VOLUME II—ENVIRONMENT

Chapter 13 Interlisp Executive

[This chapter of the *Interlisp-D Reference Manual* has been renamed Chapter 13, Executives.]

Lisp has a new kind of Executive (or Exec), designed for use in an environment with both Interlisp and Common Lisp. This executive is available in three standard modes, distinguished by their default settings for package and readtable:

XCL	New Exec. Uses XCL readtable, XCL-USER package
CL	New Exec. Uses LISP readtable, USER package
IL	New Exec. Uses INTERLISP readtable, INTERLISP package

In addition, the old Interlisp executive, the "Programmer's Assistant", is still available in this release for the convenience of Koto users:

OLD-INTERLISP	Old "Programmer's Assistant" Exec. Uses OLD-INTERLISP-T readtable, INTERLISP package. It is likely that this executive will not be supported in future releases.

When Lisp starts, it is running a single executive, the XCL Exec. You can spawn additional executives by selecting EXEC from the background menu. The type of an executive is indicated in the title of its window; e.g., the initial executive has title "Exec (XCL)". Each executive runs in its own process; when you are finished with an executive, you can simply close its window, and the process is killed.

The new executive is modeled, somewhat, on the old "Programmer's Assistant" executive and, to a first approximation, you can type to it just as you did in past releases. You should note, however, that the default executive (XCL) expects Common Lisp input syntax, and reads symbols relative to the XCL-USER package. This means that to type Interlisp symbols, you must prefix the symbol with the characters "IL:" (in upper or lower case). And even in the new IL executive, the readtable being used is the new INTERLISP readtable, in which the characters colon (:), vertical bar (|) and hash (#) all have different meanings than in Koto.

The OLD-INTERLISP exec, with one exception, uses exactly the same input syntax as in Koto; this means in particular that colon cannot be used to type package-qualfied symbols, since colon is an ordinary character there. The one exception is that there *is* a package delimiter character in the OLD-INTERLISP readtable, should you have a need to use it—Control-↑, which usually echoes as "↑↑", though it may appear as a black rectangle in some fonts.

The new executive does differ from the old one in several respects, especially in terms of its programmatic interface. Complete details of the new executive can be found in Appendix A. The Exec. Some of the important differences are:

- Executives are numbered

Executives, other than the first one, are labeled with a distinct number. This number appears in the exec window's title, and also in its prompt, next to the event number. The OLD-INTERLISP executive does not include this exec number.

- Event number allocation

The numbers for events are allocated at the time the prompt for the event is printed, but all execs still share a common event number space and history list. This means that ?? shows all events that have occurred in *any* executive, though not necessarily in the order in which the events actually occurred (since it is the order in which the event numbers were allocated). Events for which the type-in has not been completed are labeled "<in progress>" in the ?? listing. In the old executive, event numbers are not allocated until type-in is complete, which means that the number printed next to the prompt is not necessarily the number associated with the event, in the case that there has been activity in other executives.

In the new executive, relative event specifications are local to the exec; e.g., **-1** refers to the most recent event in that specific exec. In the old executive, **-1** referred to the immediately preceding event in *any* executive.

- New facility for commands

The old Executive has commands based on **LISPXMACROS**. The new Executive has its own command facility, **XCL:DEFCOMMAND**, which allows commands to be named without regard to package, and to be written with familiar Common Lisp style of argument list.

- Commands are typed *without* parentheses

In the old executive, a command could be typed with or without enclosing parentheses. In the new executive, a parenthesized form is always interpreted as an EVAL-style input, never a command.

- **SETQ** does not interact with the File Manager

In the Koto release, when you typed in the Exec

(**SETQ FOO** *some-new-value-for-FOO*)

the executive responded **(FOO reset)**, and the file package was told that **FOO**'s value changed. Any files on which **FOO** appeared as a variable would then be marked as needing to be cleaned up. If **FOO** appeared on no file, you'd be prompted to put it on one when you ran (**FILES?**).

This is still the case in the old executive. However, it is no longer the case in the new executive. If you are setting a variable that is significant to a program and you want to save it on a file, you should use the Common Lisp macro **CL:DEFPARAMETER** instead of **SETQ**. This will give the symbol a definition of type **VARIABLES** (rather than **VARS**), and it will be noticed by the File manager. If you want to change the value of the variable, you must either use

CL:DEFPARAMETER again, or edit the variable using **ED** (not **DV**).

- Programmatic interface completely different

As a first approximation, all the functions and variables in *IRM* Sections 13.3 (except the **LISPXPRINT** family) and 13.6 apply only to the Old Interlisp Executive, unless specified otherwise in Appendix A. In particular, the variables **PROMPT#FLG**, **PROMTPCHARFORMS**, **SYSPRETTYFLG**, **HISTORYSAVEFORMS**, **RESETFORMS**, **ARCHIVEFN**, **ARCHIVEFLG**, **LISPXUSERFN**, **LISPXMACROS**, **LISPXHISTORYMACROS** and **READBUF** are not used by the new Exec. The function **USEREXEC** invokes an old-style Executive, but uses the package and readtable of its caller. The function **LISPXUNREAD** has no effect on the new Exec. Callers of **LISPXEVAL** are encouraged to use **EXEC-EVAL** instead.

Some subsystems still use the old-style Executive—in particular, the tty structure editor.

Chapter 14 Errors and Breaks

Lisp extends the Interlisp break package to support multiple values and the Common Lisp lambda syntax. Interlisp errors have been converted to Common Lisp conditions.

Note that Sections 14.2 through 14.6 in the *Interlisp-D Reference Manual* have been replaced by new Debugger information (see *Common Lisp Implementation Notes)*.

Section 14.3 Break Commands

(II:14.6)

The **!EVAL** debugger command no longer exists.

(II:14.10-11)

The Break Commands = and -> are no longer supported.

Section 14.6 Creating Breaks with BREAK1

While the function **BREAK1** still exists, broken and traced functions are no longer redefined in terms of it. More primitive constructs are used.

Section 14.7 Signalling Errors

Interlisp errors now use the new XCL error system. Most of the functions still exist for compatibility with existing Interlisp code, but the underlying machinery is different. There are some incompatible differences, however, especially with respect to error numbers.

The old Interlisp error system had a set of registered error numbers for well known error conditions, and all other errors were identified by a string (an error message). In the new Lisp error system, all errors are handled by signalling an object of type **XCL:CONDITION**. The mapping from Interlisp error numbers to Lisp conditions is given below in Section 14.10.

Since one cannot in general map a condition object to an Interlisp error number, the function **ERRORN** no longer exists. The equivalent functionality exists by examining the special variable ***LAST-CONDITION***, whose value is the condition object most recently signaled.

(ERRORX ERXM) calls **CL:ERROR** after first converting **ERXM** into a condition in the following way: If **ERXM** is **NIL**, the value of ***LAST-CONDITION*** is used; if **ERXM** is an Interlisp error descriptor, it is first converted to a condition; finally, if **ERXM** is already a condition, it is passed along unchanged. **ERRORX** also sets up a proceed case for **XCL:PROCEED**, which will attempt to re-evaluate the caller of **ERRORX**, much as OK did in the old Interlisp break package.

ERROR, **HELP**, **SHOULDNT**, **RESET**, **ERRORMESS**, **ERRORMESS1**, and **ERRORSTRING** work as before. All output is directed to ***ERROR-OUTPUT***, initially the terminal.

ERROR! is equivalent to the new error system's **XCL:ABORT** proceed function, except that if no **ERRORSET** or **XCL:CATCH-ABORT** is found, it unwinds all the way to the top of the process.

SETERRORN converts its arguments into a condition, then sets the value of ***LAST-CONDITION*** to the result.

Section 14.8 Catching Errors

ERRORSET, **ERSETQ** and **NLSETQ** have been reimplemented in terms of the new error system , but their behavior is essentially the same as before. **NLSETQ** catches all errors (conditions of type **CL:ERROR** and its descendants), and sets up a proceed case for **XCL:ABORT** so that **ERROR!** will return from it. **ERSETQ** also sets up a proceed case for **XCL:ABORT**, though it does not catch errors.

One consequence of the new implementation is that there are no longer frames named **ERRORSET** on the stack; programs that explicitly searched for such frames will have to be changed.

ERRORTYPELIST is no longer supported. The equivalent functionality is provided by default handlers. Although condition handlers provide a more powerful mechanism for programmatically responding to an error condition, old **ERRORTYPELST** entries generally cannot be translated directly. Condition handlers that want to resume a computation (rather than, say, abort from a well-know stack location) generally require the cooperation of a proceed case in the signalling code; there is no easy way to provide a substitute value for the "culprit" to be re-evaluated in a general way.

One important difference between **ERRORTYPELIST** and condition handlers is their behavior with respect to **NLSETQ**. In Koto, the relevant error handler on **ERRORTYPELST** would be tried, even for errors occurring underneath an **NLSETQ**. In Lyric, the **NLSETQ** traps all errors before the default condition handlers can see the error. This means, for example, that the behavior of **(NLSETQ (OPENSTREAM --))** is now different if the **OPENSTREAM** causes a file not found error—in Koto, the system would search **DIRECTORIES** for the file; in Lyric, the **NLSETQ** returns **NIL** immediately without searching, since the default handler for **XCL:FILE-NOT-FOUND** is not invoked.

Section 14.9 Changing and Restoring System State

The special forms **RESETLST**, **RESETSAVE**, **RESETVAR**, **RESETVARS** and **RESETFORM** still exist, but are implemented by a new mechanism that also supports Common Lisp's **CL:UNWIND-PROTECT**. Common Lisp's **CL:THROW** and (in most cases) Interlisp's **RETFROM** and related control transfer constructs cause the cleanup forms of both **CL:UNWIND-PROTECT** and **RESETLST** (etc) to be performed. This is discussed in more detail in the notes for Chapter 11, the stack.

Section 14.10 Error List

Most of the Interlisp errors are mapped into condition types in Lisp. Some are no longer supported. Following is the list of error type mappings. The first name is the condition type that the error descriptor turns into. If there is a second name, it is the slot whose value is set to **CADR** of the error descriptor. Any additional pairs of items are the values of other slots set by the mapping. Attempting to use an unsupported error type number will result in a simple error to that effect.

0	Obsolete
1	Obsolete
2	**STACK-OVERFLOW**
3	**ILLEGAL-RETURN**
4	**XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'LIST**
5	**XCL:SIMPLE-DEVICE-ERROR** *MESSAGE*
6	**XCL:ATTEMPT-TO-CHANGE-CONSTANT**
7	**XCL:ATTEMPT-TO-RPLAC-NIL** *MESSAGE*
8	**ILLEGAL-GO TAG**
9	**XCL:FILE-WONT-OPEN PATHNAME**
10	**XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:NUMBER**
11	**XCL:SYMBOL-NAME-TOO-LONG**
12	**XCL:SYMBOL-HT-FULL**
13	**XCL:STREAM-NOT-OPEN** *STREAM*
14	**XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:SYMBOL**
15	Obsolete
16	**END-OF-FILE** *STREAM*
17	**INTERLISP-ERROR** *MESSAGE*
18	Not supported (control-B interrupt)

19	**ILLEGAL-STACK-ARG** *ARG*
20	Obsolete
21	**XCL:ARRAY-SPACE-FULL**
22	**XCL:FS-RESOURCES-EXCEEDED**
23	**XCL:FILE-NOT-FOUND** *PATHNAME*
24	Obsolete
25	**INVALID-ARGUMENT-LIST** *ARGUMENT*
26	**XCL:HASH-TABLE-FULL** *TABLE*
27	**INVALID-ARGUMENT-LIST** *ARGUMENT*
28	**XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE ’ARRAYP**
29	Obsolete
30	**STACK-POINTER-RELEASED** *NAME*
31	**XCL:STORAGE-EXHAUSTED**
32	Not supported (attempt to use item of incorrect type)
33	Not supported (illegal data type number)
34	**XCL:DATA-TYPES-EXHAUSTED**
35	**XCL:ATTEMPT-TO-CHANGE-CONSTANT**
36	Obsolete
37	Obsolete
38	**XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE ’READTABLEP**
39	**XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE ’TERMTABLEP**
40	Obsolete
41	**XCL:FS-PROTECTION-VIOLATION**
42	**XCL:INVALID-PATHNAME** *PATHNAME*
43	Not supported (user break)
44	**UNBOUND-VARIABLE** *NAME*
45	**UNDEFINED-CAR-OF-FORM** *FUNCTION*
46	**UNDEFINED-FUNCTION-IN-APPLY**
47	**XCL:CONTROL-E-INTERRUPT**
48	**XCL:FLOATING-UNDERFLOW**
49	**XCL:FLOATING-OVERFLOW**
50	Not supported (integer overflow)
51	**XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE ’CL:HASH-TABLE**
52	**TOO-MANY-ARGUMENTS** *CALLEE* **:MAXIMUM CL:CALL-ARGUMENTS-LIMIT**

Note that there are many other condition types in Lisp; see the error system documentation in the *Common Lisp Implementation Notes* for details.

Chapter 15 Breaking Functions and Debugging

In Lyric the uses of BREAK, TRACE, and ADVISE are unchanged, from the user's point of view, but the internals of their implementation are quite different.

For complete documentation on the new implementation of breaking, tracing and advising, see the *Common Lisp Implementation Notes*, Section 25.3.

In particular, you should note the following differences:

- The variable **BRKINFOLST** no longer exists and the format of the value of the variable **BROKENFNS** has changed. In addition, the **BRKINFO** property is no longer used.
- **BREAK** and **TRACE** no longer work on CLISP words.
- The **BREAKIN** and **UNBREAKIN** functions no longer exist. No comparable facility exists in Lisp. The user can manually insert calls to the Common Lisp function **CL:BREAK** in order to create a breakpoint at that point in the function.

Please note the following additional changes to breaking functions:

Section 15.1 Breaking Functions and Debugging

(**BREAK0** *FN WHEN COMS* — —) [Function]

The function **BREAK0** now works when applied to an undefined function. This allows you to use the breaking facility to create "stubs" that generate a breakpoint when called. You can then examine the arguments passed and use the **RETURN** command in the debugger to return the proper result(s).

The "break commands" facility (the *COMS* argument) is no longer supported. **BREAK0** now signals an error when supplied with a non-**NIL** third argument. If you need finer control over the functioning of breakpoints you are directed to the **ADVISE** facility; it offers complete control of how and when the given function is evaluated.

Passing a non-atomic argument in the form **(***FN1* **IN** *FN2***)** as the first argument to **BREAK0** still has the effect of creating a breakpoint wherever *FN2* calls *FN1*. However, it no longer creates a function named *FN1***-IN-***FN2* to do so. In addition, the format of the value of the **NAMESCHANGED** property has changed and the **ALIAS** property is no longer used.

(**TRACE** *X*) [Function]

TRACE is no longer a special case of **BREAK**, though the functions **UNBREAK** and **REBREAK** continue to work on traced functions.

In addition, the function **TRACE** no longer calls **BREAK0** in order to do its job. Also, non-atomic arguments to **TRACE** no longer specify forms the user wishes to see in the tracing output.

(**UNBREAK** *X*) [Function]

The function **UNBREAK** is no longer implemented in terms of **UNBREAK0**, although that function continues to exist.

Section 15.2 Advising

The implementation of advising has been completely reworked. While the semantics implied by the code shown in Section 15.2.1 of the *Interlisp-D Reference Manual* is still supported, the details are quite different. In particular, it is now possible to advise functions that return multiple values and for **AFTER**-style advice to access those values. Also, all advice is now compiled, rather than interpreted. The advising facility no longer makes use of the special forms **ADV-PROG**, **ADV-RETURN**, and **ADV-SETQ**.

You should also note the following changes to the advise facility:

- The editing of advice has changed slightly. In previous releases, the advice and original function-body were edited simultaneously. In Lyric, they can only be edited separately. When you finish editing the advice for a function, that function is automatically re-advised using the new advice.
- The variable **ADVINFOLST** no longer exists and the format of the value of the variable **ADVISEDFNS** has changed. In addition, the properties **ADVICE** and **READVICE** are no longer used, except in the handling of advice saved on files from previous releases. Advice saved in Lyric does not use the **READVICE** property.
- The function **ADVISEDUMP** no longer exists.
- Advice saved on files in previous releases can, in general, be loaded into the Lyric system compatibly. A known exception is the case in which a list of the form **(***FN1* **IN** *FN2***)** was given to the **ADVICE** or **ADVISE** file package commands. When **READVISE** is called on such a name, the old-style advice, on the **READVICE** property of the symbol *FN1***-IN-***FN2*, will not be found. This will eventually lead to an **XCL:ATTEMPT-TO-RPLAC-NIL** error. The user should evaluate the form
  (RETFROM ’READVISE1)
  in the debugger to proceed from the error and later evaluate
  (READVISE *FN1***-IN-***FN2***)**
  by hand to install the advice.

- The **ADVICE** and **ADVISE** File Manager commands now accept three kinds of arguments:
  a symbol, naming an advised function,
  a list in the form **(***FN1* **:IN** *FN2***)**, and
  a symbol of the form *FN1***-IN-***FN2*.
  Arguments of the form **(***FN1* **IN** *FN2***)** are not acceptable any longer. Arguments of the form *FN1***-IN-***FN2* should be converted into the equivalent form **(***FN1* **:IN** *FN2***)**.

(**ADVISE** *WHO WHEN WHERE WHAT*) [Function]

In the Lyric release of Lisp, **ADVISE** has some changes in the way arguments are treated and the possible values for those arguments. Most notably:

- In earlier releases, you could call **ADVISE** with only one argument, the name of a function. In this case, **ADVISE** "set up" the named function for advising, but installed no advice. This usage is no longer supported.
- Previously, an undocumented value of **BIND** was accepted for the *WHEN* argument to **ADVISE**. This kind of advice is no longer supported. It can be adequately simulated using **AROUND** advice.

In addition, advising Common Lisp functions works somewhat differently with respect to a function’s arguments. The arguments are not available by name. Instead, the variable **XCL:ARGLIST** is bound to a list of the values passed to the function and may be changed to affect what will be passed on.

As with the breaking facility (see above), **ADVISE** no longer creates a function named *FN1***-IN-***FN2* as a part of advising **(***FN1* **IN** *FN2***)**.

Chapter 16 List Structure Editor

The list structure editor, DEdit, is not part of the Lisp environment. It is now a Lisp Library Module. Chapter 16 has been renamed Structure Editor.

SEdit, the new Lisp editor, replaced DEdit in the Lyric release. The description of SEdit may be found in Appendix B of this volume. The commands used to invoke both SEdit and DEdit are the same.

Following is a description of the interface to the Lisp editor.

Switching Between Editors

If you have both SEdit and DEdit loaded, you can switch between them by calling: (**EDITMODE** *'EDITORNAME*) where *EDITORNAME* is one of the symbols SEdit or DEdit.

Packages

The **ED** editor interface accepts TYPE information from the Interlisp or Common Lisp packages.

Starting a Lisp Editor

In the XCL environment, calling ED with a pathname will start the editor on the coms of the file (as if DC had been called).

(ED *NAME &OPTIONAL OPTIONS*) [Function]

This function starts the Lisp editor. **ED** is the default interface to the editor. SEdit is the default Lisp editor. The same symbol, **ED**, is exported in both the IL and CL packages.

NAME is the name of any File Manager object.

OPTIONS is either a single symbol or a list of symbols, each of which is either a File Manager type or one or more of the keywords **:DISPLAY**, **:DONTWAIT**, **:CURRENT, :COMPILE-ON-COMPLETION, :CLOSE-ON-COMPLETION**, or **:NEW**. If exactly one File Manager type is given, **ED** tries to edit that type of definition for *NAME*. If more than one type is given in *OPTIONS*, **ED** will determine for which of them *NAME* has a definition. If a definition exists for more than one of the types, **ED** gives you a choice of which one to edit. If no File Manager types are given, **ED** treats *OPTIONS* as a list of all of the existing types; thus you are given a choice of all of the existing definitions of *NAME*.

The variable **FILEPKGTYPES** contains a complete list of the currently-known manager types.

If the keyword **:DISPLAY** is included in *OPTIONS*, **ED** uses menus for any prompting, (e.g., to choose one of several possible definitions to edit). If **:DISPLAY** is not included, **ED** prints its queries to and reads the user's replies from ***QUERY-IO*** (usually the Exec in which you are typing). Thus all of the following are correct ways to call the editor:

```
(ED 'NAME :DISPLAY)
(ED 'NAME 'FUNCTIONS)
(ED 'NAME '(:DISPLAY))
(ED 'NAME '(FUNCTIONS :DISPLAY))
(ED 'NAME '(FUNCTIONS VARIABLES :DISPLAY))
```

The other keywords are interpreted as follows:

:CURRENT

This is a new option with Medley that causes ED to call TYPESOF with SOURCE=CURRENT. This prevents TYPESOF from searching FILECOMS and from looking in WHERE-IS databases. The **CURRENT** option looks only for definitions that are currently loaded. When you know that the definition is loaded, use of the **CURRENT** option results in ED being significantly faster.

:DONTWAIT

Lets the edit interface return right away, rather than waiting for the edit to be complete. **DF**, **DV**, **DC**, and **DP** specify this option now, so editing from the exec will not cause the exec to wait.

:NEW

Lets you install a new definition for the name to be edited. You will be asked what type of dummy definition you wish to install based on which file manager types were included in *OPTIONS*.

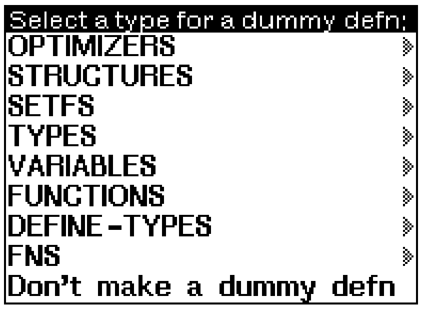
:COMPILE-ON-COMPLETION

This option specifies that the definition being edited should be compiled upon completion regardless of the completion command used.

:CLOSE-ON-COMPLETION

Tells the editor that it must close the editor window after the first completion. So in SEdit, CONTROL-X will close the window; shrinking the window is not allowed. Editor windows opened by the exec command **FIX** specify this option.

If *NAME* does not have a definition of any of the given types, **ED** can create a dummy definition of any of those types. If **:DISPLAY** is provided in *OPTIONS,* **ED** will pop-up the following menu asking you which type of definition to install. Select the template for the type of definition you wish to create from the DEFN menus and submenus:



New kinds of dummy definitions can be added to the system through the use of the **:PROTOTYPE** option to **XCL:DEFDEFINER.**

Mapping the Old Edit Interface to ED

The old functions for starting the Lisp editor (**DF**, **DV**, **DP**, and **DC**) have been reimplemented so that they work with Common Lisp. The old edit commands map to the new editor function (ED) as follows:

```
DF NAME ⇒ (ED 'NAME '(FUNCTIONS FNS :DONTWAIT))
DV NAME ⇒ (ED 'NAME '(VARIABLES VARS :DONTWAIT))
DP NAME ⇒ (ED 'NAME '(PROPERTY-LIST :DONTWAIT))
DP NAME MYPROP⇒ (ED '(NAME  MYPROP) '(PROPS :DONTWAIT))
DC NAME ⇒ (ED 'NAME '(FILES :DONTWAIT))
```

Thus, for example, when **DF** is invoked it looks first for Common Lisp `FUNCTIONS` and then for Interlisp `FNS`. **DV**, **DP** and **DC** operate in an analogous fashion.

Editing Values Directly

The TYPE you specify for the object you want to edit determines how that object is edited, i.e. by DEFINITION or VALUE. Normally you want to edit the DEFINITION (this is the default). For example, suppose *FOO* is defined as a variable; to start the editor on the DEFINITION of *FOO,* use the form:

```
(ED 'FOO)   or  (ED 'FOO 'VARIABLES)
```

There may be times when you do not have access to the DEFINITION of an object that you need to edit. This can occur when you do not have the source code loaded. You can edit its VALUE directly using the form:

```
FOR VARIABLES:  ⇒ (ED 'NAME 'IL:VARS)

FOR FUNCTIONS:  ⇒ (ED 'NAME 'IL:FNS)
```

By starting the editor on the VALUE of an object, you can change its value without changing its definition. (AR 8971)

To start the editor on the VALUE of *FOO*, for example, use the form:

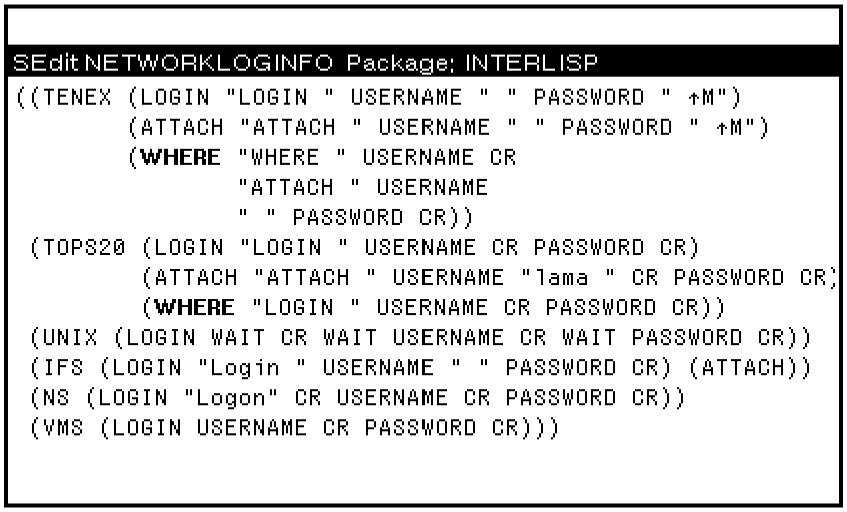
```
(ED 'FOO 'VARS)
```

EXAMPLE:

When you load a compiled file, the DEFINITION of an object is not loaded. Only the VALUE is loaded. The compiler does not store the defining forms for objects. Suppose you have compiled code for a system file loaded, but you do not have access to the sources that contain the DEFINITIONS, and you need to change the value of a system variable, say NETWORKLOGINFO. This variable has a defining form and the system knows this, but the form is not loaded and is not available. You can edit the VALUE of the variable directly using:

```
(ED 'NETWORKLOGINFO 'IL:VARS)
```

An editor window opens displaying the VALUE of NETWORKLOGINFO:



```
SEdit NETWORKLOGINFO  Package: INTERLISP
((TENEX (LOGIN "LOGIN " USERNAME " " PASSWORD " ↑M")
        (ATTACH "ATTACH " USERNAME " " PASSWORD " ↑M")
        (WHERE "WHERE " USERNAME CR
                "ATTACH " USERNAME
                " " PASSWORD CR))
 (TOPS20 (LOGIN "LOGIN " USERNAME CR PASSWORD CR)
         (ATTACH "ATTACH " USERNAME "lama " CR PASSWORD CR)
         (WHERE "LOGIN " USERNAME CR PASSWORD CR))
 (UNIX (LOGIN WAIT CR WAIT USERNAME CR WAIT PASSWORD CR))
 (IFS (LOGIN "Login " USERNAME " " PASSWORD CR) (ATTACH))
 (NS (LOGIN "Logon" CR USERNAME CR PASSWORD CR))
 (VMS (LOGIN USERNAME CR PASSWORD CR)))
```

Section 16.18 Editor Functions

(II:16.74)

The function **FINDCALLERS** has the following limitations in Lisp:

1. **FINDCALLERS** only identifies by name the occurrences inside of Interlisp FNS, not Common Lisp FUNCTIONS.

2. Because **FINDCALLERS** uses a textual search, it may report more occurrences of the specified *ATOMS* than there actually are, if the file contains symbols by the same name in another package, or symbols with the same p-name but different alphabetic case. **EDITCALLERS** still edits only the actual occurrences, since it reads the functions and operates on the real Lisp structure, not its printed representation.

Chapter 17 File Package

The Interlisp-D File Package has been renamed the File Manager. Its operation is unchanged; however, it has been extended to manipulate, load and save Common Lisp functions, variables, etc. It also allows specification of the reader environment (package and readtable) to use when writing and reading a file, solving the problem of compatibility between old and new (Common Lisp) syntax.

Note that although source files from earlier releases can be loaded into Lyric, files produced by the File Manager in the Lyric release cannot be loaded into previous releases. This is true for several reasons, the most important being that previous releases did not have packages, so symbols cannot be read back consistently.

The new File Manager includes several new types to deal with the various definition forms supported in Xerox Common Lisp. The following table associates each new type with the forms that produce definitions of that type:

FUNCTIONS	**CL:DEFUN, CL:DEFMACRO, CL:DEFINE-MODIFY-MACRO, XCL:DEFINLINE, XCL:DEFDEFINER, XCL:DEFINE-PROCEED-FUNCTION.**
VARIABLES	**CL:DEFCONSTANT, CL:DEFVAR, CL:DEFPARAMETER, XCL:DEFGLOBALVAR, XCL:DEFGLOBALPARAMETER**
STRUCTURES	**CL:DEFSTRUCT, XCL:DEFINE-CONDITION**
TYPES	**CL:DEFTYPE**
SETFS	**CL:DEFSETF, CL:DEFINE-SETF-METHOD**
DEFINE-TYPES	**XCL:DEF-DEFINE-TYPE**
OPTIMIZERS	**XCL:DEFOPTIMIZER**
COMMANDS	**XCL:DEFCOMMAND**

Note that the types listed above, as well as all the old File Manager types, are symbols in the INTERLISP package. In addition, the "filecoms" variable of a file and its rootname are also both in the INTERLISP package. You should be careful when typing to a Common Lisp exec to qualify all such symbols with the prefix **IL:**; e.g.,

```
3>(setq il:foocoms '((il:functions bar) (il:prop il:filetype il:foo)))
```

to indicate you want the function BAR (in the current package) to live on a file with rootname FOO, and also that FOO's FILETYPE property should be saved.

Reader Environments and the File Manager

(II:17.1)

In order for **READ** to correctly read back the same expression that **PRINT** printed, it is necessary that both operations be performed in the same reader environment, i.e., the collection of parameters that affect the way the reader interprets the characters appearing on the input stream. In previous releases of Interlisp there was, for all practical purposes, a single such environment, defined entirely by the readtable *FILERDTBL*. In the Lyric release of Lisp there are two significantly different readtables in which to read (Common Lisp and Interlisp). In addition, there are more parameters than just the readtable that can potentially affect **READ:** the current package and the read base (the bindings of ***PACKAGE*** and ***READ-BASE***).

To handle this diversity, a new type of object is introduced, the **READER-ENVIRONMENT**, consisting of a readtable, a package, and a read/print base. Every file produced by the File Manager has a header at the beginning specifying the reader environment for that file. **MAKEFILE** and the compiler produce this header, while **LOAD**, **LOADFNS**, and other file-reading functions read the header in order to set their reading environment correctly. Files written in older releases of Lisp lack this header and are interpreted as having been written in the environment consisting of the readtable *FILERDTBL* and the package INTERLISP. Thus, you need take no special action to be able to load Koto source files into Lyric; characters that are "special" in Common Lisp, such as colon, semi-colon and hash, are interpreted as the "ordinary" characters they were in Koto.

The File Manager’s reader environments are specified as a property list of alternating keywords and values of the form (**:READTABLE** *readtable* **:PACKAGE** *package* **:BASE** *base*). The **:BASE** pair is optional and defaults to 10. The values for *readtable* and *package* should either be strings naming a readtable and package, or expressions that can be evaluated to produce a readtable and package. In the former case, the readtable or package *must* be one that already exists in a virgin Lisp sysout (or at least in any Lisp image in which you might attempt *any* operation that reads the file). If an expression is used, care should be exercised that the expression can be evaluated in an environment where no packages or readtables, other than the documented ones, are presumed to exist. For hints and guidelines on writing the *package* expression for files that create or use their own private packages, please see Chapter 11 of the *Common Lisp Implementation Notes*.

When **MAKEFILE** is writing a source file, it uses the following algorithm to determine the reading environment for the new file:

1. If the root name for the file has the property **MAKEFILE-ENVIRONMENT**, the property’s value is used. It should be in the form described above. Note that if you want the file always to be written in this environment, you should save the **MAKEFILE-ENVIRONMENT** property itself on the file, using a (**PROP MAKEFILE-ENVIRONMENT** *file*) command in the filecoms.

2. If a previous version of the file exists, **MAKEFILE** uses the previous version’s environment. **MAKEFILE** does this even when given option **NEW** or the previous version is no longer accessible, assuming it still has the previous version’s environment in its cache. If the previous version was written in an older release, and hence has no explicit reader environment, **MAKEFILE** uses the environment (**:READTABLE** "INTERLISP" **:PACKAGE** "INTERLISP" **:BASE** 10).

3. If no previous version exists (this is a new file), **MAKEFILE** uses the value of ***DEFAULT-MAKEFILE-ENVIRONMENT***, initially **(:READTABLE** "XCL" **:PACKAGE** "INTERLISP" **:BASE** 10).

Note that changing the value of ***DEFAULT-MAKEFILE-ENVIRONMENT*** only affects new files. If you decide you don’t like the environment in which an existing file is written, you must give the file a **MAKEFILE-ENVIRONMENT** property to override any prior default.

Since the XCL readtable is case-insensitive, you should avoid using it for files that contain many mixed-case symbols or old-style Interlisp comments, as these will be printed with many escape delimiters. This is why the default for reprinted Koto sources is the INTERLISP readtable.

The readtable named LISP (the pure Common Lisp readtable) should ordinarily not be used as part of a **MAKEFILE** environment. It exists solely for the use of "pure" Common Lisp (as in the CL Exec), and thus has no provision for font escapes (inserted by the Lisp prettyprinter) to be treated as whitespace. Most users will want to use either XCL or INTERLISP as the readtable for files.

If the environment for the new version of the file differs from that of the previous version, **MAKEFILE** copies unchanged FNS definitions by actually reading from the old file, rather than just copying characters as it otherwise would. Similarly, when **RECOMPILE** or **BRECOMPILE** attempt to recompile a file for which the previous compiled version's reader environment is different, they must compile afresh all the functions on the file, i.e., they behave like **TCOMPL** or **BCOMPL**.

Modifying Standard Readtables

In the past, programmers have been periodically tempted to change standard readtables, such as **T** and **FILERDTBL**, typically by adding macros to read certain objects in a convenient way. For example, the PQUOTE LispUsers module defined single quote as a macro in **FILERDTBL**. Unfortunately, changing a standard readtable means that unless you are very careful, you cannot read other users' files that were not written with your change, and they cannot read your files without obtaining your macro. Furthermore, the effects are often subtle. Rather than breaking, the system merely reads the file incorrectly. For example, reading a file written with PQUOTE in an environment lacking PQUOTE produces many symbols with a single quote packed on the front.

This confusion can be avoided with MAKEFILE reader environments. To add your own special macro:

1. Copy some standard readtable; e.g., (COPYRDTBL "INTERLISP").
2. Give it a distinguished name of its own, by using (READTABLEPROP *rdtbl* 'NAME "*yourname*").
3. Make your change in the copied readtable.
4. Use your new private readtable to write your files: use its name ("*yourname*") in the **MAKEFILE-ENVIRONMENT** property of selected files and/or change ***DEFAULT-MAKEFILE-ENVIRONMENT*** to affect all your new files.
5. Make sure to save your new readtable. It is usually most convenient to include the code to create it (steps 1-3) in your system initialization, but you could even write a self-contained expression to use in a single file's **MAKEFILE-ENVIRONMENT** property.

With this strategy, your system will read all files in the proper environment—your own files with your private readtable and other users' files in their environments, including the standard environments, which you have carefully avoided polluting. If another user tries to load one of your files into an environment that doesn't know about your private readtable, **LOAD** will give an error immediately (readtable not found), rather than loading the file quietly but incorrectly.

Programmer's Interface to Reader Environments

The following function and macro are available for programmers to use. Note that reader environments only control the parameters that determine read/print consistency. There are other parameters, such as ***PRINT-CASE***, that affect the appearance of the output without affecting its ability to be read. Thus, reader environments are not sufficient to handle problems of, for example, repainting expressions on the display in exactly the same total environment in which they were first written.

(**MAKE-READER-ENVIRONMENT** *PACKAGE READTABLE BASE*) [Function]

Creates a **READER-ENVIRONMENT** object with the indicated components. The arguments must be valid values for the variables ***PACKAGE***, ***READTABLE*** and ***PRINT-BASE***; names are not sufficient. If any of the arguments is **NIL**, the current value of the corresponding variable is used. Thus **(MAKE-READER-ENVIRONMENT)** returns an object that captures the current environment.

(**WITH-READER-ENVIRONMENT** *ENVIRONMENT . FORMS*) [Macro]

Evaluates each of the *FORMS* with ***PACKAGE***, ***READTABLE***, ***PRINT-BASE*** and ***READ-BASE*** bound to the values in the *ENVIRONMENT* object. Both ***PRINT-BASE*** and ***READ-BASE*** are bound to the single *BASE* value in the environment.

(**GET-ENVIRONMENT-AND-FILEMAP** *STREAM DONTCACHE*) [Function]

Parses the header of a file produced by the File Manager and returns up to four values:

1. The reader environment in which the file was written;
2. The file's "filemap", used to locate functions on the file;
3. The file position where the FILECREATED expression starts; and
4. A value used internally by the File Manager.

STREAM can be a full file name, in which case this function returns NIL unless the information was previously cached. Otherwise, *STREAM* is a stream open for input on the file. It must be randomly accessible (unless information is available from the cache). If the file is in Common Lisp format (it begins with a comment), then value 1 is the default Common Lisp reader environment (readtable LISP, package USER) and the other values are NIL. Otherwise, if the file is not in File Manager format, values 1 and 2 are NIL, 3 is zero.

If *DONTCACHE* is true, the function does not cache any information it learns about File Manager files; otherwise, the information is cached to speed up future inquiries.

Section 17.1 Loading Files

(II:17.5)

Integration of Interlisp and Common Lisp LOAD functions

There are four kinds of files that can be loaded in Lisp:

1. Interlisp and Common Lisp source files produced by the File Manager using, for example, the **MAKEFILE** function.
2. Standard Common Lisp source files produced with a text editor either in Lisp or from some other Common Lisp implementation.
3. DFASL files of compiled code, produced by the new XCL Compiler, **CL:COMPILE-FILE** (extension DFASL)
4. LCOM files of compiled code, produced by the old Interlisp Compiler (**BCOMPL, TCOMPL**).

Types 1 and 4 were the only kind of files that you could load in Koto; types 2 and 3 are new with Lyric. Both **IL:LOAD** and **CL:LOAD** are capable of loading all four kinds of files. However, they use the following rules to make the types of files unambiguous so that they can be loaded in the correct reader environment.

- If the file begins with an open parenthesis (possibly after whitespace and font switch characters), it is assumed to be of type 1 or 4: files produced by the File Manager. The first expression on the file (at least) is assumed to be written in the old **FILERDTBL** environment; for new Lyric files this expression defines the reader environment for the remainder of the file. See the section, Reader Environments and File Manager for details.
- If the file begins with the special FASL signature byte (octal 221), it is assumed to be a compiled file in FASL format, and is processed by the FASL loader. The FASL loader ignores the *LDFLG* argument to **IL:LOAD**, treating all files as though *LDFLG* were **SYSLOAD** (redefinition occurs, is not undoable, and no File Manager information is saved).
- If the file begins with a semicolon, it is assumed to be a pure Common Lisp file. The expressions on the file are read with the standard Common Lisp readtable and in package USER (unless a package argument was given to **LOAD**; see below).
- If the file begins with any other character, **LOAD** doesn't know what to do. Currently, it treats the file as a pure Common Lisp file (as if it started with a comment).

Thus, if you prepare Common Lisp text files you should be sure to begin them with a comment so that **LOAD** can tell the file is in Common Lisp syntax.

The function **CL:LOAD** accepts an additional keyword **:PACKAGE**, whose value must be a package object; the function **IL:LOAD** similarly has an optional fourth argument *PACKAGE*. If a package argument is given, then **LOAD** reads Common Lisp text files (type 2 above) with ***PACKAGE*** bound to the specified package. In the case of File Manager files (types 1 and 4), the package argument overrides the package specified in the file's reader environment.

(II:17.6-17.8)

The Interlisp functions **LOADFNS**, **LOADFROM**, **LOADVARS** and **LOADCOMP** do not work on FASL files. They do still work on files produced by the old compiler (extension LCOM).

(II:17.9)

FILESLOAD (also used by the File Manager's **FILES** command) now searches for compiled files by looking for a file by the specified name whose extension is in the list ***COMPILED-EXTENSIONS***. The default value for ***COMPILED-EXTENSIONS*** in the Lyric release is (DFASL LCOM). It searches the list of extensions in order for each directory on the search path. This means that FASL files are loaded in preference to old-style compiled files.

Section 17.2 Storing Files

The Lyric release contains two different compilers, the Interlisp Compiler that was present in Koto and previous releases, and the new XCL Compiler (see the next section, Chapter 18 Compiler). With more than one compiler available, the question arises as to which compiler will be used by the functions **CLEANUP** and **MAKEFILE.** The default behavior of these functions in Lyric is to always use the new XCL Compiler. This default can be changed, either on a file-by-file basis or system-wide. Most users, however, will have no need to change the default.

When the *C* or *RC* option has been given to **MAKEFILE,** the system first looks for the value of the **FILETYPE** property on the symbol naming the file. For example, for the file "{DSK}<LISPFILES>MYFILE", the property list of the symbol **MYFILE** would be examined.

The **FILETYPE** property should be either a symbol from the list below or a list containing one of those symbols. The following symbols are allowed and have the given meanings:

:TCOMPL	Compile this file by calling either **TCOMPL** or **RECOMPILE**, depending upon which of the C or RC options was passed to **MAKEFILE**.
:BCOMPL	Compile this file by calling either **BCOMPL** or **BRECOMPILE**, depending upon which of the C or RC options was passed to **MAKEFILE**. This is equivalent to the Koto behavior.
:COMPILE-FILE	Compile this file by calling **CL:COMPILE-FILE**, regardless of which option was passed to **MAKEFILE**.

If no **FILETYPE** property is found, then the function whose name is the value of the variable ***DEFAULT-CLEANUP-COMPILER*** is used. The only legal values for this variable are **TCOMPL**, **BCOMPL**, and **CL:COMPILE-FILE**. Initially, ***DEFAULT-CLEANUP-COMPILER*** is set to **CL:COMPILE-FILE**.

If you choose to set the **FILETYPE** property of file name, you should take care that the filecoms for that file saves the value of that property on the file. This will ensure that the same compiler will be used every time the file is loaded. To save the value of the property, you should include a line in the coms like the following:

```
 (PROP FILETYPE MYFILE)
```

where MYFILE is the symbol naming your file.

Section 17.8.2 Defining New File Manager Types

(II:17.30)

The File Manager has been extended to allow File Manager types that accept any Lisp object as a name. A consequence of this is that any user-defined type's **HASDEF** function should be prepared to accept objects other than symbols as the *NAME* argument. Names are compared using **EQUAL**.

Definers: A New Facility for Extending the File Manager

The Definer facility is provided to make the process of adding a certain common kind of File Manager type easy. All of the new File Manager types in the Lyric release (including **FUNCTIONS**, **VARIABLES**, **STRUCTURES**, etc.) and almost all of the new defining macros (including **CL:DEFUN**, **CL:DEFPARAMETER**, **CL:DEFSTRUCT**, etc.) were themselves created using the Definer facility.

In previous releases, adding new types and commands to the File Manager involved deeply understanding the way in which it worked and defining a number of functions to carry out certain operations on the new type/command. Further, making functions and macros save away definitions of the new type was similarly subtle and generally difficult or complicated to do. With the addition of Common Lisp, it was realized that a large number of new types and commands would be added, all needing essentially the same implementation of the various operations. In addition, many new defining macros were to be added and all of them needed to save definitions.

As an explanation of the Definer facility, we will describe how **VARIABLES** and **CL:DEFPARAMETER** could be added into the system, if they were not already there.

First, a little background about our example. The macro **CL:DEFPARAMETER** is used in Common Lisp to globally declare a given variable to be special and to give it an initial value. (For the purposes of this example, we will ignore the documentation-string given to real **CL:DEFPARAMETER** forms.) The value of a call to the macro should be the name of the variable being defined. An acceptable definition of this macro might appear as follows:

```
(DEFMACRO CL:DEFPARAMETER (SYMBOL EXPRESSION)
   ‘(PROGN
      (CL:PROCLAIM ’(CL:SPECIAL ,SYMBOL))
      (SETQ ,SYMBOL ,EXPRESSION)
      ’,SYMBOL))
```

There are some problems with using such a simple definition in the Lisp environment, however. For example, if a call to this macro were typed to the Exec, the File Manager would not be told to notice it. Thus, there would be no convenient way to remember to add the form to the filecoms of some file and thus to save it away. Also, note that the macro does not pay attention to the **DFNFLG** variable; thus, loading a file containing a **CL:DEFPARAMETER** form would always set the variable to the value of the initial expression, even when **DFNFLG** was set to **ALLPROP**. This could make editing code using this variable difficult.

We will now proceed to fix these problems by getting the Definer facility involved. There are two steps involved in using Definers:

- Unless one of the currently-existing File Manager types is appropriate for definitions using the new macro, a new type must be created. The macro **XCL:DEF-DEFINE-TYPE** is used for this purpose.
- The macro must be defined in such a way that the File Manager can tell that it should notice and save uses of the macro and under which File Manager type the uses should be saved. The macro **XCL:DEFDEFINER** is used for this purpose.

Since we are pretending for the example that the File Manager type **VARIABLES** is not defined, we decide that definitions using **CL:DEFPARAMETER** should not be given any of the already-existing types. We must define a type, therefore, and we decide to call it **VARIABLES**. The following **XCL:DEF-DEFINE-TYPE** form will do the trick:

```
(XCL:DEF-DEFINE-TYPE VARIABLES "Common Lisp
variables")
```

The first argument to **XCL:DEF-DEFINE-TYPE** is the name for the new type. The second argument is a descriptive string, to be used when printing out messages about the type.

With the new type thus created, we can now use **XCL:DEFDEFINER** to redefine the macro. Simply changing the word **DEFMACRO** into **XCL:DEFDEFINER** and adding an argument specifying the new type suffices to change our earlier definition into a use of the Definer facility:

```
(XCL:DEFDEFINER CL:DEFPARAMETER VARIABLES
                                (SYMBOL EXPRESSION)
   ‘(PROGN
      (CL:PROCLAIM ’(CL:SPECIAL ,SYMBOL))
      (SETQ ,SYMBOL ,EXPRESSION)
      ’,SYMBOL))
```

(In fact, we could also remove the final **’,SYMBOL**; **XCL:DEFDEFINER** automatically arranges for the new macro to return the name of the new definition.) Now, if we were to type the form

```
(CL:DEFPARAMETER *FOO* 17)
```

into the Exec and then call the function **FILES?**, we would be presented with something like the following:

```
24> (FILES?)
the Common Lisp variables: *FOO*
...to be dumped.  want to say where the above go?
```

As with other File Manager types, our definitions are being kept track of. If we answer Yes to the above question and specify a file in which to save the definition, a command like the following will be added to the filecoms:

```
(VARIABLES *FOO*)
```

Actually, the output from **FILES?** as shown above is not quite accurate. In reality, we would also be asked about the following:

```
the Common Lisp functions/macros: CL:DEFPARAMETER
the Definition types: VARIABLES
```

The File Manager is also watching for new types and new Definers being created and will let us save those definitions as well. These would be listed in the filecoms as follows:

```
(DEFINE-TYPES VARIABLES)
(FUNCTIONS CL:DEFPARAMETER)
```

All of these definitions are full-fledged File Manager citizens. The functions **GETDEF**, **HASDEF**, **PUTDEF**, **DELDEF**, etc. all work with the new type. We can edit the definition of ***FOO*** above simply by specifying the type to the **ED** function:

```
(ED '*FOO* 'VARIABLES)
```

When we exit the editor, the new definition will be saved and, unless **DFNFLG** is set to **PROP** or **ALLPROP**, evaluated.

It is now time to fully describe the macros **XCL:DEF-DEFINE-TYPE** and **XCL:DEFDEFINER**.

XCL:DEF-DEFINE-TYPE *NAME DESCRIPTION* &KEY :UNDEFINER [*Macro*]

Creates a new File Manager type and command with the given *NAME*. The string *DESCRIPTION* will be used to describe the type in printed messages. The new type implements **PUTDEF** operations by evaluating the definition form, **GETDEF** and **HASDEF** by looking up the given name in an internal hash-table, using **EQUAL** as the equality test on names, and **DELDEF** by removing any named definition from the hash-table. If the **:UNDEFINER** argument is provided, it should be the name of a function to be called with the *NAME* argument to any **DELDEF** operations on this type. The **:UNDEFINER** function can perform any other operations necessary to completely delete a definition.

XCL:DEF-DEFINE-TYPE forms are File Manager definitions of type **DEFINE-TYPES**.

As an example of the full use of **XCL:DEF-DEFINE-TYPE**, here is the complete definition of the type VARIABLES as it exists in the Lyric release:

```
(XCL:DEF-DEFINE-TYPE  VARIABLES "Common Lisp variables"
                      :UNDEFINER UNDOABLY-MAKUNBOUND)
```

The function **UNDOABLY-MAKUNBOUND** is described in Appendix D of these Release Notes.

XCL:DEFDEFINER {*NAME* | (*NAME* {*OPTION*}*)} *TYPE ARG-LIST* &BODY *BODY* [*Macro*]

Creates a macro named *NAME*, calls to which are seen as File Manager definitions of type *TYPE*. *TYPE* must be a File Manager type previously defined using **XCL:DEF-DEFINE-TYPE**. *ARG-LIST* and *BODY* are precisely as in **DEFMACRO**. A macro defined using

XCL:DEFDEFINER differs from one defined using **DEFMACRO** in the following ways:

- *BODY* will be evaluated if and only if the value of **DFNFLG** is not one of **PROP** or **ALLPROP**.
- The form returned by *BODY* will be evaluated in a context in which the File Manager has been temporarily disabled. This allows Definers to expand into other Definers without the subordinate ones being noticed by the File Manager.
- Calls to Definers return the name of the new definition (as, for example, **CL:DEFUN** and **CL:DEFPARAMETER** are defined to do).
- Calls to Definers are noticed and remembered by the File Manager, saved as a definition of type *TYPE*.
- SEdit- and Interlisp-style comment forms (those with a CAR of IL:*) are stripped from the macro call before it is passed to *BODY*. (This comment-removal is partially controlled by the value of the variable ***REMOVE-INTERLISP-COMMENTS***, described below.)

The following *OPTION*s are allowed:

`(:UNDEFINER` *FN*`)`

If **DELDEF** is called on a name whose definition is a call to this Definer, *FN* will be called with one argument, the name of the definition. This option allows for Definer-specific actions to be taken at **DELDEF** time. This is useful when more than one Definer exists for a given type. *FN* should be a form acceptable as the argument to the **FUNCTION** special form.

`(:NAME` *NAME-FN*`)`

By default, the Definer facility assumes that the first argument to any macro defined using **XCL:DEFDEFINER** will be the name under which the definition should be saved. This assumption holds true for almost all Common Lisp defining macros, including **CL:DEFUN**, **CL:DEFMACRO**, **CL:DEFPARAMETER** and **CL:DEFVAR**. It doesn’t work, however, for a few other forms, such as **CL:DEFSTRUCT** and **XCL:DEFDEFINER** itself. When defining a macro for which that assumption is false, the **:NAME** option should be used. *NAME-FN* should be a function of one argument, a call to the Definer. It should return the Lisp object naming the given definition (most commonly a symbol, but any Lisp object is permissible). For example, the **:NAME** option in the definitions of **CL:DEFSTRUCT** and **XCL:DEFDEFINER** is as follows:

```
(:NAME (LAMBDA (FORM)
          (LET ((NAME (CADR FORM)))
             (COND ((LITATOM NAME)
                    NAME)
                   (T  (CAR NAME))))))
```

NAME-FN should be a form acceptable as the argument to the **FUNCTION** special form (i.e., a symbol naming a function or a LAMBDA-form).

(:PROTOTYPE *DEFN-FN*)

When the editor function **ED** is passed a name with no definitions, the user is offered a choice of several ways to create a prototype definition. Those choices are specified with the **:PROTOTYPE** option to **XCL:DEFDEFINER.** *DEFN-FN* should be a function of one argument, the name to be defined using this Definer. *DEFN-FN* should return either NIL, if no definition of that name can be created with this Definer, or a form that, when evalauted, would create a definition of that name. For example, the **:PROTOTYPE** option for **CL:DEFPARAMETER** might look as follows:

```
(:PROTOTYPE (LAMBDA (NAME)
               (AND (LITATOM NAME)
                    '(CL:DEFPARAMETER ,NAME "Value"))))
```

An example using all of the features of **XCL:DEFDEFINER** is the definition of **XCL:DEFDEFINER** itself, which begins as follows:

```
(XCL:DEFDEFINER (XCL:DEFDEFINER
                    (:UNDEFINER \DELETE-DEFINER)
                    (:NAME
                       (LAMBDA (FORM)
                          (LET ((NAME (CADR FORM)))
                             (COND ((LITATOM NAME)
                                        NAME)
                                   (T  (CAR NAME))))))
                    (:PROTOTYPE
                       (LAMBDA (NAME)
                          (AND (LITATOM NAME)
                               '(XCL:DEFDEFINER ,NAME "Type"
                                                ("Arg List")
                                                "Body")))))
                FUNCTIONS
                (NAME-AND-OPTIONS TYPE ARG-LIST &BODY BODY)
    ...)
```

The following variable is used in the process of removing SEdit- and Interlisp-style comments from Definer forms:

REMOVE-INTERLISP-COMMENTS [*Variable*]

Interlisp-style comments are forms whose **CAR** is the symbol **IL:***. It is possible for certain lists in Lisp code to begin with **IL:*** but not be a comment (for example, a **SELECTQ** clause). When such a list is discovered, the value of ***REMOVE-INTERLISP-COMMENTS*** is examined. If it is **T**, the list is assumed to be a comment and is removed without comment. If it is **:WARN**, a warning message is printed, saying that a possible comment was not stripped from the code. If ***REMOVE-INTERLISP-COMMENTS*** is **NIL**, the list is not removed, but no warning is printed. This variable is initially set to **:WARN**.

(CL:EVAL-WHEN *WHEN COM*$_1$ *... COM*$_N$*)* [*File Package Command*]

Interprets each of the commands *COM*$_1$ *... COM*$_N$ as a file package command, but output is wrapped in CL:EVAL-WHEN.

EXAMPLE:

```
(CL:EVAL-WHEN (CL:EVAL CL:COMPILE)
 (OPTIMIZERS FOO))
```

will cause the following to be written to the file:

```
(CL:EVAL-WHEN (CL:COMPILE)
  (DEFOPTIMIZER FOO <optimizer for FOO>))
```

Chapter 18 Compiler

The Lyric release contains two distinct Lisp compilers:

- The Interlisp Compiler, described in detail in Section 18 of the *IRM,*
- The new XCL Compiler, described in the *Common Lisp Implementation Notes*.

The Interlisp Compiler provides compatibility with previous releases of Interlisp-D. It continues to work in very much the same way as it did in Koto; as before, it compiles all of the Interlisp language. The Interlisp Compiler does not, however, compile the Common Lisp language and will not be extended to do so. The Lyric release is the last release to contain the Interlisp Compiler as a component; future releases will have only the new XCL Compiler. The XCL Compiler is designed to handle both Interlisp and Common Lisp.

Several incompatible changes have been made in the compiled object code produced by the Interlisp Compiler. This means that *all user code must be recompiled in Lyric*. Code compiled in Koto or previous releases will not load into Lyric, and code compiled in Lyric wil not load into earlier releases. The filename extension for Interlisp compiled files has been changed from DCOM to LCOM in order to minimize possible confusion.

The XCL Compiler writes its output on a new kind of object file, the DFASL file. These files are quite different from the DCOM/LCOM files produced by the Interlisp Compiler. DFASL files are somewhat more compact, much faster to load and can represent a wider range of data objects than was possible in LCOMs.

Interlisp source files from Koto can be compiled using the new XCL compiler. However, some files need to be remade in Lyric before compilation: files containing bitmaps, Interlisp arrays, or the **UGLYVARS** and/or **HORRIBLEVARS** File Manager commands. To compile such a file, first **LOAD** it, then call **MAKEFILE** to write it back out. This action causes the bitmaps and other unusual objects to be written back in a format acceptable to the new compiler.

The default behavior of the File Manager's **CLEANUP** and **MAKEFILE** functions is to use the new XCL Compiler to compile files, rather than the old Interlisp Compiler. To change this behavior, see Section 17.2, Storing Files.

Note that if you call the compiler explicitly, rather than via **CLEANUP** or **MAKEFILE**, you should be careful to specify the correct compiler. The new compiler is invoked by calling **CL:COMPILE-FILE**. If you inadvertantly call **BCOMPL** on a file for which **CLEANUP** has routinely been using the new XCL compiler, there are two undesirable consequences: (1) Any Common Lisp functions on the file will not be compiled (the Interlisp compiler does not recognize **CL:DEFUN**), and (2) the DFASL files produced by earlier calls on the XCL compiler will still be loaded by **FILESLOAD** in preference to the LCOM file produced by **BCOMPL**.

Lisp provides a facility, **XCL:DEFOPTIMIZER**, by which you can advise the compiler about efficient compilation of certain functions and macros. **XCL:DEFOPTIMIZER** works with both the old Interlisp Compiler and the Lyric XCL Compiler. See the *Common Lisp Implementation Notes* for a description of the compiler.

Warning when Loading Compiled Files

CAUTION: Files compiled in Medley cannot be loaded back into Lyric. Medley-compiled .LCOM and .DFASL files will produce an error message when loaded into Lyric. (Lyric-compiled .LCOM and .DFASL files can be loaded and run in Medley.) If you need to run a Medley file in Lyric, load the source file and use the Lyric compiler.

Warning with Declarations

CAUTION: There is a feature of the BYTECOMPILER that is not supported by either the XCL compiler or SEdit. It is possible to insert a comment at the beginning of your function that looks like

```
(* DECLARATIONS: --)
```

The tail, or -- section, of this comment is taken as a set of local record declarations which are then used by the compiler in that function just as if they had been declared globally. The XCL compiler does not directly support this feature. If the body of the function gets DWIMIFIED for some other reason, the record declarations will happen to be noticed, otherwise they will not be seen and the compiler will signal an error if it can’t find an appropriate top-level record definition.

There are two caveats that you should note:

1. The compiler will give error messages "undefined record name ..." for the records that are declared this way, but will generate correct code.

2. SEdit does not recognize such declarations. Thus, if the "Expand" command is used in SEdit, the expansion will not be done with these record declarations in effect. The code that you see in the editor will not be the same code compiled by the BYTECOMPILER.

Section 18.3 Local Variables and Special Variables

(II:18.5)

The new execs always use the Common Lisp interpreter, causing LET and PROG statements at top level, particularly in a so-called Interlisp exec, to create lexical bindings, rather than deep or "special" bindings This can be worked around by setting **il:specvars** to T, which will cause the interpreter to create special bindings for all variables. This can also be worked around by wrapping the form to be "interlisp evaluated" in the IL:INTERLISP special form, which causes the Interlisp interpreter to be invoked.

Chapter 19 Masterscope

Masterscope is now a Lisp Library Module, not part of the environment.

Chapter 21 CLISP

CLISP infix forms do not work under the Common Lisp evaluator; only "clean" CLISP prefix forms are supported. You should run DWIMIFY in Koto on all other CLISP code before attempting to load it in Lyric. The remainder of this note describes the specific limitations on CLISP in Lyric.

There are two broad classes of transformations that DWIM applies to Lisp code:

1. A sort of macro expander that transforms **IF**, **FOR**, **FETCH**, etc. forms into "pure" Lisp code in well-defined ways.
2. A heuristic "corrector" that performs spelling correction and transforms CLISP infix forms such as X+Y into (PLUS X Y), sometimes having to make guesses as to whether X+Y might really have been the name of a variable.

An operational way of distinguishing the two is that DWIMIFY applied to code of type (1) makes no alterations in the code, whereas for code of type (2) it physically changes the form. Another difference is that code of type (2) must be dwimified before it can be compiled (user typically sets **DWIMIFYCOMPFLG** to **T**), whereas the compiler is able to treat code of type (1) as a special kind of macro.

Broadly speaking, code of type (2) is no longer fully supported. In particular, DWIM is invoked only when the code is encountered by the Interlisp evaluator. This means code typed to an "Old Interlisp" Executive, and code inside of an interpreted Interlisp function. Furthermore, some CLISP infix forms no longer DWIMIFY correctly. It is likely that CLISP infix will not be supported at all in future releases.

Expressions typed to the new Executives and inside of Common Lisp functions are run by the Common Lisp evaluator (**CL:EVAL**). As far as this evaluator is concerned, DWIM does not exist, and forms beginning with "CLISP" words **(IF**, **FOR**, **FETCH,** etc) are macros. These macros perform no DWIM corrections, so all of the subforms must be correct to begin with. This is a change from past releases, where the DWIM expansion of a CLISP word form also had the side effect of transforming any CLISP infix that it might have contained. For example, the macro expansion of

```
(if X then Y+1)
```

treats Y+1 as a variable, rather than as an addition. The correct form is

```
(if X then (PLUS Y 1)),
```

which is the way an explicit call to DWIMIFY would transform it.

If you have CLISP code from Koto you are advised to DWIMIFY the code before attempting to run or compile it in Lyric. Because of differences in the environments, not all CLISP constructs will DWIMIFY correctly in Lyric. In particular, the following do not work reliably, or at all:

1. The list-composing constructs using < and > do not DWIMIFY if the < is unpacked (an isolated symbol), because in Common Lisp, < is a perfectly valid CAR of form. On the other hand, the closing > *must* be unpacked if the last list element is quoted, since, for example, `(<A 'B>)` reads as `(<A (QUOTE B>))`.

2. Because of the conventional use of the characters `*` and `-` in Common Lisp names, those characters are only recognized as CLISP operators when they appear unpacked.

3. On the other hand, the operators + and / are the names of special variables in Common Lisp (Steele, p. 325), and hence cause no error when passed unpacked to the evaluator. Thus `(LIST X + Y)` returns a list of three elements, with no resort to DWIM; however, the parenthesized version `(LIST (X + Y))` and the packed version `(LIST X+Y)` both work.

If you routinely DWIMIFY code, so that no CLISP infix forms (type 2 above) remain on your source files, you may not need to make any changes. However, note that the fact that DWIMIFY of prefix forms implicitly performed infix transformations can hide code that escaped being completely dwimified before being written to a file.

There is a further caution regarding even routinely dwimified code that has not been edited since before Koto. Two uses of the assignment operator (_) no longer work, if not explicitly dwimified, because their canonical form (the output of DWIMIFY) has changed, and the old form is no longer supported when the form is simply evaluated, macro-expanded, or compiled (with **DWIMIFYCOMPFLG** = **NIL**):

1. Iterative statement bindings must always be lists. For example, the old form

   ```
   (bind X_2 for Y in --)
   ```

is now canonically

```
(bind (X _ 2) for Y in --).
```

2. In a WITH expression, assignments must be dwimified to remove _. For example, the old form

```
(with MYRECORD MYFIELD _ (FOO))
```

is now canonically

```
(with MYRECORD (SETQ MYFIELD (FOO))).
```

DWIMIFY in Koto correctly made these transformations; however, in some older releases, it did not. Such old code must be explicitly dwimified (which you can do for these cases in Lyric). The errors resulting from failure to do so can be subtle. In particular, the compiler issues no special warning when such code is compiled. For example, in case 1, the macro expansion of the old form treats the symbol `X_2` as a variable to bind, rather than as a binding of the variable `X` with initial value 2. The only hint from the compiler that anything is amiss is likely to be an indication that the variable `X` is used freely but not bound. Case 2 is even subtler: the symbols `MYFIELD` and _ are treated as symbols to be evaluated; since their values are not used, the compiler optimizes them away, reducing the entire expression to simply `(FOO)`, and there is thus no warning of any sort from the compiler.

Chapter 22 Performance Issues

Section 22.3 Performance Measuring

(II:22.8)

The Interlisp-D **TIME** function has been withdrawn and replaced with the Common Lisp **TIME** macro (the symbol **TIME** is shared between IL and CL and thus need not be typed with a package prefix). The functionality of the *TIMEN* and *TIMETYP* arguments to the old **TIME** can be had by keywords to the **TIME** macro. The *Common Lisp Implementation Notes* describe the new **TIME** macro and its associated command in more detail.

[This page intentionally left blank]