Programming Language—Common Lisp

25. Environment

25.1 The External Environment

25.1.1 Top level loop

The top level loop is the Common Lisp mechanism by which the user normally interacts with the Common Lisp system. This loop is sometimes referred to as the *Lisp read-eval-print loop* because it typically consists of an endless loop that reads an expression, evaluates it and prints the results.

The top level loop is not completely specified; thus the user interface is *implementation-defined*. The top level loop prints all values resulting from the evaluation of a *form*. Figure 25–1 lists variables that are maintained by the *Lisp read-eval-print loop*.

*	+	/	-
**	++	//	
***	+++	///	

Figure 25–1. Variables maintained by the Read-Eval-Print Loop

25.1.2 Debugging Utilities

Figure 25–2 shows *defined names* relating to debugging.

debugger-hook	**documentation**	**step**
apropos	**dribble**	**time**
apropos-list	**ed**	**trace**
break	**inspect**	**untrace**
describe	**invoke-debugger**	

Figure 25–2. Defined names relating to debugging

25.1.3 Environment Inquiry

Environment inquiry *defined names* provide information about the hardware and software configuration on which a Common Lisp program is being executed.

Figure 25–3 shows *defined names* relating to environment inquiry.

features	**machine-instance**	**short-site-name**
lisp-implementation-type	**machine-type**	**software-type**
lisp-implementation-version	**machine-version**	**software-version**
long-site-name	**room**	

Figure 25–3. Defined names relating to environment inquiry.

25.1.4 Time

Time is represented in four different ways in Common Lisp: *decoded time*, *universal time*, *internal time*, and seconds. *Decoded time* and *universal time* are used primarily to represent calendar time, and are precise only to one second. *Internal time* is used primarily to represent measurements of computer time (such as run time) and is precise to some *implementation-dependent* fraction of a second called an *internal time unit*, as specified by **internal-time-units-per-second**. An *internal time* can be used for either *absolute* and *relative time* measurements. Both a *universal time* and a *decoded time* can be used only for *absolute time* measurements. In the case of one function, **sleep**, time intervals are represented as a non-negative *real* number of seconds.

Figure 25–4 shows *defined names* relating to *time*.

decode-universal-time	**get-internal-run-time**
encode-universal-time	**get-universal-time**
get-decoded-time	**internal-time-units-per-second**
get-internal-real-time	**sleep**

Figure 25–4. Defined names involving Time.

25.1.4.1 Decoded Time

A ***decoded time*** is an ordered series of nine values that, taken together, represent a point in calendar time (ignoring *leap seconds*):

Second

An *integer* between 0 and 59, inclusive.

Minute

An *integer* between 0 and 59, inclusive.

Hour

An *integer* between 0 and 23, inclusive.

Date

An *integer* between 1 and 31, inclusive (the upper limit actually depends on the month and year, of course).

Month

An *integer* between 1 and 12, inclusive; 1 means January, 2 means February, and so on; 12 means December.

Year

An *integer* indicating the year A.D. However, if this *integer* is between 0 and 99, the "obvious" year is used; more precisely, that year is assumed that is equal to the *integer* modulo 100 and within fifty years of the current year (inclusive backwards and exclusive forwards). Thus, in the year 1978, year 28 is 1928 but year 27 is 2027. (Functions that return time in this format always return a full year number.)

Day of week

An *integer* between 0 and 6, inclusive; 0 means Monday, 1 means Tuesday, and so on; 6 means Sunday.

Daylight saving time flag

A *boolean* that, if *true*, indicates that daylight saving time is in effect.

Time zone

A *time zone*.

Figure 25–5 shows *defined names* relating to *decoded time*.

decode-universal-time	**get-decoded-time**

Figure 25–5. Defined names involving time in Decoded Time.

25.1.4.2 Universal Time

Universal time is an *absolute time* represented as a single non-negative *integer*—the number of seconds since midnight, January 1, 1900 GMT (ignoring *leap seconds*). Thus the time 1 is 00:00:01 (that is, 12:00:01 a.m.) on January 1, 1900 GMT. Similarly, the time 2398291201 corresponds to time 00:00:01 on January 1, 1976 GMT. Recall that the year 1900 was not a leap year; for the purposes of Common Lisp, a year is a leap year if and only if its number is divisible by 4, except that years divisible by 100 are not leap years, except that years divisible by 400 are leap years. Therefore the year 2000 will be a leap year. Because *universal time* must be a non-negative *integer*, times before the base time of midnight, January 1, 1900 GMT cannot be processed by Common Lisp.

decode-universal-time	**get-universal-time**
encode-universal-time	

Figure 25–6. Defined names involving time in Universal Time.

25.1.4.3 Internal Time

Internal time represents time as a single *integer*, in terms of an *implementation-dependent* unit called an *internal time unit*. Relative time is measured as a number of these units. Absolute time is relative to an arbitrary time base.

Figure 25–7 shows *defined names* related to *internal time*.

get-internal-real-time	**internal-time-units-per-second**
get-internal-run-time	

Figure 25–7. Defined names involving time in Internal Time.

25.1.4.4 Seconds

One function, **sleep**, takes its argument as a non-negative *real* number of seconds. Informally, it may be useful to think of this as a *relative universal time*, but it differs in one important way: *universal times* are always non-negative *integers*, whereas the argument to **sleep** can be any kind of non-negative *real*, in order to allow for the possibility of fractional seconds.

sleep

Figure 25–8. Defined names involving time in Seconds.

decode-universal-time *Function*

Syntax:

decode-universal-time *universal-time* `&optional` *time-zone*
→ *second, minute, hour, date, month, year, day, daylight-p, zone*

Arguments and Values:

universal-time—a *universal time*.

time-zone—a *time zone*.

second, *minute*, *hour*, *date*, *month*, *year*, *day*, *daylight-p*, *zone*—a *decoded time*.

Description:

Returns the *decoded time* represented by the given *universal time*.

If *time-zone* is not supplied, it defaults to the current time zone adjusted for daylight saving time. If *time-zone* is supplied, daylight saving time information is ignored. The daylight saving time flag is **nil** if *time-zone* is supplied.

Examples:

```
 (decode-universal-time 0 0) → 0, 0, 0, 1, 1, 1900, 0, false, 0

;; The next two examples assume Eastern Daylight Time.
 (decode-universal-time 2414296800 5) → 0, 0, 1, 4, 7, 1976, 6, false, 5
 (decode-universal-time 2414293200) → 0, 0, 1, 4, 7, 1976, 6, true, 5

;; This example assumes that the time zone is Eastern Daylight Time
;; (and that the time zone is constant throughout the example).
 (let* ((here (nth 8 (multiple-value-list (get-decoded-time)))) ;Time zone
        (recently (get-universal-time))
        (a (nthcdr 7 (multiple-value-list (decode-universal-time recently))))
        (b (nthcdr 7 (multiple-value-list (decode-universal-time recently here)))))
   (list a b (equal a b))) → ((T 5) (NIL 5) NIL)
```

Affected By:

Implementation-dependent mechanisms for calculating when or if daylight savings time is in effect for any given session.

See Also:

encode-universal-time, **get-universal-time**, Section 25.1.4 (Time)

encode-universal-time *function*

Syntax:

encode-universal-time *second minute hour date month year*
&optional *time-zone*

→ *universal-time*

Arguments and Values:

second, *minute*, *hour*, *date*, *month*, *year*, *time-zone*—the corresponding parts of a *decoded time*. (Note that some of the nine values in a full *decoded time* are redundant, and so are not used as inputs to this function.)

universal-time—a *universal time*.

Description:

encode-universal-time converts a time from Decoded Time format to a *universal time*.

If *time-zone* is supplied, no adjustment for daylight savings time is performed.

Examples:

```
 (encode-universal-time 0 0 0 1 1 1900 0) → 0
 (encode-universal-time 0 0 1 4 7 1976 5) → 2414296800
;; The next example assumes Eastern Daylight Time.
 (encode-universal-time 0 0 1 4 7 1976) → 2414293200
```

See Also:

decode-universal-time, **get-decoded-time**

get-universal-time, get-decoded-time *Function*

Syntax:

get-universal-time ⟨*no arguments*⟩ → *universal-time*

get-decoded-time ⟨*no arguments*⟩
→ *second, minute, hour, date, month, year, day, daylight-p, zone*

Arguments and Values:

universal-time—a *universal time*.

second, *minute*, *hour*, *date*, *month*, *year*, *day*, *daylight-p*, *zone*—a *decoded time*.

Description:

get-universal-time returns the current time, represented as a *universal time*.

get-decoded-time returns the current time, represented as a *decoded time*.

Examples:

```
;; At noon on July 4, 1976 in Eastern Daylight Time.
 (get-decoded-time) → 0, 0, 12, 4, 7, 1976, 6, true, 5
;; At exactly the same instant.
 (get-universal-time) → 2414332800
;; Exactly five minutes later.
 (get-universal-time) → 2414333100
;; The difference is 300 seconds (five minutes)
 (- * **) → 300
```

Affected By:

The time of day (*i.e.*, the passage of time), the system clock's ability to keep accurate time, and the accuracy of the system clock's initial setting.

Exceptional Situations:

An error of *type* **error** might be signaled if the current time cannot be determined.

See Also:

decode-universal-time, **encode-universal-time**, Section 25.1.4 (Time)

Notes:

```
(get-decoded-time) ≡ (decode-universal-time (get-universal-time))
```

No *implementation* is required to have a way to verify that the time returned is correct. However, if an *implementation* provides a validity check (*e.g.*, the failure to have properly initialized the system clock can be reliably detected) and that validity check fails, the *implementation* is strongly encouraged (but not required) to signal an error of *type* **error** (rather than, for example, returning a known-to-be-wrong value) that is *correctable* by allowing the user to interactively set the correct time.

sleep *Function*

Syntax:

sleep *seconds* → **nil**

Arguments and Values:

seconds—a non-negative *real*.

Description:

Causes execution to cease and become dormant for approximately the seconds of real time indicated by *seconds*, whereupon execution is resumed.

Examples:

```
 (sleep 1) → NIL

;; Actually, since SLEEP is permitted to use approximate timing,
;; this might not always yield true, but it will often enough that
;; we felt it to be a productive example of the intent.
 (let ((then (get-universal-time))
       (now  (progn (sleep 10) (get-universal-time))))
   (>= (- now then) 10))
→ true
```

Side Effects:

Causes processing to pause.

Affected By:

The granularity of the scheduler.

Exceptional Situations:

Should signal an error of *type* **type-error** if *seconds* is not a non-negative *real*.

apropos, apropos-list *Function*

Syntax:

apropos *string* &optional *package* → ⟨*no values*⟩

apropos-list *string* &optional *package* → *symbols*

Arguments and Values:

string—a *symbol name designator*.

package—a *package designator* or **nil**. The default is **nil**.

symbols—a *list* of *symbols*.

Description:

These functions search for *interned symbols* whose *names* contain the substring *string*.

For **apropos**, as each such *symbol* is found, its name is printed on *standard output*. In addition, if such a *symbol* is defined as a *function* or *dynamic variable*, information about those definitions might also be printed.

For **apropos-list**, no output occurs as the search proceeds; instead a list of the matching *symbols* is returned when the search is complete.

If *package* is *non-nil*, only the *symbols accessible* in that *package* are searched; otherwise all *symbols accessible* in any *package* are searched.

Because a *symbol* might be available by way of more than one inheritance path, **apropos** might print information about the *same symbol* more than once, or **apropos-list** might return a *list* containing duplicate *symbols*.

Whether or not the search is case-sensitive is *implementation-defined*.

Affected By:

The set of *symbols* which are currently *interned* in any *packages* being searched.

apropos is also affected by ***standard-output***.

describe *Function*

Syntax:

describe *object* `&optional` *stream* → ⟨*no values*⟩

Arguments and Values:

object—an *object*.

stream—an *output stream designator*. The default is *standard output*.

Description:

describe displays information about *object* to *stream*.

For example, **describe** of a *symbol* might show the *symbol*'s value, its definition, and each of its properties. **describe** of a *float* might show the number's internal representation in a way that is useful for tracking down round-off errors. In all cases, however, the nature and format of the output of **describe** is *implementation-dependent*.

describe can describe something that it finds inside the *object*; in such cases, a notational device such as increased indentation or positioning in a table is typically used in order to visually distinguish such recursive descriptions from descriptions of the argument *object*.

The actual act of describing the object is implemented by **describe-object**. **describe** exists as an interface primarily to manage argument defaulting (including conversion of arguments **t** and **nil** into *stream objects*) and to inhibit any return values from **describe-object**.

describe is not intended to be an interactive function. In a *conforming implementation*, **describe** must not, by default, prompt for user input. User-defined methods for **describe-object** are likewise restricted.

Side Effects:

Output to *standard output* or *terminal I/O*.

Affected By:

standard-output and ***terminal-io***, methods on **describe-object** and **print-object** for *objects* having user-defined *classes*.

See Also:

inspect, **describe-object**

describe-object *Standard Generic Function*

Syntax:

describe-object *object stream* → *implementation-dependent*

Method Signatures:

describe-object *(object standard-object) stream*

Arguments and Values:

object—an *object*.

stream—a *stream*.

Description:

The generic function **describe-object** prints a description of *object* to a *stream*. **describe-object** is called by **describe**; it must not be called by the user.

Each implementation is required to provide a *method* on the *class* **standard-object** and *methods* on enough other *classes* so as to ensure that there is always an applicable *method*. Implementations are free to add *methods* for other *classes*. Users can write *methods* for **describe-object** for their own *classes* if they do not wish to inherit an implementation-supplied *method*.

Methods on **describe-object** can recursively call **describe**. Indentation, depth limits, and circularity detection are all taken care of automatically, provided that each *method* handles exactly one level of structure and calls **describe** recursively if there are more structural levels. The consequences are undefined if this rule is not obeyed.

In some implementations the *stream* argument passed to a **describe-object** method is not the original *stream*, but is an intermediate *stream* that implements parts of **describe**. *Methods* should therefore not depend on the identity of this *stream*.

Examples:

```
(defclass spaceship ()
  ((captain :initarg :captain :accessor spaceship-captain)
   (serial# :initarg :serial-number :accessor spaceship-serial-number)))

(defclass federation-starship (spaceship) ())

(defmethod describe-object ((s spaceship) stream)
  (with-slots (captain serial#) s
    (format stream "~&~S is a spaceship of type ~S,~
                    ~%with ~A at the helm ~
                      and with serial number ~D.~%"
            s (type-of s) captain serial#)))

(make-instance 'federation-starship
               :captain "Rachel Garrett"
               :serial-number "NCC-1701-C")
→ #<FEDERATION-STARSHIP 26312465>

(describe *)
▷ #<FEDERATION-STARSHIP 26312465> is a spaceship of type FEDERATION-STARSHIP,
▷ with Rachel Garrett at the helm and with serial number NCC-1701-C.
→ ⟨no values⟩
```

See Also:

describe

Notes:

The same implementation techniques that are applicable to **print-object** are applicable to **describe-object**.

The reason for making the return values for **describe-object** unspecified is to avoid forcing users to include explicit `(values)` in all of their *methods*. **describe** takes care of that.

trace, untrace

trace, untrace *Macro*

Syntax:

trace {*function-name*}* → *trace-result*

untrace {*function-name*}* → *untrace-result*

Arguments and Values:

function-name—a *function name*.

trace-result—*implementation-dependent*, unless no *function-names* are supplied, in which case *trace-result* is a *list* of *function names*.

untrace-result—*implementation-dependent*.

Description:

trace and **untrace** control the invocation of the trace facility.

Invoking **trace** with one or more *function-names* causes the denoted *functions* to be "traced." Whenever a traced *function* is invoked, information about the call, about the arguments passed, and about any eventually returned values is printed to *trace output*. If **trace** is used with no *function-names*, no tracing action is performed; instead, a list of the *functions* currently being traced is returned.

Invoking **untrace** with one or more function names causes those functions to be "untraced" (*i.e.*, no longer traced). If **untrace** is used with no *function-names*, all *functions* currently being traced are untraced.

If a *function* to be traced has been open-coded (*e.g.*, because it was declared **inline**), a call to that *function* might not produce trace output.

Examples:

```
 (defun fact (n) (if (zerop n) 1 (* n (fact (- n 1)))))
→ FACT
 (trace fact)
→ (FACT)
;; Of course, the format of traced output is implementation-dependent.
 (fact 3)
▷ 1 Enter FACT 3
▷ | 2 Enter FACT 2
▷ |   3 Enter FACT 1
▷ |   | 4 Enter FACT 0
▷ |   | 4 Exit FACT 1
▷ |   3 Exit FACT 1
▷ | 2 Exit FACT 2
```

```
▷ 1 Exit FACT 6
→ 6
```

Side Effects:

Might change the definitions of the *functions* named by *function-names*.

Affected By:

Whether the functions named are defined or already being traced.

Exceptional Situations:

Tracing an already traced function, or untracing a function not currently being traced, should produce no harmful effects, but might signal a warning.

See Also:

trace-output, **step**

Notes:

trace and **untrace** may also accept additional *implementation-dependent* argument formats. The format of the trace output is *implementation-dependent*.

Although **trace** can be extended to permit non-standard options, *implementations* are nevertheless encouraged (but not required) to warn about the use of syntax or options that are neither specified by this standard nor added as an extension by the *implementation*, since they could be symptomatic of typographical errors or of reliance on features supported in *implementations* other than the current *implementation*.

step *Macro*

Syntax:

step *form* → {*result*}*

Arguments and Values:

form—a *form*; evaluated as described below.

results—the *values* returned by the *form*.

Description:

step implements a debugging paradigm wherein the programmer is allowed to *step* through the *evaluation* of a *form*. The specific nature of the interaction, including which I/O streams are used and whether the stepping has lexical or dynamic scope, is *implementation-defined*.

step evaluates *form* in the current *environment*. A call to **step** can be compiled, but it is acceptable for an implementation to interactively step through only those parts of the computation that are interpreted.

It is technically permissible for a *conforming implementation* to take no action at all other than normal *execution* of the *form*. In such a situation, `(step` *form*`)` is equivalent to, for example, `(let ()` *form*`)`. In implementations where this is the case, the associated documentation should mention that fact.

See Also:

trace

Notes:

Implementations are encouraged to respond to the typing of `?` or the pressing of a "help key" by providing help including a list of commands.

time *Macro*

Syntax:

time *form* → {*result*}*

Arguments and Values:

form—a *form*; evaluated as described below.

results—the *values* returned by the *form*.

Description:

time evaluates *form* in the current *environment* (lexical and dynamic). A call to **time** can be compiled.

time prints various timing data and other information to *trace output*. The nature and format of the printed information is *implementation-defined*. Implementations are encouraged to provide such information as elapsed real time, machine run time, and storage management statistics.

Affected By:

The accuracy of the results depends, among other things, on the accuracy of the corresponding functions provided by the underlying operating system.

The magnitude of the results may depend on the hardware, the operating system, the lisp implementation, and the state of the global environment. Some specific issues which frequently affect the outcome are hardware speed, nature of the scheduler (if any), number of competing processes (if any), system paging, whether the call is interpreted or compiled, whether functions called are compiled, the kind of garbage collector involved and whether it runs, whether internal data structures (e.g., hash tables) are implicitly reorganized, *etc.*

See Also:

get-internal-real-time, **get-internal-run-time**

Notes:

In general, these timings are not guaranteed to be reliable enough for marketing comparisons. Their value is primarily heuristic, for tuning purposes.

For useful background information on the complicated issues involved in interpreting timing results, see *Performance and Evaluation of Lisp Programs*.

internal-time-units-per-second *Constant Variable*

Constant Value:

A positive *integer*, the magnitude of which is *implementation-dependent*.

Description:

The number of *internal time units* in one second.

See Also:

get-internal-run-time, **get-internal-real-time**

Notes:

These units form the basis of the Internal Time format representation.

get-internal-real-time *Function*

Syntax:

get-internal-real-time ⟨*no arguments*⟩ → *internal-time*

Arguments and Values:

internal-time—a non-negative *integer*.

Description:

get-internal-real-time returns as an *integer* the current time in *internal time units*, relative to an arbitrary time base. The difference between the values of two calls to this function is the amount of elapsed real time (*i.e.*, clock time) between the two calls.

Affected By:

Time of day (*i.e.*, the passage of time). The time base affects the result magnitude.

See Also:

internal-time-units-per-second

get-internal-run-time *Function*

Syntax:

get-internal-run-time ⟨*no arguments*⟩ → *internal-time*

Arguments and Values:

internal-time—a non-negative *integer*.

Description:

Returns as an *integer* the current run time in *internal time units*. The precise meaning of this quantity is *implementation-defined*; it may measure real time, run time, CPU cycles, or some other quantity. The intent is that the difference between the values of two calls to this function be the amount of time between the two calls during which computational effort was expended on behalf of the executing program.

Affected By:

The *implementation*, the time of day (*i.e.*, the passage of time).

See Also:

internal-time-units-per-second

Notes:

Depending on the *implementation*, paging time and garbage collection time might be included in this measurement. Also, in a multitasking environment, it might not be possible to show the time for just the running process, so in some *implementations*, time taken by other processes during the same time interval might be included in this measurement as well.

disassemble *Function*

Syntax:

disassemble *fn* → **nil**

Arguments and Values:

fn—an *extended function designator* or a *lambda expression*.

Description:

The *function* **disassemble** is a debugging aid that composes symbolic instructions or expressions in some *implementation-dependent* language which represent the code used to produce the *function* which is or is named by the argument *fn*. The result is displayed to *standard output* in an *implementation-dependent* format.

If *fn* is a *lambda expression* or *interpreted function*, it is compiled first and the result is disassembled.

If the *fn designator* is a *function name*, the *function* that it *names* is disassembled. (If that *function* is an *interpreted function*, it is first compiled but the result of this implicit compilation is not installed.)

Examples:

```
(defun f (a) (1+ a)) → F
(eq (symbol-function 'f)
    (progn (disassemble 'f)
           (symbol-function 'f))) → true
```

Affected By:

standard-output.

Exceptional Situations:

Should signal an error of *type* **type-error** if *fn* is not an *extended function designator* or a *lambda expression*.

documentation, (setf documentation) *Standard Generic Function*

Syntax:

documentation *x* &optional *doc-type* → *documentation*

documentation, (setf documentation)

(setf documentation) *new-value x* &optional *doc-type* → *new-value*

Method Signatures:

documentation (*x* **function**) &optional *doc-type*

documentation (*x* **list**) &optional *doc-type*

documentation (*x* **method-combination**) &optional *doc-type*

documentation (*x* **package**) &optional *doc-type*

documentation (*x* **standard-class**) &optional *doc-type*

documentation (*x* **standard-method**) &optional *doc-type*

documentation (*x* **structure-class**) &optional *doc-type*

documentation (*x* **symbol**) &optional *doc-type*

(setf documentation) *new-value* (*x* **function**) &optional *doc-type*

(setf documentation) *new-value* (*x* **list**) &optional *doc-type*

(setf documentation) *new-value* (*x* **method-combination**) &optional *doc-type*

(setf documentation) *new-value* (*x* **package**) &optional *doc-type*

(setf documentation) *new-value* (*x* **standard-class**) &optional *doc-type*

(setf documentation) *new-value* (*x* **standard-method**) &optional *doc-type*

(setf documentation) *new-value* (*x* **structure-class**) &optional *doc-type*

(setf documentation) *new-value* (*x* **symbol**) &optional *doc-type*

Arguments and Values:

x—an *object*.

doc-type—If *x* is a *symbol* or a *list*, *doc-type* must be one of **compiler-macro**, **function**, **method-combination**, **setf**, **structure**, **type**, or **variable**; otherwise, *doc-type* must not be supplied.

documentation—a *string*, or **nil**.

new-value—a *string*.

Description:

The *generic function* **documentation** returns the *documentation string* associated with the given *object* if it is available; otherwise it returns **nil**.

The *generic function* `(setf documentation)` updates the *documentation string* associated with *x* to *new-value*. If *x* is a *list*, it must be of the form `(setf` *symbol*`)`.

Documentation strings are made available for debugging purposes. *Conforming programs* are permitted to use *documentation strings* when they are present, but should not depend for their correct behavior on the presence of those *documentation strings*. An *implementation* is permitted to discard *documentation strings* at any time for *implementation-defined* reasons.

For those situations where a *doc-type* argument is permitted, the nature of the *documentation string* returned depends on the *doc-type*, as follows:

compiler-macro

Returns the *documentation string* of the *compiler macro* whose *name* is the *symbol* *x*.

function

Returns the *documentation string* of the *function*, *macro*, or *special operator* whose *name* is the *function name* *x*.

method-combination

Returns the *documentation string* of the *method combination* whose *name* is the *symbol* *x*.

setf

Returns the *documentation string* of the *setf expander* whose *name* is the *symbol* *x*.

structure

Returns the *documentation string* of the *structure name* which is the *symbol* *x*.

type

Returns the *documentation string* of the *class* whose *name* is the *symbol* *x*, if there is such a *class*. Otherwise, it returns the *documentation string* of the *type* which is the *type specifier symbol* *x*.

variable

Returns the *documentation string* of the *dynamic variable* or *constant variable* whose *name* is the *symbol* *x*.

A *conforming implementation* or a *conforming program* may extend the set of *symbols* that are acceptable as the *doc-type*. If *doc-type* is not recognized, an error is signaled.

The remaining *methods* are specialized on the *class* of the argument *x*, as follows:

standard-method

Returns the *documentation string* associated with the *method* *x*.

function

Returns the *documentation string* associated with the *function* *x*.

standard-class or **structure-class**

Returns the *documentation string* associated with the *class* *x*.

package

Returns the *documentation string* associated with the *package* *x*.

Exceptional Situations:

If *x* is a *method object*, a *class object*, a *package object*, a *function object*, or a *method combination object*, the second argument must not be supplied, or else an error of *type* **program-error** is signaled.

If a *symbol* is not recognized as an acceptable *doc-type* argument by the *implementation*, an error of *type* **error** is signaled.

Notes:

This standard prescribes no means to retrieve the *documentation strings* for individual slots specified in a **defclass** form, but *implementations* might still provide debugging tools and/or programming language extensions which manipulate this information. Implementors wishing to provide such support are encouraged to consult the *Metaobject Protocol* for suggestions about how this might be done.

room *Function*

Syntax:

room `&optional` *x* → *implementation-dependent*

Arguments and Values:

x—one of **t**, **nil**, or `:default`.

Description:

room prints, to *standard output*, information about the state of internal storage and its management. This might include descriptions of the amount of memory in use and the degree of memory compaction, possibly broken down by internal data type if that is appropriate. The nature and format of the printed information is *implementation-dependent*. The intent is to provide information that a *programmer* might use to tune a *program* for a particular *implementation*.

`(room nil)` prints out a minimal amount of information. `(room t)` prints out a maximal amount of information. `(room)` or `(room :default)` prints out an intermediate amount of information that is likely to be useful.

Side Effects:

Output to *standard output*.

Affected By:

standard-output.

ed *Function*

Syntax:

`ed &optional` *x* → *implementation-dependent*

Arguments and Values:

x—**nil**, a *pathname*, a *string*, or a *function name*. The default is **nil**.

Description:

ed invokes the editor if the *implementation* provides a resident editor.

If *x* is **nil**, the editor is entered. If the editor had been previously entered, its prior state is resumed, if possible.

If *x* is a *pathname* or *string*, it is taken as the *pathname designator* for a *file* to be edited.

If *x* is a *function name*, the text of its definition is edited. The means by which the function text is obtained is *implementation-defined*.

Exceptional Situations:

The consequences are undefined if the *implementation* does not provide a resident editor.

Might signal **type-error** if its argument is supplied but is not a *symbol*, a *pathname*, or **nil**.

Implementation-dependent additional conditions might be signaled as well.

See Also:

pathname, **logical-pathname**, **compile-file**, **load**

Notes:

Whether **ed** recognizes *logical pathname namestrings* is *implementation-defined*.

inspect *Function*

Syntax:

inspect *object* → *implementation-dependent*

Arguments and Values:

object—an *object*.

Description:

inspect is an interactive version of **describe**. The nature of the interaction is *implementation-dependent*, but the purpose of **inspect** is to make it easy to wander through a data structure, examining and modifying parts of it.

Side Effects:

implementation-dependent.

Affected By:

implementation-dependent.

Exceptional Situations:

implementation-dependent.

See Also:

describe

Notes:

Implementations are encouraged to respond to the typing of `?` or a "help key" by providing help, including a list of commands.

dribble *Function*

Syntax:

dribble &optional *pathname* → *implementation-dependent*

Arguments and Values:

pathname—a *pathname designator*.

Description:

Either *binds* ***standard-input*** and ***standard-output*** or takes other appropriate action, so as to send a record of the input/output interaction to a file named by *pathname*. **dribble** is intended to create a readable record of an interactive session.

If *pathname* is a *logical pathname*, it is translated into a physical pathname as if by calling **translate-logical-pathname**.

(dribble) terminates the recording of input and output and closes the dribble file.

If **dribble** is *called* while a *stream* to a "dribble file" is still open from a previous *call* to **dribble**, the effect is *implementation-defined*. For example, the already-*open stream* might be *closed*, or dribbling might occur both to the old *stream* and to a new one, or the old *stream* might stay open but not receive any further output, or the new request might be ignored, or some other action might be taken.

Affected By:

The *implementation*.

Notes:

Whether **dribble** recognizes *logical pathname namestrings* is *implementation-defined*.

dribble can return before subsequent *forms* are executed. It also can enter a recursive interaction loop, returning only when (dribble) is done.

dribble is intended primarily for interactive debugging; its effect cannot be relied upon when used in a program.

— *Variable*

Value Type:

a *form*.

Initial Value:

implementation-dependent.

Description:

The *value* of - is the *form* that is currently being evaluated by the *Lisp read-eval-print loop*.

Examples:

```
(format t "~&Evaluating ~S~%" -)
▷ Evaluating (FORMAT T "~&Evaluating ~S~%" -)
→ NIL
```

Affected By:

Lisp read-eval-print loop.

See Also:

+ (*variable*), * (*variable*), / (*variable*), Section 25.1.1 (Top level loop)

+, ++, +++ *Variable*

Value Type:

an *object*.

Initial Value:

implementation-dependent.

Description:

The *variables* +, ++, and +++ are maintained by the *Lisp read-eval-print loop* to save *forms* that were recently *evaluated*.

The *value* of + is the last *form* that was *evaluated*, the *value* of ++ is the previous value of +, and the *value* of +++ is the previous value of ++.

Examples:

```
(+ 0 1) → 1
(- 4 2) → 2
(/ 9 3) → 3
(list + ++ +++) → ((/ 9 3) (- 4 2) (+ 0 1))
(setq a 1 b 2 c 3 d (list a b c)) → (1 2 3)
(setq a 4 b 5 c 6 d (list a b c)) → (4 5 6)
(list a b c) → (4 5 6)
(eval +++) → (1 2 3)
#.`(,@++ d) → (1 2 3 (1 2 3))
```

Affected By:

Lisp read-eval-print loop.

See Also:

- (*variable*), * (*variable*), / (*variable*), Section 25.1.1 (Top level loop)

*, **, *** *Variable*

Value Type:

an *object*.

Initial Value:

implementation-dependent.

Description:

The *variables* *, **, and *** are maintained by the *Lisp read-eval-print loop* to save the values of results that are printed each time through the loop.

The *value* of * is the most recent *primary value* that was printed, the *value* of ** is the previous value of *, and the *value* of *** is the previous value of **.

If several values are produced, * contains the first value only; * contains **nil** if zero values are produced.

The *values* of *, **, and *** are updated immediately prior to printing the *return value* of a top-level *form* by the *Lisp read-eval-print loop*. If the *evaluation* of such a *form* is aborted prior to its normal return, the values of *, **, and *** are not updated.

Examples:

```
(values 'a1 'a2) → A1, A2
'b → B
(values 'c1 'c2 'c3) → C1, C2, C3
(list * ** ***) → (C1 B A1)

(defun cube-root (x) (expt x 1/3)) → CUBE-ROOT
(compile *) → CUBE-ROOT
(setq a (cube-root 27.0)) → 3.0
(* * 9.0) → 27.0
```

Affected By:

Lisp read-eval-print loop.

See Also:

- (*variable*), + (*variable*), / (*variable*), Section 25.1.1 (Top level loop)

Notes:

```
*   ≡ (car /)
**  ≡ (car //)
*** ≡ (car ///)
```

/, //, ///   *Variable*

Value Type:

a *proper list*.

Initial Value:

implementation-dependent.

Description:

The *variables* /, //, and /// are maintained by the *Lisp read-eval-print loop* to save the values of results that were printed at the end of the loop.

The *value* of / is a *list* of the most recent *values* that were printed, the *value* of // is the previous value of /, and the *value* of /// is the previous value of //.

The *values* of /, //, and /// are updated immediately prior to printing the *return value* of a top-level *form* by the *Lisp read-eval-print loop*. If the *evaluation* of such a *form* is aborted prior to its normal return, the values of /, //, and /// are not updated.

Examples:

```
(floor 22 7) → 3, 1
(+ (* (car /) 7) (cadr /)) → 22
```

Affected By:

Lisp read-eval-print loop.

See Also:

- (*variable*), + (*variable*), * (*variable*), Section 25.1.1 (Top level loop)

lisp-implementation-type, lisp-implementation-version *Function*

Syntax:

lisp-implementation-type ⟨*no arguments*⟩ → *description*

lisp-implementation-version ⟨*no arguments*⟩ → *description*

Arguments and Values:

description—a *string* or **nil**.

Description:

lisp-implementation-type and **lisp-implementation-version** identify the current implementation of Common Lisp.

lisp-implementation-type returns a *string* that identifies the generic name of the particular Common Lisp implementation.

lisp-implementation-version returns a *string* that identifies the version of the particular Common Lisp implementation.

If no appropriate and relevant result can be produced, **nil** is returned instead of a *string*.

Examples:

```
 (lisp-implementation-type)
→ "ACME Lisp"
or→ "Joe's Common Lisp"
 (lisp-implementation-version)
→ "1.3a"
→ "V2"
or→ "Release 17.3, ECO #6"
```

short-site-name, long-site-name *Function*

Syntax:

short-site-name ⟨*no arguments*⟩ → *description*

long-site-name ⟨*no arguments*⟩ → *description*

Arguments and Values:

description—a *string* or **nil**.

Description:

short-site-name and **long-site-name** return a *string* that identifies the physical location of the computer hardware, or **nil** if no appropriate *description* can be produced.

Examples:

```
 (short-site-name)
→ "MIT AI Lab"
or→ "CMU-CSD"
 (long-site-name)
→ "MIT Artificial Intelligence Laboratory"
or→ "CMU Computer Science Department"
```

Affected By:

The implementation, the location of the computer hardware, and the installation/configuration process.

machine-instance *Function*

Syntax:

machine-instance ⟨*no arguments*⟩ → *description*

Arguments and Values:

description—a *string* or **nil**.

Description:

Returns a *string* that identifies the particular instance of the computer hardware on which Common Lisp is running, or **nil** if no such *string* can be computed.

Examples:

```
 (machine-instance)
→ "ACME.COM"
or→ "S/N 123231"
or→ "18.26.0.179"
or→ "AA-00-04-00-A7-A4"
```

Affected By:

The machine instance, and the *implementation*.

See Also:

machine-type, **machine-version**

machine-type *Function*

Syntax:

machine-type ⟨*no arguments*⟩ → *description*

Arguments and Values:

description—a *string* or **nil**.

Description:

Returns a *string* that identifies the generic name of the computer hardware on which Common Lisp is running.

Examples:

```
 (machine-type)
→ "DEC PDP-10"
or→ "Symbolics LM-2"
```

Affected By:

The machine type. The implementation.

See Also:

machine-version

machine-version *Function*

Syntax:

machine-version ⟨*no arguments*⟩ → *description*

Arguments and Values:

description—a *string* or **nil**.

Description:

Returns a *string* that identifies the version of the computer hardware on which Common Lisp is running, or **nil** if no such value can be computed.

Examples:

```
(machine-version) → "KL-10, microcode 9"
```

Affected By:

The machine version, and the *implementation*.

See Also:

machine-type, **machine-instance**

software-type, software-version *Function*

Syntax:

software-type ⟨*no arguments*⟩ → *description*

software-version ⟨*no arguments*⟩ → *description*

Arguments and Values:

description—a *string* or **nil**.

Description:

software-type returns a *string* that identifies the generic name of any relevant supporting software, or **nil** if no appropriate or relevant result can be produced.

software-version returns a *string* that identifies the version of any relevant supporting software, or **nil** if no appropriate or relevant result can be produced.

Examples:

```
(software-type) → "Multics"
(software-version) → "1.3x"
```

Affected By:

Operating system environment.

Notes:

This information should be of use to maintainers of the *implementation*.

user-homedir-pathname *Function*

Syntax:

user-homedir-pathname `&optional` *host* → *pathname*

Arguments and Values:

host—a *string*, a *list* of *strings*, or `:unspecific`.

pathname—a *pathname*, or **nil**.

Description:

user-homedir-pathname determines the *pathname* that corresponds to the user's home directory on *host*. If *host* is not supplied, its value is *implementation-dependent*. For a description of `:unspecific`, see Section 19.2.1 (Pathname Components).

The definition of home directory is *implementation-dependent*, but defined in Common Lisp to mean the directory where the user keeps personal files such as initialization files and mail.

user-homedir-pathname returns a *pathname* without any name, type, or version component (those components are all **nil**) for the user's home directory on *host*.

If it is impossible to determine the user's home directory on *host*, then **nil** is returned. **user-homedir-pathname** never returns **nil** if *host* is not supplied.

Examples:

```
(pathnamep (user-homedir-pathname)) → true
```

Affected By:

The host computer's file system, and the *implementation*.

Table of Contents