7.COMMON LISP IMPLEMENTATION

This section describes new features and enhancements that implement Common Lisp into the Lisp operating environment within the Medley release. This information supplements the *Common Lisp Implementation Notes,* Lyric release. Medley enhancements are indicated with revision bars in the right margin.

New Features Since Lyric

The following description summarizes the new Common Lisp implementation features that have been added or changed since the Lyric release.

New compiler Interface -- The Medley compiler gives better progress reports and it is now possible to invoke the compiler on any definer (not just functions, as before).

New Implementation of Defstruct -- A new version of defstruct compiles more compactly and gives more options so that defstruct has at least as much functionality as the Interlisp record package.

Adoption of features and clarifications suggested by the Common Lisp Cleanup Committee -- Among other changes, the behavior of **append** on dotted lists is now better defined, and a new function **xcl:row-major-aref** has been added.

Common Lisp Veneer on the Interlisp record package -- A collection of macros that make the use of existing Interlisp datatypes more appealing has been added.

Performance enhancements -- A closure caching scheme now insures that repeated calls to symbol-functions of the same symbol will return EQ compiled-function objects.

New opcodes have been added for several common list functions, such as **member** and **assoc**.

Common Lisp Definers

The Medley release contains a new implementation of definers and a reworking of the top level of the XCL Compiler. These represent upward compatible changes that have the effect of allowing the Common Lisp compiler to print out progress reports indicating which definer is currently being compiled. To receive the full benefit of these changes, recompile any file containing a **defdefiner** expression.

It is now possible to compile individual definers by using any of the following forms:

Compile-Definer

(xcl:compile-definer *name type*)

Compile and install the definer of type *type* named *name* .

EXAMPLE:

```
(xcl:compile-definer 'foo 'structures)
```

In this example, the definer will compile and install the structures definition of foo.

Compile-Form

(xcl:compile-form *form*)

Compile and evaluate *form*.

EXAMPLE:

```
(xcl:compile-form '(progn (defconstant c 1) (defun foo (a b) (+ c a
b))))
```

In this example, the definer will compile and evaluate the progn using compile-file semantics.

EXAMPLE:

```
(xcl:compile-form '(with-collection (dotimes (i 10) (collect i))))
```

In this example, the definer returns:

(0 1 2 3 4 5 6 7 8 9)

Define-File-Environment

Rather than establishing **il:makefile-environment** props and **il:filetypes** on the root name of a file, you can define a file environment using the form:

(xcl:define-file-environment *filename* &key *readtable package base compiler*)

This produces an object of file-manager type **xcl:file-environments**. The *filename* can be either a string or a symbol. The rootname of the file is constructed by interning the *filename* in the Interlisp package. Puts the *compiler* argument (if any) under the **il:filetype** prop of the file rootname. Puts the *readtable, package* and *base* arguments (if any) under the **il:makefile-environment** prop of the file rootname. None of the arguments are evaluated. There are no defaults.

EXAMPLE:

```
(xcl:define-file-environment myfile :package "XCL-USER" :readtable
"XCL" :compiler :compile-file)
```

In this example, *compile-file* is put under the **il:filetype** prop of *myfile.* The *readtable*, *XCL* and *compile* arguments are put under the **il:makefile-environment** prop of *myfile.*

NOTE: **xcl:define-file-environment** is a definer and hence will not be installed if **il:dfnflg** is **il:prop** or if a file is prop loaded.

Site-Name Special Uses

The following special variables are defined and may be set in your init file to inform Common Lisp of site information:

xcl:*short-site-name*

This variable is used in the function **short-site-name.**

xcl:*long-site-name*

This variable is used in the function **long-site-name**.

EXAMPLES:

```
(setq xcl:*short-site-name* "AIS")
(setq xcl:*long-site-name* "Artificial Intelligence Systems")
```

In these examples, (short-site-name) returns "AIS" and (long-site-name) returns "Artificial Intelligence Systems".

Record Access

The Medley release contains several methods for accessing existing Interlisp records using Common Lisp syntax. These features help to integrate Interlisp and Common Lisp. The following sections describe these additions.

Define-Record

(xcl:define-record *name interlisp-record-name*
&key *conc-name constructor predicate fast-accessors*) [Definer]

Creates a structures object named by the symbol *name* that provides Common Lisp accessors, settors, predicates and constructors for the Interlisp record named by the symbol *interlisp-record-name*. The Interlisp record must be defined before the **xcl:define-record** expression is evaluated. The keyword arguments are treated as in defstruct. The package of constructed names is taken from the value of ***package*** at the time of evaluation (as in defstruct). The system contains no predeclared **define-record**s.

EXAMPLE:

The form:

```
(xcl:define-record menu il:menu)
```

allows you to write:

```
(menu-items foo)  and (setf (menu-items foo) fie)
```

rather than:

```
(il:fetch (il:menu il:items) il:of foo)
```

Record-Fetch

(**xcl:record-fetch** *record field object*) [Macro]

Evaluates *object*. Does not evaluate *record* and *field*. Both *record* and *field* must be symbols. Symbols with the same p-names are interned in the Interlisp package and are used to construct an **il:fetch** form. **xcl:record-fetch** may be used with **setf** and expands to the suitable replace form.

Record-FFetch

(**xcl:record-ffetch** *record field object*) [Macro]

Similar to **xcl:record-fetch**, but an **il:ffetch** form is generated instead. Evaluates *object*. Does not evaluate *record* and *field*. Both *record* and *field* must be symbols. Symbols with the same p-names are interned in the Interlisp package and are used to construct an **il:ffetch** form. Ffetch may be used with **setf** and expands to the suitable freplace form.

Record-Create

(**xcl:record-create** *record* &rest *keyword-pairs*) [Macro]

Evaluates the second element of each pair. Does not evaluate *record (record* must be a symbol) . A symbol with the same p-name is interned in the Interlisp package and used to construct an **il:create** form. The rest of the arguments form keyword pairs. The first element of each pair should be a symbol such that a symbol with the same p-name exists in the Interlisp package and names either a valid slot for this record or is one of **:using, :copying, :reusing**, or **:smashing**.

Array Reference

(**xcl:row-major-aref** *array index*) [Function]

Returns the element of *array* given by the row-major-index *index*. The array can be of any dimension. This function can be used with **setf** .

Shadowing of Global Macros

The XCL Compiler now properly handles shadowing of global macros by lexical functions. In the Lyric Compiler, lexical functions defined with **flet** did not shadow global definitions of the same name. This has been fixed in Medley.

Evaluating Load-time Expressions

The XCL Compiler now handles **il:loadtimeconstant** correctly. The new Compiler substitutes the entire expression for each reference to the value of a load-time constant. There are potential problems if the code depends on the expression being evaluated exactly once, e.g. if it contains (IDATE).

Common Lisp Defstruct Options

The Medley release contains a new implementation of **defstruct** that offers greater compiled-code compaction, and several new extensions that increase efficiency. This implementation introduces functionality that allows **defstruct** to parallel the

Interlisp record module in flexibility. These features also help to integrate Interlisp and Common Lisp. The following sections describe these additions.

Defstruct Options

:inline

Can be one or both of :accessor and :predicate or t, implying '(:accessor :predicate) or nil, implying no optimizations allowed or :only, implying all accessors and the predicate will be inline only and not funcallable (not usable with the Lisp primitive "**funcall**"). The default is '(:accessor :predicate).

Copiers and constructors are never inline. The option (:inline :only) implies that no funcallable accessors will be generated (similarly, the predicate, if any, will not be funcallable).

:fast-accessors

Can be t or nil. t implies inline accessors will not type check. The default is nil.

Note that funcallable accessors (if any), always type check, if possible.

NOTE: This represents a change from the Lyric implementation, which allowed specification of a list of slot names that had fast inline accessors.

:template

Can be t or nil, t implies that no datatype will be instantiated. (:template t) implies no :type option. The default is nil.

Templated defstructs have no predicates, copiers or constructs. It is an error to supply any such option in combination with (:template t). Templated defstructs are intended to be used as are IL:blockrecord's. It is possible for a templated defstruct to include another templated structure, but it is an error for a standard defstruct to include a templated structure.

Funcallable accessors (accessors that may be used with the Lisp primative "funcall") share code with suitable closure templates if the defstruct is compiled with the XCL Compiler. Byte compiled defstructs still generate explicit defun's for all funcallable accessors.

Defstruct Slot Options

:type

The following specialized types are recognized:

(unsigned-byte {1 - 16})

(signed-byte {16, 32})

float, etc.

(member t nil)

il:fullpointer

il:xpointer

il:fullxpointer

Warning When Using Defstruct

Defstruct automatically generates a number of auxilliary functions without checking whether redefining those functions will affect the system. To avoid redefining key functions, you should be aware of the names that will be used. For example:

Do not attempt to define a Structure named TREE. This use of Defstruct implicitly redefines the built-in Common Lisp function COPY-TREE, which renders your system inoperable.

If you have already tried to define a (DEFSTRUCT TREE A B) structure by mistake, you will need to reload your system.

Macros for Collecting Objects

xcl:with -collection

(xcl:with-collection &body *forms***)** [Macro]

(xcl:collect *form***)** [Macro]

This pair of macros is provided for efficiently collecting objects into a list. In Common Lisp, there is no direct facility provided for doing this, so one must either push objects onto a list, then reverse it, or maintain a tail pointer to the list and use **rplacd** to add new items. The latter has an efficient implementation in Xerox Common Lisp, and **xcl:with-collection** is provided to take advantage of it.

Lexically within the body of an **xcl:with-collection**, the macro **xcl:collect** is defined. It will append the value of its argument to the end of the list being collected. The value of **xcl:with-collection** is the collected list.

xcl:collect may be used inside of functions passed as arguments to other functions.

EXAMPLE:

```
(xcl:with-collection
  (maphash
    #'(lambda (key val)
        (when (interesting-p val) (xcl:collect key)))
    the-hash-table))
```

will collect a list of all the "interesting" keys in the order that they were encountered.

It is an error to use **xcl:collect** outside the scope of an **xcl:with-collection.** Proper lexical nesting is observed, so an instance of **xcl:collect** applies to the most deeply nested **xcl:with-collection** that is is found in.

Macros for Writing Macros

xcl:once-only

(xcl:once-only *({variable}*)* &body *forms)* [Macro]

This macro is provided to aid in writing macros. **xcl:once-only** helps solve the problem of multiple evaluation of subforms of a macro.

EXAMPLE:

```
(defmacro test (reference form)
  '(setf ,reference (cons ,form ,form)))
```

This example has the problem that **form** will be evaluated twice. To avoid this, one might instead write:

```
(defmacro test (reference form)
  (let ((value (gensym)))
    '(let ((,value ,form))
       (setf ,reference (cons ,value ,value)))))
```

This solves the problem of multiple evaluation, but introduces some others. If **form** is in fact something simple, like a reference to a variable or a literal, there was no need to create the temporary variable, thus "wasting" a symbol. This can be extremely important in Xerox Common Lisp as symbol space is limited and symbols are never reclaimed. If there are many temporary values to be computed, the macro definition becomes cluttered with calls to gensym that obscure the essence of the code.

xcl:once-only helps solve these problems. For each of the variables listed, **xcl:once-only** determines if its value (at macroexpansion time) is simple: a symbol or a literal. If it is, appearances of that variable in the macroexpansion will remain unchanged. If it is not, the macroexpansion will contain code to store the value in a temporary **gensym'**ed variable and use that variable in the macroexpansion. Thus, the example could be written as

```
(defmacro test (reference form)
  (xcl:once-only (form)
    '(setf ,reference (cons ,form ,form))))
```

Then `(test (aref the-array x) y)` will expand to something like

```
(setf (aref the-array x) (cons y y))
```

while `(test (aref the-array x) (random-form))` will expand to something like

```
(let ((#:g377 (random-form)))
  (setf (aref the-array x) (cons #:g377 #:g377)))
```

Note that **xcl:once-only** does not attempt to preserve order of evaluation. If this is important then you will still have to create temporary variables yourself.

Common Lisp Append Datatypes

A clarification adopted by X3J13 involves the behavior of the APPEND function with non-lists. The cdr of the last cons in any but the last argument given to APPEND is discarded (whether NIL or not) when preparing the list to be returned. In the case where there is no last cons (i.e., the argument is not a list) in any but the last list argument, the entire argument is effectively ignored. In this situation, if the last argument is a non-list, the result of APPEND can be a non-list. NB: APPEND and COPY-LIST now produce different results for non-lists.

EXAMPLE:

```
(append '(a b c . d) '())
```

produces the result:

```
(a b c)
```

EXAMPLE:

```
(append '(a b . c) '() 3)
```

produces the result:

```
(a b . 3)
```

EXAMPLE:

```
(append  3  17)
```

produces the result:

```
17.
```

Closure Cache

The Medley sysout contains a closure cache that provides increased time and space efficiency. Less new memory is allocated because repeated calls to symbol-function of the same symbol now will cons exactly one closure object. Repeated calls to symbol-function of the same symbol now return EQ-compiled function objects.

Symbols and Packages

Pkg -goto and In-package

PKG-GOTO is now a synonym for IN-PACKAGE. The PKG-GOTO function can be used to change packages in an exec.

PKG-GOTO takes one argument, which can be either a double-quoted string, a symbol, or a package structure. This function is used to set package in an exec.

(xcl:pkg-goto *package-name* &key *nicknames use***)** [Function]

PKG-GOTO operates like IN-PACKAGE, but asks for confirmation if a new package is being created. The function is useful at the top level in the exec, to avoid creating new packages when a name is misspelled.

Defpackage Export argument

Defpackage's EXPORT argument now accepts strings. Optionally, strings can be given to :EXPORT instead of symbols. This is recommended when defpackage is used in the makefile-environment property of a file. The strings are interned in the package being defined and then exported.

Debugging Tools

Breaking

Even with HELPDEPTH set to zero, some errors do not cause a break. In Koto and the old Interlisp execs in Lyric, the workaround is:

(SETTOPVAL 'HELPFLAG 'BREAK!)

In Medley and Lyric's new execs, HELPFLAG is bound but not continually reset. The workaround:

(SETQ HELPFLAG 'BREAK!)

affects the current exec until the next time you call RESET (or control-D). If you want the change in HELPFLAG to be seen by other processes, you still need to use SETTOPVAL, and RESET any execs in which you want to see the effect.

For related information, see the Medley error system variable XCL:*BREAK-ON-SIGNALS* described in Appendix E.

Advising

In Lyric, putting a second piece of advice on a function caused the system to believe that the function was in fact not advised, so any further advice threw out the already existing advice. This has been fixed. In Medley, the correct list entries are made regardless of whether the function was previously advised.

In Lyric, loading a file with advice caused multiple instances of the advice to be instantiated. To prevent this, ADVISE is now changed in Medley in the following way: When a new piece of advice is put on a function, the system examines the already existing advice to see if the some advice is already there. If so, the old advice is removed before adding the new advice. Sameness is determined by a test similar to CL:EQUALP, except that *case* distinctions are significant in strings and characters. The priority and location of the advice is taken into account when determining the "sameness." This makes it possible, for instance, to have identical advice be both :FIRST and :LAST.

Advice is no longer replicated when loaded more than once.

The debugger and inspector now display interpreted lexical closures conveniently. Displayed lexical closure contents include the function contained, and any lexical bindings in the closure. Compiled closures are not conveniently inspectable. Common Lisp eval stack frames show their associated lexical environment in a similar manner.

The :when option to XCL:BREAK-FUNCTION no longer causes the broken function to return NIL when the break is not taken. The correct values are returned.

Argument Names Displayed for Interpreted Functions

In the debugger, the frame inspector window will now display the argument names for interpreted Common Lisp functions. Previously, it gave them pseudonames "arg0" "arg1" etc.

Lexical Variables Evaluated by Debugger

The debugger EVAL command now evaluate expressions in the lexical environment --i.e., you can evaluate an expression and use variables that are lexically bound in your code. Only the lexical environment at the point of the break can be evaluated. You can't presently back up to any given lexical environment.

EXAMPLE:

```
(defun fact(x)(if(= 1 x)nil(*x(fact(1-x)))))
(fact 4)
                    ;; breaks. if you then type
EVAL x
2
```

Pathname Component Fixed in FS-ERROR

In Lyric, only one of the three FS-ERROR conditions was passed a pathname component, resulting in the File Cacher not knowing which file had the error, or resulting in pathname being lost when PROTECTION VIOLATION or FILE SYSTEM RESOURCES EXCEEDED were signaled. This problem occurred most noticeably in Lyric when Interlisp errors were translated to XCL. This condition has been fixed in Medley. FS-ERROR now correctly receives all the pathname components.

Compiler Optimizations

Warning when using LABELS construct

In Lyric, use of the LABELS construct generated circular structure that would not get collected. Interpreted, a LABELS construct always creates this non-collectible structure. Compiled, such structure would be created if there were non-tail-recursive or mutually referencing subfunctions. The values of any closed-over variables are captured by this structure and thus also not collected, potentially causing large storage leaks. The latter situation has been relieved somewhat for Medley.

In Medley, the unavoidable circularity has been reduced to include only the mutually referencing functions, but not any of the other data that they access. Thus, the uncollectable structure is created only when a new copy of the code blocks are created, such a by compiling the function containing the LABELS rather than each time that function is called.

COMS added to dfasl files

The Medley compiler has been modified to better handle the il:define-file-info, and defpackage forms. Now, loading a dfasl file is not implicitly SYSLOAD. Since the file COMS for the file is now included in the dfasl, that file will be noticed by the file manager unless the load is explicitly SYSLOAD. (SYSLOADing of compiled lcom and dfasl files is recommended.)

In Lyric, dfasls of file manager files did not contain the COMS of the file. In Medley, COMS are present in dfasl files, just as they are in lcom files. As with lcom files, the COMS will not be loaded when the LDFLG argument to LOAD is SYSLOAD, nor will the name of the file be added to FILELST, but instead will be added to SYSFILES.

Note: We discourage loading either sort of compiled file (lcom or dfasl) with any value for LDFLG but SYSLOAD. Unless you intend to edit a file, you should always load it SYSLOAD. Even when you intend to edit it, it is usually preferable to SYSLOAD it and then load the source PROP. If there are too many source files for this to be practical, we recommend use of the WHERE-IS Library module.

While the location of definitions is made known to the edit interface when files are loaded, it can be very inefficient when files are not SYSLOADed. If, for example, you load ten compiled files with LDFLG = NIL and then evaluate (ED 'FOO), then the COMS of all ten files must be searched for definitions of each manager type with name FOO. With forty manager types this comes to 400 parses of COMS -- a time-consuming operation. If you instead load the compiled files SYSLOAD and the sources PROP, then no COMS need be searched, as checking for definitions of each manager type is sufficient.

Loadflg argument

The Medley release contains a new keyword argument to **cl:load.**

(**cl:load** *filename* &key *verbose print if-does-not-exist loadflg)*

The *loadflg* argument follows the sematics of the loadflg argument to **il:load**, with the exception that the loadflg argument will always be interned in the Interlisp package.

EXAMPLE:

```
(cl:load "Mycompiled-file.dfasl" :loadflg :sysload)
```

In this example, "Mycompiled-file.dfasl" will load without the file manager noticing that file.

Note: As explained in the previous section, we discourage loading either sort of compiled file (lcom or dfasl) with any value for *ldflg* but SYSLOAD.

Changes in CL:MAP, CL:WRITE-STRING, CL:COERCE , CL:GENSYM and IL:DEFERREDCONSTANT

In Lyric, a compiled call to CL:MAP that had been used for effect would occasionally cons up a new list anyway. It would fail in

the case that the first argument was a constant that evaluated to NIL, but not NIL itself, e.g. 'NIL. This has been fixed and no longer occurs in Medley.

CL:WRITE-STRING is now twice as fast and creates no new structure.

CL:COERCE now correctly returns the original object in all cases where Common Lisp and Lisp require it.

The CL Compiler now compiles CL:GENSYM properly.

IL:DEFERREDCONSTANT is now handled correctly by the XCL compiler.

ADD.PROCESS no longer coerces the process name to a symbol. Rather, process names are treated as case-insensitive strings. Thus, you can use strings for process names, and when typing process commands to an exec, you need not worry about getting the alphabetic case correct.

Compiler keeps Special &REST arguments

The CL Compiler now retains special &REST arguments. The Lyric compiler threw away special &REST arguments. This has been fixed in the Medley CL Compiler.

Compiler ignores TEdit formatting

COMPILE-FILE will now ignore TEdit formatting, but only if TEdit is loaded.

Compiler notices Tail-recursive Lexical Functions

The XCL Compiler now performs tail recursion elimination on FLETed lexical functions.

Compiler Error Message "BUG: Inconsistent stack depths seen"

You may occasionally see this error message while compiling. Normally, error messages from the compiler beginning with "BUG" indicate an internal compiler error. In this particular case, the compiler error may reflect an error in the code you are compiling.

There is currently no compile-time argument checking. The compiler performs an optimization that turns a tail-recursive function call into a jump back to the beginning of the function. If this tail-recursive call has the wrong number of arguments, the stack modeler in the assembler will detect this as incosistent stack depths, leading to the above error message.

EXAMPLE:

```
(defun bad-length (x n)
  (if (endp x) n (bad-length (cdr x))))
```

Compiling this form will result in the error "BUG: Inconsistent stack depths seen." The recursive call to bad-length has only one argument, but the function expects two.

Thus, if you see this error message, you should check for tail-recursive function calls with the wrong number of arguments.

Format ~C and WRITE-CHAR

In accordance with a recommendation of X3J13, the ~C FORMAT operation with no modifiers now behaves exactly the same as WRITE-CHAR for characters with no bits. The Medley release of XCL conforms to this; the Lyric release did not. If you need to obtain the Lyric behavior of ~C, use ~:C.

WITH-OUTPUT-TO-STRING and WITH-INPUT-FROM-STRING

For consistency with WITH-OPEN-STREAM and WITH-OPEN-FILE, WITH-OUTPUT-TO-STRING and WITH-INPUT-FROM-STRING now close the stream on exit from the form. WITH-OUTPUT-TO-STRING is now significantly faster when writing long strings.

[This page intentionally left blank]