CHAPTER 25
MISCELLANEOUS FEATURES

25.1. The Compiler

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

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

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

Xerox Common Lisp Extensions to Section 25.1

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

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

`:output-file`

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

`:error-file`

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

`:errors-to-terminal`

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

`:lap-file`

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

`:load`

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

`:file-manager-format`

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

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

`:process-entire-file`

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

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

Supported Features of the Interlisp Compiler

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

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

Unsupported Features of the Interlisp Byte Compiler

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

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

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

Compiler Optimizers: The XCL:Defoptimizer Facility

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

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

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

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

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

Defining optimizers

New optimizers are defined using the following macro:

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

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

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

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

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

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

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

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

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

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

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

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

Examples

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

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

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

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

Operations on Compilation Contexts

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

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

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

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

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

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

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

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

Creates a new context object with the given properties.

Examples

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

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

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

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

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

Operations on Lexical Environment Objects

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

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

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

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

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

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

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

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

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

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

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

Expanding Compiler Optimizers

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

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

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

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

25.2. Documentation

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

25.3. Debugging Tools

Breaking, Tracing and Advising: the Wrappers Facility

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

Concepts Common to Breaking, Tracing and Advising

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

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

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

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

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

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

Interlisp functions:

Lambda spread functions (ARGTYPE 0)

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

NLambda spread functions (ARGTYPE 1)

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

Lambda no-spread functions (ARGTYPE 2)

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

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

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

NLambda no-spread functions (ARGTYPE 3)

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

Common Lisp functions:

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

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

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

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

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

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

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

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

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

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

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

Breaking: Setting Debugger Breakpoints

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is equivalent to

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Tracing: Recording Function Calls and Returns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Advising: Modifying the Behavior of Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Stepping

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

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

Only interpreted code can be stepped.

To Begin Stepping

`step` *form* [*Macro*]

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

What's Displayed

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

The Next Step

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

<space> evaluate until `eval` is called again.

Next evaluate the current form without stepping its subforms, halt on the next form after this one

Finish finish evaluation without stepping any more subforms.

Debugger enter the debugger.

↑ abort all stepping, returning to top level.

Xerox Common Lisp Debugger

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

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

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

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

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

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

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

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

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

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

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

`eval` [*Debugger command*]

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

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

`ub` [*Debugger command*]

Removes the current breakpoint, if there is one.

Not available from the menu.

`value` [*Debugger command*]

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

↑ , stop [*Debugger commands*]

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

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

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

Not available from the menu.

proceed, pr [*Debugger command*]

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

For example, if you evaluated

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

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

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

Selecting No will abort the command.

ok [*Debugger command*]

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

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

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

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

`il:lastpos` [*Variable*]

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

`?=` [*Debugger command*]

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

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

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

`il:breakdelimiter` [*Variable*]

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

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

`bt` [*Debugger command*]

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

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

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

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

`dbt` [*Debugger command*]

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

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

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

`btv` [*Debugger command*]

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

Not available from the menu.

`btv!` [*Debugger command*]

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

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

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

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

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

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

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

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

For example, if the stack looks like:

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

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

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

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

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

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

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

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

Controlling When to Enter the Debugger

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

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

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

`il:helpflag` [*Variable*]

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

`il:helpdepth` [*Variable*]

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

`il:helpclock` [*Variable*]

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

`il:helptime` [*Variable*]

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

Interface to the Debugger

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

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

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

`il:autobacktraceflg` [*Variable*]

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

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

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

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

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

25.4. Environmental Inquiries

25.4.1 Time Functions and Commands

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

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

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

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

There's also an Exec command:

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

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

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

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