlike normal macros, optimizers should not be used for the definition of new language features; they are only understood by the compiler and thus will not be recognized in interpreted code. The usual paradigm involves the use of `defun` or `defmacro` to define the general case of a new form and the definition of optimizers to take advantage of common special cases.

Optimizers have access to the lexical environment and *compilation-context* of the form. The latter is a representation of certain information about the use to which the value of the form will be put; for example, whether or not the value will be used and, if so, how many values are expected.

The compiler uses optimizers to encode many of the source-to-source transformations it employs; you can add to this store of knowledge to achieve improved performance of both built-in Xerox Common Lisp constructs and new, user-written ones.

The compilers also provide a set of facilities for accessing the information carried in the lexical environment objects passed to macros and optimizers via the &environment lambda-list keyword. It is possible both to make queries on that object and to create new ones which only differ on a given set of names.

Defining optimizers

New optimizers are defined using the following macro:

`xcl:defoptimizer` *form-name* [*opt-name*] [*arg-list* [*decl* | *doc-string*]* *body*] [*Macro*]

form-name is an optional symbol which is the `car` of forms to which this optimizer should be applied.

opt-name is a symbol used as the name of the function created to perform the optimization (for purposes of breaking, advising, etc.).

arg-list is a standard defmacro argument list, allowing the usual &environment keyword and one more: `&context` *ctxt*. *ctxt* is a variable to be bound to a value that can be queried for information about the evaluation context of the form. For example, it is possible to determine whether or not the given form is being evaluated for effect or value. Some optimizers produce different expansions under different conditions.

The *arg-list* and *body* may be simultaneously omitted, in which case *opt-name* should name a previously-defined function of three arguments: a form, an environment object, and a compilation context. Previously-defined optimizers may be used for this purpose, allowing the user to specify a single optimizer for a large number of kinds of forms.

It is possible for more than one optimizer to be defined for the same *form-name*; new ones are added to a list and do not replace any previous ones. The only exception to this is when a new optimizer is defined for the same *form-name* and *opt-name* as an earlier one; in this case, the old optimizer is replaced by the new one. Note that no guarantees are made about the order of the optimizers in the list; optimizations should not depend upon whether or not other optimizations have been performed.

The `xcl:defoptimizer` form produces a File Manager definition of type optimizers. The name of the definition is the list (*form-name* `:optimized-by` *opt-name*) unless no *opt-name* was given, in which case the definition is named simply *form-name*.

The compiler, in considering a new form, first looks to see if any optimizers are defined for the `car` of the

form. If so, they are each applied in turn. An optimizer may refuse to change the form by returning any one of the following three values:

1. The symbol `compiler:pass`.
2. A form `eq` to the given one. To do this, the argument-list of the optimizer must have specified the `&whole` keyword.
3. The symbol `il:ignoremacro`. This is provided purely for backward compatibility with Interlisp-D macros.

If an optimizer returns one of these values, the compiler will move on to the next one on the list. Whenever an optimizer does not return one of these (that is, it actually performs an optimization), the compiler begins the whole process anew, starting with the first optimizer on the list for the new car of the returned form. This allows optimizers to produce forms which themselves have optimizers.

If all of the optimizers on the list have refused to change the form, the compiler will finally check for an ordinary macro definition, as produced by defmacro. This priority of optimizers over macros allows you to put optimizers on macros.

Examples

The following simple optimizer changes (`eq` *form* `nil`) into (`not` *form*):

```
(xcl:defoptimizer eq eq-nil-check (&whole form)
    (cond ((eq nil (second form))
                `(not ,(third form)))
          ((eq nil (third form))
                `(not ,(second form)))
          (t form)))
```

Note the return of the input form as a refusal to apply the optimization. A slightly more complex optimizer, actually in use in the system, open codes calls to the function `nth` when given a small integer argument:

```
(xcl:defoptimizer nth (n-form list-form)
    (if (and (typep n-form 'fixnum)
             (<= 0 n-form 10))
        `(car ,(let ((cdr-form list-form))
                 (dotimes (i ,n-form cdr-form)
                    (setq cdr-form '(cdr ,cdr-form)))))
        'compiler:pass))
```

Operations on Compilation Contexts

Optimizers can arrange to be passed a *compilation-context* argument. This value encodes information about the position of the given form in the code around it. The following functions can access that information (all of these functions are in the compiler package). For brevity in what follows, we will refer to "the form", meaning the form that was passed to the optimizer along with the given context.

`compiler:context-top-level-p` *ctxt* [*Function*]

Returns true if and only if the form appears at "top level" in the file being compiled.

`compiler:context-values-used` *ctxt* [*Function*]

Returns the number of values the surrounding code expects from the form. This is 0 if the form will be evaluated for effect, a positive integer if a specific number of values are expected, and `:unknown` if the compiler is unable to tell how many will be used. Forms providing the returned value of a function or occurring in the arguments to the `multiple-value-call` special form can cause this latter condition.

`compiler:context-predicate-p` *ctxt* [*Function*]

Returns true if and only if the form appears in a context in which only the `nil`-ness of the value is used, as in the predicate position of an `if`. In general, `context-predicate-p` will only be true of contexts for which `context-values-used` returns 1.

```
make-context &key (top-level-p nil)                    [Function]
                  (values-used :unknown)
                  (predicate-p nil)
```

Creates a new context object with the given properties.

Examples

Information about the context of a form can come in handy for functions that return multiple values. For example, the following might be a worthwhile optimizer on the `floor` function, which normally returns two values: the result of the rounding and the remainder. This code checks for the (frequent) case in

which the remainder is unused and translates the call to `floor` into a call on a (hypothetical) lower-level function which does not compute it.

```
(xcl:defoptimizer floor (&whole form &context ctxt)
    (if (= (context-values-used ctxt) 1)
    `(lisp::%div-floor ,@(cdr form))
    'compiler:pass))
```

Another example uses the context of a call to `intersection` to decide whether or not it is really necessary to cons together the result; if the call is being used as a predicate, a faster and more storage-efficient version can be substituted instead:

```
(xcl:defoptimizer intersection intersection-predicate
            (&whole form &context ctxt)
    (if (context-predicate-p ctxt)
    `(lisp::%share-a-member-p ,@(cdr form))
    form))
```

Operations on Lexical Environment Objects

The `&environment` values optionally passed to macros and optimizers are entirely unspecified in Common Lisp. No operations exist on them and it is not possible for the user to create or change one. It is frequently the case that an optimizer can produce better expansions given access to the lexical environment information contained in such values. The following functions implement that access:

`compiler:env-boundp` *env var* [*Function*]

Returns `:global`, `:special` or `:lexical`, as appropriate, if the symbol *var* is either bound or declared as a variable in the environment *env*. If *var* is not bound or declared in a lexically-apparent place, `env-boundp` returns `nil`.

`compiler:env-fboundp` *env fn* [*Function*]

Returns either `:function` or `:macro`, as appropriate, if and only if the symbol *fn* is bound as a function (in `flet` or `labels`) or macro (in `macrolet`) in the environment *env*. If `:macro` is returned, then a second value is also returned, the expansion function for the macro definition. If *fn* is not bound in a lexically-apparent place, `env-fboundp` returns nil.

`compiler:make-empty-env` [*Function*]

Returns an environment on which `env-boundp` and `env-fboundp` always return `nil`.

`compiler:copy-env-with-var` *env var* `&optional` (*kind* `:lexical`) [*Function*]

Returns a copy of *env* in which the symbol *var* is bound as a variable of kind *kind*. It is an error for *kind* to be `nil` or a value not returnable by `env-boundp`. The *env* may be given as `nil`, in which case it is equivalent to passing the result of calling `make-empty-env`.

`compiler:copy-env-with-fn` *env fn* `&optional` (*kind* `:function`) *exp-fn* [*Function*]

Returns a copy of *env* in which the symbol *fn* is bound as a function or macro, depending upon the value of *kind*. It is an error for *kind* to be `nil` or a value not returnable by `env-fboundp`. If *kind* is `:macro`, then *exp-fn*, an expansion function taking a form and an environment and returning a new form, must be provided. The *env* may be given as `nil`, in which case it is equivalent to passing the result of calling `make-empty-env`.

Expanding Compiler Optimizers

The following two functions are available for use in expanding compiler optimizers under program control.

`compiler:optimize-and-macroexpand` *form env ctxt* [*Function*]

`compiler:optimize-and-macroexpand-1` *form env ctxt* [*Function*]

Analagous to the functions `macroexpand` and `macroexpand-1` of Common Lisp, these entries into the compiler perform expansion of compiler optimizers and normal macros on the given form. The first function will apply such expansions until none are possible while the second will expand the form at most once. Both functions return two values: the new form (or the old one if nothing was done) and either `t` or `nil`, depending upon whether or not any expansions actually took place.

25.2. Documentation

Anything that was created with `xcl:def-define-type` can have documentation.

25.3. Debugging Tools

Breaking, Tracing and Advising: the Wrappers Facility

Xerox Common Lisp greatly extends the trace facility described in *Common Lisp: the Language* and adds two more extremely useful tools for debugging: setting breakpoints and advising existing functions. Collectively, these three tools are known as the *Wrappers* facility.

Concepts Common to Breaking, Tracing and Advising

As might be guessed from the name, the Wrappers facility works by encapsulating a named function definition in a new function. The new function can control when and how the original function is called and can specify other actions to occur around that call. The different aspects of the Wrappers facility (breaking, tracing and advising) specify different sets of actions, depending upon their individual semantics. For example, the tracing facility simply arranges to print out certain information before and after calling the original function. All encapsulating code, including any provided by the user, is compiled before installation. Thus, little or no performance penalty is paid for use of the Wrappers facility. Note that, if the original function is running interpreted, it will remain so; only the encapsulation will be compiled.

There are two ways to specify the function upon which the Wrappers facility will operate. In the simpler of the two, the user passes a symbol naming the function. All calls to this function, from anywhere in the system, will be affected by the encapsulation. For cases in which such a widespread effect would be either unsafe or otherwise undesirable, the user may specify the precise set of functions whose calls should be affected. The **:in** argument to the Wrappers functions is used for specifying this list. For example, to have tracing output printed every time the

function **foo** is called from any of the functions **bar**, **baz**, and **bax**, the following call should be used:

```
(xcl:trace-function 'foo :in '(bar baz bax))
```

The **:in** argument may be given as a symbol if only one function is to be specified.

The breaking and advising aspects of the Wrappers facility allow for the specification of arbitrary expressions to be evaluated under certain conditions. Such expresions are evaluated in a lexical environment that depends upon the kind of function being wrapped. The following lays out the rules for determining what variables are lexically available:

Interlisp functions:

Lambda spread functions (ARGTYPE 0)

Expressions in wrapped lambda spread functions may refer to and set any of the arguments to the original function by the names given in the original function's definition.

NLambda spread functions (ARGTYPE 1)

As with lambda spread functions, expressions in wrapped nlambda spread functions may refer to and set any of the arguments to the original function by the names given in the original function's definition.

Lambda no-spread functions (ARGTYPE 2)

Because compiling a lambda no-spread loses information, the lexical environment of expressions in wrapped lambda no-spread functions is different for the interpreted and compiled cases.

When the original function is interpreted, expressions may refer to the named parameter specified in the function definition. The Interlisp functions **il:arg** and **il:setarg** may be used with that parameter to examine and change the arguments that were passed to the wrapped function and will be passed to the original function.

When the original function is compiled, the name of the original parameter has, in general, been lost. As a result, expressions must use the name **il:u** instead of the one used in the original function's definition. As in the interpreted case, this variable may be passed to **il:arg** and **il:setarg** to access and change the arguments.

NLambda no-spread functions (ARGTYPE 3)

Expressions in wrapped nlambda no-spread functions may refer to and set the argument using the name given in the original function's definition.

Common Lisp functions:

Because of semantic difficulties involving the treatment of `&optional` and `&key` parameters with default values and associated supplied-p parameters, expressions in wrapped Common Lisp functions have access to the arguments via the single `&rest` parameter `xcl:arglist`. The elements of this list may be examined and the value of `xcl:arglist` changed in order to modify the arguments that will be passed to the original function. In the Lyric release of Xerox Lisp, it is safe to destructively modify the list in `xcl:arglist`; it is guaranteed to be freshly-consed and thus not to share structure with any other list.

As an example, consider a function with the following parameter list:

```
(a &optional b &rest c &key d e)
```

An expression in a wrapped version of this function could use the following expressions to discover the values of the five different parameters:

```
a  --  (first xcl:arglist)
b  --  (if (null (cdr xcl:arglist))
           nil
           (second xcl:arglist))

c  --  (cddr xcl:arglist)
d  --  (getf (cddr xcl:arglist) :d)
e  --  (getf (cddr xcl:arglist) :e)
```

The following expression could be used to provide the value 17 for `b` in the case where no value was supplied:

```
(if (null (cdr xcl:arglist))        ; b was not
supplied
   (setq xcl:arglist (list (first xcl:arglist)
                           17)))
```

Finally, the following expression could be used after the one above to either provide the value 0 for the `:d` keyword if none was supplied or to increase by 1 the value that was supplied:

```
(cond ((null (cddr xcl:arglist))    ; No keywords were supplied
       (setq xcl:arglist (nconc xcl:arglist (list :d 0))))
      ((null (getf (cddr xcl:arglist) :d))
                                    ; There are keywords, but
                                    ; not :d.
       (setf (getf (cddr xcl:arglist) :d)
             0))
      (t                            ; The keyword :d was
                                    ; supplied.
       (incf (getf (cddr xcl:arglist) :d))))
```

The mechanism for accessing arguments to wrapped Common Lisp functions may be revised in a future release.

Breaking: Setting Debugger Breakpoints

Common Lisp provides only one way for a user to intentionally and cleanly enter the debugger, the function `break`. While it is possible for users to insert calls to this function at desired locations in their code, this is not generally convenient, especially when the code to contain the breakpoint is compiled or was written by others, such as Xerox Lisp system code. The breakpoint facility allows the user to specify a function as described above and arranges for calls to that function to enter the debugger before actually making the call. The user can then examine the arguments passed to the function and the general state of the computation. Afterwards, the debugger's `ok` command can be used to continue the computation by executing the originally-intended function-call. Alternatively, the user could choose to abort the computation using the ↑ command or other means. This style of setting breakpoints is known as *breaking* the designated function.

It is sometimes desirable to exert some control over whether or not a particular breakpoint activates (i.e., actually enters the debugger) before calling the function. The breaking facility allows for the specification of an arbitrary expression to be evaluated to determine whether or not the debugger entry should occur. If the given expression returns a non-nil value, the debugger is entered as usual. Otherwise, the program behaves as if no breakpoint were set and calls the broken function. Such conditionalizing expressions are known as *break-when* expressions. The lexical environment available to break-when expressions is as described in the general discussion of Wrappers above.

The breaking facility allows the specification of a special kind of breakpoint, the *one-shot* breakpoint. Such breakpoints are guaranteed to activate exactly once, the first time they are encountered. This feature can be extremely useful when setting breakpoints in functions used by the debugger, such as those that open windows, compute backtraces, etc. If a normal breakpoint was used, an infinite recursion would result, with the debugger repeatedly calling itself in order to respond to the breakpoint. One-shot breakpoints avoid this problem.

`xcl:break-function` *fn-to-break* `&key :in (:when t)` [*Function*]

Breaks the designated *fn-to-break* as described earlier, unbreaking it first if it was already broken. The `:in` argument may be used as specified in the general Wrappers description above. The `:when` argument is used for specifying a break-when expression; the expression defaults to `t`. If the `:when` argument is given as `:once`, a one-shot breakpoint is installed.

`xcl:unbreak-function` *fn-to-unbreak* `&key :in :no-error` [*Function*]

Restores the designated *fn-to-unbreak* to its original, unbroken state. The `:in` argument may be used as specified in the general Wrappers description above. If the designated function is not broken, an error message is printed, unless the `:no-error` argument is specified and non-`nil`.

`xcl:rebreak-function` *fn-to-rebreak* `&key :in` [*Function*]

Breaks the designated *fn-to-rebreak* using the same break-when expression as was used the last time it was broken. The function is unbroken first if it was already broken. The `:in` argument may be used as specified in the general Wrappers description above.

The following functions comprise the original Interlisp-D interface to the breaking facility. They are provided in the Lyric release for backward compatibility. Existing user programs employing the breaking facility should be changed to use the new functions, described above. The old interface may be eliminated in a future release.

`il:break0` *fn* `&optional` (*when* `t`) [*Function*]

If *fn* is a symbol, this is equivalent to

```
(xcl:break-function fn :when when)
```

If *fn* is a list of the form (*fn1* **in** *fn2*), this is equivalent to

```
(xcl:break-function fn1 :in fn2 :when when)
```

Otherwise, *fn* should be a list and `il:break0` is called recursively on each member of *fn*, all with the given value of *when*.

`il:break` *x* [*NLambda No-Spread Function*]

For each argument that is either a symbol or a list in the form (*fn1* **in** *fn2*), the call `(il:break0` *arg* `t)` is performed. Each other list argument is used as the arguments in a call to `il:break0`; that is, the call `(apply 'il:break0` *arg*`)` is performed. For example,

```
(il:break foo (bax (> n 2)))
```

is equivalent to

```
(progn (il:break0 'foo t)
       (il:break0 'bax '(> n 2)))
```

`il:unbreak0` *fn* [*Function*]

If *fn* is a symbol, this is equivalent to

```
(xcl:unbreak-function fn)
```

Otherwise, *fn* should a list in the form (*fn1* **in** *fn2*) in which case this is equivalent to

```
(xcl:unbreak-function fn1 :in fn2)
```

`il:unbreak` *fns* [*NLambda No-Spread Function*]

All of the arguments should be either symbols or lists in the form (*fn1* **in** *fn2*). This is equivalent to calling `il:unbreak0` on each of the arguments. If no arguments are given, this is equivalent to calling `xcl:unbreak-function` on all functions currently broken, in reverse order of their breaking. If exactly one argument, **t**, is given, the most-recently broken function is unbroken.

`il:rebreak` *fns* [*NLambda No-Spread Function*]

For each argument that is a symbol, this is equivalent to

```
(xcl:rebreak-function arg)
```

For each argument that is a list in the form (*fn1* **in** *fn2*), this is equivalent to

```
(xcl:rebreak-function fn1 :in fn2)
```

If no arguments are given, all functions that have ever been unbroken are rebroken, in reverse order of being unbroken. If exactly one argument, `t`, is given, the most-recently unbroken function is rebroken.

Tracing: Recording Function Calls and Returns

The tracing aspect of Wrappers provides for recording, in a human-readable fashion, all of the calls to and returns from a given set of functions. On entry to the affected functions, information is printed giving the name of the function and the names and values of all of the arguments. On exit, more is printed, including, again, the name of the function and the value (or values) returned. The information from nested calls to traced functions is printed indented under the entry information for the outer calls.

For backward compatibility with Interlisp-D, tracing is treated in some ways as a special case of breaking. In particular, the functions `il:unbreak`, `il:unbreak0`, and `xcl:unbreak-function` will serve to turn off tracing on a given function. Also, the functions `xcl:rebreak-function` and `il:rebreak` and `xcl:rebreak-function` will restore a function to its traced state. This special-casing behavior is likely to change in future releases.

`xcl:trace-function` *fn-to-trace* `&key :in` [*Function*]

Traces the designated *fn-to-trace*. The **:in** argument may be used as specified in the general Wrappers description above. If the function was broken, it is first unbroken.

`trace` {*fn*}* [*Macro*]

For each argument given, if *fn* is a symbol naming a function, this is equivalent to

```
(xcl:trace-function fn)
```

If *fn* is a list in the form (*fn1* **:in** *fn2*), this is equivalent to

```
(xcl:trace-function fn1 :in fn2)
```

If no arguments are given, this returns a list of all functions currently traced.

`untrace` {*fn*}* [*Macro*]

For each argument given, if *fn* is a symbol naming a function, this is equivalent to

```
(xcl:unbreak-function fn)
```

If *fn* is a list in the form (*fn1* :**in** *fn2*), this is equivalent to

```
(xcl:unbreak-function fn1 :in fn2)
```

If no arguments are given, all functions currently traced are untraced.

All tracing information is printed to the stream that is the current value of `*trace-output*`. Initially, `*trace-output*` is bound to a window named "*Trace-Output*". This window will pop up whenever tracing output is printed; it can be closed whenever it is not needed. Should users have a need to create a new tracing window, the function `xcl:create-trace-window` is provided.

`xcl:create-trace-window &key (:region nil:traceregion) (:open? nil) (:title "*Trace-Output*")` [*Function*]

Creates and returns a window suitable for the value of `*trace-output*`. The `:region` argument is used for the location and size of the window; it defaults to the value of the variable `il:traceregion`, initially an area in the lower left corner of the display. If the `:open?` argument is non-`nil`, the window is opened immediately; otherwise, it will stay closed until the first time tracing information is printed to it. The `:title` argument provides the title for the window.

Three variables are provided to allow the user to customize the format of the tracing information.

`xcl:*trace-length*` [*Variable*]
`xcl:*trace-level*` [*Variable*]

During the printing of the values of arguments and the returned values of traced functions the printing-control variables `*print-length*` and `*print-level*` are bound to the values of these variables. Both are initially set to `nil`.

`xcl:*trace-verbose*` [*Variable*]

Certain non-essential parts of the tracing information are printed only when the value of `xcl:*trace-verbose*` is non-`nil`. In the Lyric

release of Xerox Lisp, the following pieces of information are so controlled:

- The lambda-list keywords `&optional`, `&rest`, and `&key`, normally printed as separators between the various kinds of arguments.
- Trailing unsupplied `&optional` arguments, normally printed as "*name* unsupplied".

Initially, `xcl:*trace-verbose*` is set to `t`.

Advising: Modifying the Behavior of Functions

The most powerful aspect of the Wrappers facility is advice. The advising aspect is sufficiently expressive to be able to subsume the other two, breaking and tracing. With it, the user can specify arbitrary expressions to be evaluated before, after or around the body of the original function. Here are some ways in which advice has been used to advantage:

- Changing the effective default value of an argument or even overriding supplied values when they are in some way unsatisfactory.
- Binding certain special variables around all calls to a given function.
- Customizing the behavior of certain system functions for individual users.
- Building breaking or tracing interfaces that go beyond the facilities described above, suited specially to local circumstances.

It is possible for a given function to have more than one piece of advice attached to it simultaneously. If `xcl:advise-function` is called when the designated function is already advised, the new advice is added to that already existing. The relative ordering of multiple pieces of advice is controlled by the `:priority` attributes of the pieces of advice involved, described below. Multiply-advised functions have but one "layer" of wrapping around them; all of the advice has been merged into a single whole.

There are three important attributes for a given piece of advice, `:when`, `:priority`, and the advice-expression itself.

`:when` can be one of `:before`, `:after`, or `:around` as described below:

`:before` advice is executed before the original function is called. It may examine and/or change the values of arguments passed to the function and can even avoid the call to the original function, specifying the values to be returned. More simply, it can also specify independent actions to be performed before calling the original function. When more than one piece of `:before` is present, they are executed one after another in order. Thus, later advice can be affected by the workings of earlier.

`:after` advice is executed after the original function has been called. It may examine and/or change the value (or values) to be returned by the function. More simply, it can also specify independent actions to be performed after calling the original function. As with the previous case, when more than one piece of `:after` advice is present, they are executed one after another in order.

`:around` advice is literally wrapped around the call to the original function. The advice-expression can contain one or more calls to the macro `xcl:inner`; these specify the locations of calls to the original function. Thus, `:around` advice can be used, for example, to bind special variables around the original call, or to conditionally avoid calling the original function. When multiple pieces of `:around` advice are present, earlier ones are nested inside later ones.

For backward compatibility with Interlisp-D, the symbol `il:*` may be used instead of a call to `xcl:inner` to specify where calls to the original function should be placed. This convention may be desupported in future releases.

The following code template illustrates the positioning of the three kinds of advice and the lexical environment available to them.

```
(block nil
    (xcl:destructuring-bind (il:!value &rest il:!other-values)
                            (multiple-value-list
                                (block nil
                                    before-advice
                                    around-advice-and-original-call) )
        after-advice
        (apply #'values il:!value il:!other-values)))
```

Note that, in addition to the lexical entities shown here, some representation of the arguments to the function is available as well, depending upon the kind of function. See the general information on Wrappers, above, for complete details.

The variables `il:!value` and `il:!other-values` are used for backward compatibility with Interlisp-D. However, the code above does not properly handle original functions that return no values at all. Advising such functions in the Lyric release will change their behavior, causing them to return a single value, `nil`. In the next release, the mechanism will be changed slightly so that a single variable, `values`, will hold a list of all of the values returned by the function. At that time, `:after` advice using the variables `il:!value` and `il:!other-values` may have to be changed. A particularly common example of functions that return no values at all is reader-macro functions. Beware of advising such functions in the Lyric release.

`:priority` can be one of `:first` or `:last`, meaning that the given piece of advice should be placed at the beginning or end, respectively, of the list of pieces of advice with the same `:when` attribute.

For backward compatibility with Interlisp-D, a list in the form (`il:before` . *coms*) or (`il:after` . *coms*) is also acceptable as the `:priority` attribute. In this case, *coms* should be a list of commands to the Interlisp-D list-structure editor. These commands will be applied to the list of pieces of advice with the same `:when` attribute; when they complete, the given advice will be inserted either before or after the selected piece, as specified. This compatibility feature

may be removed in a future release, possibly to be replaced by another facility with similar functionality.

The Xerox Lisp interface to advising consists of the following three functions:

`xcl:advise-function` *fn-to-advise form* `&key :in` [*Function*]
`(:when :before)`
`(:priority :last)`

Advises the designated *fn-to-advise* using the given *form* as the advice-expression and the given `:when` and `:priority` attributes. The `:in` argument may be used as specified in the general Wrappers description above. If the designated *fn-to-advise* is already advised, the new advice is added that already existing and the new, merged advice is applied to the original function.

`xcl:unadvise-function` *fn-to-unadvise* `&key :in :no-error` [*Function*]

Removes the advice from the designated *fn-to-unadvise*. The `:in` argument may be used as specified in the general Wrappers description above. If the designated function is not advised, an error message is printed, unless the `:no-error` argument is supplied and non-`nil`.

`xcl:readvise-function` *fn-to-readvise* `&key :in` [*Function*]

Advises the designated *fn-to-readvise* using the advice that was present the last time the function was unadvised. The `:in` argument may be used as specified in the general Wrappers description above.

The following three functions comprise the original Interlisp-D interface to the advising facility. They are provided in the Lyric release for backward compatibility. Existing user programs employing the advising facility should be changed to use the new functions, described above. The old interface may be eliminated in a future release.

`il:advise` *who when where what* [*Function*]

Advises the function named by *who*, using *what* as the advice-expression. The `:when` attribute is taken from the *when* argument and the `:priority` argument.

The *when* and *where* arguments are optional. If only two arguments are given to `il:advise`, they are interpreted as *who* and *what*, respectively. If three are given, they are *who*, *when*, and *what*.

If *when* is not supplied, `:before` is used. The *when* argument `il:before` is treated as a synonym for `:before`, as `il:after` is for `:after` and `il:around` for `:around`.

If *where* is not supplied, `:last` is used. The *where* arguments `il:last`, `il:bottom` and `il:end` are treated as synonyms for `:last`, as `il:first` and `il:top` are for `:first`.

If *who* is a symbol, this is equivalent to

```
(xcl:advise-function who what
                     :when when
                     :priority
 where)
```

If *who* is a list in the form (*fn1* **in** *fn2*), this is equivalent to

```
(xcl:advise-function fn1 what
                     :in fn2
                     :when when
                     :priority
 where)
```

Otherwise, *who* should be a list of symbols and/or sublists in the form (*fn1* **in** *fn2*). Each of the elements of the list is treated in turn, as shown above.

`il:unadvise` *fns* [*NLambda No-Spread Function*]

For each argument given, if it is a symbol this is equivalent to

```
(xcl:unadvise-function arg)
```

If the argument is a list in the form (*fn1* **in** *fn2*), this is equivalent to

```
(xcl:unadvise-function fn1 :in fn2)
```

If no arguments are given, then all currently-advised functions are unadvised. If a single argument of `t` is given, the most-recently advised function is unadvised.

`il:readvise` *fns* [*NLambda No-Spread Function*]

For each argument given, if it is a symbol this is equivalent to

```
(xcl:readvise-function arg)
```

If the argument is a list in the form (*fn1* **in** *fn2*), this is equivalent to

```
(xcl:readvise-function fn1 :in fn2)
```

If no arguments are given, then all functions that have ever been unadvised are readvised. If a single argument of `t` is given, the most-recently unadvised function is readvised.

Alone of the three aspects of the Wrappers facility, advising has some interaction with the File Manager. It is possible to save advice on a file and to optionally arrange for that advice to be re-applied when the file is loaded. The File Manager notices every time a function is advised or readvised and two File Manager commands exist for the saving of that advice:

`il:advice` {*advice-name*}* [*File Manager command*]
`il:advise` {*advice-name*}* [*File Manager command*]

advice-name should be either a symbol or a list in the form (*fn1* **:in** *fn2*), where *fn1* and *fn2* are symbols. The advice on the indicated function is saved on the file. The `il:advice` command only arranges for the advice to be stored away when the file is loaded. The `il:advise` command additionally arranges for that advice to be installed on the indicated functions. If the `il:advice` command is used, the user can call the function `xcl:readvise-function` to install the stored away advice.

For backward compatibility with Interlisp-D, an *advice-name* that is an INTERLISP symbol in the form *fn1*-**in**-*fn2* is interpreted as if it were (*fn1* **:in** *fn2*), with *fn1* and *fn2* interned in the INTERLISP package. This compatibility feature may be removed in a future release.

Stepping

Single stepping is a way of observing the evaluation of a form. At each point where the `eval` function is called execution is halted and the form about to be evaluated, along with any arguments, is printed.

When a computation is completed the result is also printed. In a sense, single stepping is like performing a stop and go trace of all the functions in an evaluation.

Only interpreted code can be stepped.

To Begin Stepping

`step` *form* [*Macro*]

Single-steps the execution of form. Returns the result of executing *form*.

What's Displayed

Each step has an indentation level, initially 0, which increases every time `eval` is called in a subform. At the start of each step, the form is printed at the current indentation level, then a space and the prompt ": ". When a subform's evaluation is completed its result is printed at the start of a new line. Variable and constant evaluation is shown by printing the variable or constant, an equal sign, and its value.

The Next Step

When execution is halted and the ":" prompt reappears, you have several options for the next step. Display a list of these options by pressing the ? key. They are:

<space>	evaluate until `eval` is called again.
Next	evaluate the current form without stepping its subforms, halt on the next form after this one
Finish	finish evaluation without stepping any more subforms.
Debugger	enter the debugger.
↑	abort all stepping, returning to top level.

Xerox Common Lisp Debugger

In Xerox Common Lisp, errors, interrupts and breakpoints wind up calling the Debugger. The debugger is an interactive Exec which can run under a Lisp computation, and display useful information about the state of the computation. This allows the user to interrogate the state of the world and affect the course of the computation.

When the debugger is entered, a separate "debugger window" is brought up. All interaction within the debugger occurs inside this separate debugger window. The default prompt for debugger windows is the character ":". Input to the debugger is evaluated within the dynamic context where the error occurred.

Note: In the Lyric release, forms typed to the debugger do not have access to the lexical context where the error occurred, even in interpreted code. This lack will be addressed in a future release.

In addition, the debugger recognizes a number of useful commands in addition to the normal Exec commands. These provide an easy way to interrogate the state of the computation.

The debugger may be entered in several different ways. Some interrupt characters invoke the debugger when they are typed. Errors may invoke the debugger. Finally, the user can add breakpoints, either around entire functions (using `xcl:break-function`) or around individual expressions.

Within the debugger the user has access to all the power of Lisp; any operations available at the Exec are also available within a debugger window, including all of the Exec commands, e.g., to redo or undo previously executed events.

Once in the debugger, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction. That is, the debugger will itself catch aborts, errors and the like. The debugger catches the `il:error` (CONTROL-E) interrupt but does not "turn off" the `il:reset` interrupt, so a CONTROL-D interrupt character will force an immediate return back to the top level.

When the debugger is invoked, a new window is brought up. The title of the debugger window indicates the type of condition that invoked the debugger. If the debugger is invoked under another call to the debugger, a new window is created. The initial placement of the debugger is relative to the placement of the typescript window of the process being invoked.

Note: Storage errors (running out of storage) will not try to open a new window, since this might cause the error to occur repeatedly.

Note: The debugger can also operate in a mode where a new window is not created. If the variable `il:wbreak` is `nil`, debugging interactions occur within the same window as the primary typescript window.

Debugger commands can be invoked either by typing them in, or, for those debugger commands in `xcl:*debugger-menu-items*`, by invoking them directly from the middle-button pop-up menu in a debugger window. The operation of interactive commands differs depending on how they are invoked: for those invoked by typing the command, the interaction happens in the typescript window, while those invoked by mouse action cause a mouse/menu interaction instead.

`eval` [*Debugger command*]

For breakpoints, this command evaluates the breakpointed function/expression, and prints the values it returns. A subsequent `value` command will (re)print these values. A subsequent `ok` command will merely return the values already computed. However, a subsequent `eval` command will perform the computation again.

For error calls to the debugger, this command attempts to back up the stack to the last "user" function and reapply it to its arguments, presuming that somehow the user has modified the computation. If successful, this value will be returned by a subsequent `ok` command from the user function. The "user" function is determined by looking back on the stack to the last stack frame which is not part of the interpreter.

`ub` [*Debugger command*]

Removes the current breakpoint, if there is one.

Not available from the menu.

`value` [*Debugger command*]

Prints the result of the last `eval` command executed in this debugger instance. If no `eval` has been done yet, simply prints "Not yet evaluated. "

↑ , stop [*Debugger commands*]

Abort the Debugger, making it "go away" without returning a value. This is a useful way to unwind to a higher level debugger or Exec.

return &optional *expression* [*Debugger command*]

expression is evaluated, and returned as the value the debugger call. For example, one could use the eval command and follow this with return (reverse value).

Not available from the menu.

proceed, pr [*Debugger command*]

It constructs a list of currently enabled proceed cases, then prompts the user to select one to invoke. If this command is invoked from the debugger exec, il:askuser is used to select a proceed case. If this command was invoked from the debugger's menu, a menu of proceed cases to select from is presented. In either case, the proceed cases will be described by the results of invoking their report methods.

For example, if you evaluated

```
(xcl:proceed-case (break)
   (xcl:use-value (x)
         :report "Provide a value to use as the result"
      x)
   (nil ()
         :report "Just return NIL"
      nil))
```

and then executed the **PR** command in the debugger, you would see:

```
1 - Return from BREAK
2 - Provide a value to use as the result
3 - Just return NIL
4 - Unwind to ERRORSET
No - don't proceed
Proceed how?
```

Selecting No will abort the command.

ok [*Debugger command*]

If the debugger was entered through a breakpoint, the debugger first executes an eval if the user has not done so already. These values are then returned as the values of the breakpointed function/expression.

If the debugger was entered through an error, the ok command first calls the function `xcl:proceed`. For many errors, there is a proceed case by that name enabled that will reapply the last "user"function (before the error) to its arguments. If this call to `xcl:proceed` returns, this means that there was no proceed case with the name `xcl:proceed` enabled, so the debugger will ask the user to select a proceed case to invoke, just as the `pr` command would.

`pb` *variable* [*Exec command*]

Prints the bindings of the special variable *variable*. This command is available in top-level Execs as well as in the Debugger, but is most useful in the Debugger.

`il:lastpos` [*Variable*]

Some debugger commands manipulate the stack. The special variable `il:lastpos` contains a stack pointer to the "focus" for stack commands. When the debugger is entered, `il:lastpos` is bound to a stack pointer to the user frame before whatever called the the debugger, e.g., the frame before the call to `error`, `il:errorx`, etc.

`?=` [*Debugger command*]

This command is used is to interrogate the values of the arguments at `il:lastpos`. For example, if `foo` has three arguments (`x y z`), then typing `?=` when at `foo` will produce:

```
12:?=
X = value of X
Y = value of Y
Z = value of Z
13:
```

`?=` is a universal mnemonic for displaying argument names and their corresponding values. Additional frame information can be obtained using the debugger frame menu, but a typed `?=` is often a quick way of getting information.

`il:breakdelimiter` [*Variable*]

For output to the typescript window, the value of `il:breakdelimiter`, initially a string with a new-line character in it, is printed to delimit the

output of ?= and backtrace commands. Resetting it to "," would produce more compact output.

`bt` [*Debugger command*]

Shows a backtrace of "interesting" function names starting at `il:lastpos`. Whether or not a function is interesting is determined by the predicate in the variable `il:*short-backtrace-filter*`.

When invoked from the debugger menu with the mouse, causes a menu of frames to be attached to the Debugger window.

`il:*short-backtrace-filter*` [*Variable*]

Contains a predicate used by the `bt` command to determine if a frame is interesting. The initial definition of "interesting" is that a frame is interesting if it corresponds to a "user" function, currently defined as any symbol whose name does not begin with the character backslash (\).

`dbt` [*Debugger command*]

Same as invoking `bt` from the debugger menu, i.e., causes a menu of frames that `bt` would have listed to be attached to the debugger window.

`bt!, dbt!` [*Debugger command*]

Like `bt` and `dbt`, respectively, but show all frames, not just interesting ones.

`btv` [*Debugger command*]

Shows not only all frames, but also all special bindings on the stack, beginning at `il:lastpos`.

Not available from the menu.

`btv!` [*Debugger command*]

Prints *everything* on the stack, including binary stack locations, etc. Normally for system debugging only.

`@ &rest` *frame-specification* *[Debugger command]*

Resets the variable `il:lastpos`, according to *frame-specification*; the position is set first according to the initial debugger entry position, and then, for each element in frame-specification, the frame is changed.

Note that `@` on a line by itself resets `il:lastpos` to its initial value. Normally, symbols witin a `@` command refer to the next frame with the given symbol as the frame-name, e.g., "`@ +`" sets `il:lastpos` to a pointer to the last + frame. The following tokens are looked for (using `string-equal`, i.e., package does not matter) and treated specially:

`@` This effectively means to leave `il:lastpos` alone, i.e., not reset it before processing the rest of the line.

a number *n* Move `il:lastpos` down the stack *n* frames back. E.g., "`@ 3`" means 3 frames before the initial call, and "`@ @ 3`" means 3 *more* frames.

`/` The next element on the line (which should be a positive integer) specifies that the *previous* symbol should be searched for that many times. For example, "`@ foo / 3`" is equivalent to "`@ foo foo foo`."

`=` Resets `il:lastpos` to the *value* of the next expression, e.g., if the value of `foo` is a stack pointer, "`@ = foo fie`" will search for `fie` in the environment specified by (the value of) `foo`.

For example, if the stack looks like:

[9] `debugger`
[8] `foo`
[7] `cond`
[6] `fie`
[5] `cond`
[4] `fie`
[3] `cond`
[2] `fie`
[1] `fum`

then "`@ fie cond`" will set `il:lastpos` to the position corresponding to *[5]*; "`@ @ cond`" will then set `il:lastpos` to *[3]*; and "`@ fie / 3 1`" to *[1]*.

If the search is still unsuccessful, `@` aborts. When `@` finishes, it returns the name of the frame at `il:lastpos`, i.e., (`il:stkname il:lastpos`).

Note: `il:lastpos` is also reset by selecting a frame in an attached backtrace menu.

`revert &rest` *frame-specification* [*Debugger command*]

Goes back to a stack frame and reenters the function called at that point with the arguments found on the stack.

If no argument is given to `revert`, it reverts to the frame selected by `il:lastpos`. Otherwise, the `revert` command processes the rest of the line similarly to the `@` command, e.g., `"revert foo 1"` is equivalent to `"@ foo 1"` followed by `revert`.

`revert` is useful for restarting a computation in the situation where a bug is discovered at some point *below* where the problem actually occurred. `revert` essentially says "go back there and start over."

Controlling When to Enter the Debugger

For simple errors which occur as the result of user type-in, it is sometimes more convenient to merely use the `fix` command to correct the input or to retype it than to enter the debugger, proceed, or the like. Thus, in Xerox Common Lisp, the error system employs a simple heuristic in the function `xcl:enter-debugger-p`. The actual algorithm is described in detail below; however, the parameters affecting the decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

`xcl:enter-debugger-p` *pos condition* [*Function*]

`xcl:enter-debugger-p` is called by the error routines to decide whether or not to actually enter the debugger when an error occurs. *pos* is the stack position at which the error occurred; *condition* is the error condition. Returns `t` if the debugger should occur; `nil` if the computation should simply abort.

`il:helpflag` [*Variable*]

If `il:helpflag` is `nil`, `xcl:enter-debugger-p` will return `nil`. If `il:helpflag` is `break!`, then `xcl:enter-debugger-p` will return `t`. Otherwise, `xcl:enter-debugger-p` will look at the stack depth, using `il:helpdepth`.

`il:helpdepth` [*Variable*]

If more than `il:helpdepth` "interesting" function frames occur between the error call and the type-in form that eventually caused it, `xcl:enter-debugger-p` will return `t`. Otherwise, `xcl:enter-debugger-p` will look at the amount of time elapsed since execution was started for the expression that invoked this exec.

`il:helpclock` [*Variable*]

At each Exec command (including inside the debugger) the variable `il:helpclock` is rebound to the current value of `(get-internal-real-time)`.

`il:helptime` [*Variable*]

If more than `il:helptime` milliseconds of runtime since `il:helpclock` have elapsed, then `xcl:enter-debugger-p` will be true. The time criterion for breaking can be suppressed by setting `il:helptime` to `nil`.

Interface to the Debugger

`il:|MaxBkMenuWidth|` [*Variable*]

`il:|MaxBkMenuHeight|` [*Variable*]

The variables `il:|MaxBkMenuWidth|` (default 125) and `il:|MaxBkMenuHeight|` (default 300) control the maximum size of the backtrace menu. If this menu is too small to contain all of the frames in the backtrace, it is made scrollable in both vertical and horizontal directions.

`il:autobacktraceflg` [*Variable*]

This variable controls when and what kind of backtrace menu is automatically brought up. The value of `il:autobacktraceflg` can be one of the following:

`nil` The backtrace menu is not automatically brought up (the default).

`t` On error breaks the `bt` menu is brought up.

`il:bt!` On error breaks the `bt!` menu is brought up.

loop in which only eval expressions can be typed. If the recursive debugger entry was because of a breakpoint, the second debugger invocation is ignored.

25.4. Environmental Inquiries

25.4.1 Time Functions and Commands

In addition to the time functions explained in *Common Lisp: the Language*, Xerox Common Lisp provides the following function:

`time` *form* `&key :repeat :output :datatypes` [*Function*]

Reports the time required to evaluate *form*, and returns the value of *form*. By default, timing information is sent to `*trace-output*`, which is usually a window with the same title. If the `:repeat` argument is provided, it should be an integer which indicates the number of times to repeat the evaluation of form. If the `:output` argument is provided, it should be a valid stream argument (like `*terminal-io*`). If `:datatypes` is provided, it should be a list of data type names; `time` will then only report storage allocations for the listed datatypes.

`time` does not use global state, so it may be nested, etc.

There's also an Exec command:

`time` *expression* `&key :repeat :datatypes` [*Command*]

The `time` command sends its output to `*terminal-io*` by default.

`room &optional` *x* [*Function*]

The function `room` is implemented as `(il:storage)`; the optional argument is currently ignored. Documentation for `il:storage` may be found in the Lisp Library module GCHAX.