Programming Language—Common Lisp

26. Glossary

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

26.1 Glossary

Each entry in this glossary has the following parts:

- the term being defined, set in boldface.
- optional pronunciation, enclosed in square brackets and set in boldface, as in the following example: [**ˈaˌlist**]. The pronunciation key follows *Webster's Third New International Dictionary the English Language, Unabridged*, except that "ε" is used to notate the schwa (upside-down "e") character.
- the part or parts of speech, set in italics. If a term can be used as several parts of speech, there is a separate definition for each part of speech.
- one or more definitions, organized as follows:
  - an optional number, present if there are several definitions. Lowercase letters might also be used in cases where subdefinitions of a numbered definition are necessary.
  - an optional part of speech, set in italics, present if the term is one of several parts of speech.
  - an optional discipline, set in italics, present if the term has a standard definition being repeated. For example, "*Math.*"
  - an optional context, present if this definition is meaningful only in that context. For example, "(of a *symbol*)".
  - the definition.
  - an optional example sentence. For example, "This is an example of an example."
  - optional cross references.

In addition, some terms have idiomatic usage in the Common Lisp community which is not shared by other communities, or which is not technically correct. Definitions labeled "*Idiom.*" represent such idiomatic usage; these definitions are sometimes followed by an explanatory note.

Words in *this font* are words with entries in the glossary. Words in example sentences do not follow this convention.

When an ambiguity arises, the longest matching substring has precedence. For example, "*complex float*" refers to a single glossary entry for "*complex float*" rather than the combined meaning of the glossary terms "*complex*" and "*float*."

Subscript notation, as in "$something_n$" means that the *n*th definition of "*something*" is intended. This notation is used only in situations where the context might be insufficient to disambiguate.

The following are abbreviations used in the glossary:

Abbreviation	Meaning
adj.	adjective
adv.	adverb
ANSI	compatible with one or more ANSI standards
Comp.	computers
Idiom.	idiomatic
IEEE	compatible with one or more IEEE standards
ISO	compatible with one or more ISO standards
Math.	mathematics
Trad.	traditional
n.	noun
v.	verb
v.t.	transitive verb

Non-alphabetic

() [**ˈnil**], *n.* an alternative notation for writing the symbol **nil**, used to emphasize the use of *nil* as an *empty list*.

A

absolute *adj.* 1. (of a *time*) representing a specific point in time. 2. (of a *pathname*) representing a specific position in a directory hierarchy. See *relative*.

access *n.*, *v.t.* 1. *v.t.* (a *place*, or *array*) to *read*$_1$ or *write*$_1$ the *value* of the *place* or an *element* of the *array*. 2. *n.* (of a *place*) an attempt to *access*$_1$ the *value* of the *place*.

accessibility *n.* the state of being *accessible*.

accessible *adj.* 1. (of an *object*) capable of being *referenced*. 2. (of *shared slots* or *local slots* in an *instance* of a *class*) having been defined by the *class* of the *instance* or *inherited* from a *superclass* of that *class*. 3. (of a *symbol* in a *package*) capable of being *referenced* without a *package prefix* when that *package* is current, regardless of whether the *symbol* is *present* in that *package* or is *inherited*.

accessor *n.* an *operator* that performs an *access*. See *reader* and *writer*.

active *adj.* 1. (of a *handler*, a *restart*, or a *catch tag*) having been *established* but not yet *disestablished*. 2. (of an *element* of an *array*) having an index that is greater than or equal to zero, but less than the *fill pointer* (if any). For an *array* that has no *fill pointer*, all *elements* are considered *active*.

actual adjustability *n.* (of an *array*) a *generalized boolean* that is associated with the *array*, representing whether the *array* is *actually adjustable*. See also *expressed adjustability* and **adjustable-array-p**.

actual argument *n. Trad.* an *argument*.

actual array element type *n.* (of an *array*) the *type* for which the *array* is actually specialized, which is the *upgraded array element type* of the *expressed array element type* of the *array*. See the *function* **array-element-type**.

actual complex part type *n.* (of a *complex*) the *type* in which the real and imaginary parts of the *complex* are actually represented, which is the *upgraded complex part type* of the *expressed complex part type* of the *complex*.

actual parameter *n. Trad.* an *argument*.

actually adjustable *adj.* (of an *array*) such that **adjust-array** can adjust its characteristics by direct modification. A *conforming program* may depend on an *array* being *actually adjustable* only if either that *array* is known to have been *expressly adjustable* or if that *array* has been explicitly tested by **adjustable-array-p**.

adjustability *n.* (of an *array*) 1. *expressed adjustability*. 2. *actual adjustability*.

adjustable *adj.* (of an *array*) 1. *expressly adjustable*. 2. *actually adjustable*.

after method *n.* a *method* having the *qualifier* `:after`.

alist [ˈā,list], *n.* an *association list*.

alphabetic *n.*, *adj.* 1. *adj.* (of a *character*) being one of the *standard characters* `A` through `Z` or `a` through `z`, or being any *implementation-defined* character that has *case*, or being some other *graphic character* defined by the *implementation* to be *alphabetic*$_1$. 2. a. *n.* one of several possible *constituent traits* of a *character*. For details, see Section 2.1.4.1 (Constituent Characters) and Section 2.2 (Reader Algorithm). b. *adj.* (of a *character*) being a *character* that has *syntax type constituent* in the *current readtable* and that has the *constituent trait alphabetic*$_{2a}$. See Figure 2–8.

alphanumeric *adj.* (of a *character*) being either an *alphabetic*$_1$ *character* or a *numeric* character.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

ampersand *n.* the *standard character* that is called "ampersand" (`&`). See Figure 2–5.

anonymous *adj.* 1. (of a *class* or *function*) having no *name* 2. (of a *restart*) having a *name* of **nil**.

apparently uninterned *adj.* having a *home package* of **nil**. (An *apparently uninterned symbol* might or might not be an *uninterned symbol*. *Uninterned symbols* have a *home package* of **nil**, but *symbols* which have been *uninterned* from their *home package* also have a *home package* of **nil**, even though they might still be *interned* in some other *package*.)

applicable *adj.* 1. (of a *handler*) being an *applicable handler*. 2. (of a *method*) being an *applicable method*. 3. (of a *restart*) being an *applicable restart*.

applicable handler *n.* (for a *condition* being *signaled*) an *active handler* for which the associated type contains the *condition*.

applicable method *n.* (of a *generic function* called with *arguments*) a *method* of the *generic function* for which the *arguments* satisfy the *parameter specializers* of that *method*. See Section 7.6.6.1.1 (Selecting the Applicable Methods).

applicable restart *n.* 1. (for a *condition*) an *active handler* for which the associated test returns *true* when given the *condition* as an argument. 2. (for no particular *condition*) an *active handler* for which the associated test returns *true* when given **nil** as an argument.

apply *v.t.* (a *function* to a *list*) to *call* the *function* with arguments that are the *elements* of the *list*. "Applying the function **+** to a list of integers returns the sum of the elements of that list."

argument *n.* 1. (of a *function*) an *object* which is offered as data to the *function* when it is *called*. 2. (of a *format control*) a *format argument*.

argument evaluation order *n.* the order in which *arguments* are evaluated in a function call. "The argument evaluation order for Common Lisp is left to right." See Section 3.1 (Evaluation).

argument precedence order *n.* the order in which the *arguments* to a *generic function* are considered when sorting the *applicable methods* into precedence order.

around method *n.* a *method* having the *qualifier* `:around`.

array *n.* an *object* of *type* **array**, which serves as a container for other *objects* arranged in a Cartesian coordinate system.

array element type *n.* (of an *array*) 1. a *type* associated with the *array*, and of which all *elements* of the *array* are constrained to be members. 2. the *actual array element type* of the *array*. 3. the *expressed array element type* of the *array*.

array total size *n.* the total number of *elements* in an *array*, computed by taking the product of the *dimensions* of the *array*. (The size of a zero-dimensional *array* is therefore one.)

assign *v.t.* (a *variable*) to change the *value* of the *variable* in a *binding* that has already been *established*. See the *special operator* **setq**.

association list *n.* a *list* of *conses* representing an association of *keys* with *values*, where the *car* of each *cons* is the *key* and the *cdr* is the *value* associated with that *key*.

asterisk *n.* the *standard character* that is variously called "asterisk" or "star" (`*`). See Figure 2–5.

at-sign *n.* the *standard character* that is variously called "commercial at" or "at sign" (`@`). See Figure 2–5.

atom *n.* any *object* that is not a *cons*. "A vector is an atom."

atomic *adj.* being an *atom*. "The number 3, the symbol `foo`, and **nil** are atomic."

atomic type specifier *n.* a *type specifier* that is *atomic*. For every *atomic type specifier*, *x*, there is an equivalent *compound type specifier* with no arguments supplied, `(`*x*`)`.

attribute *n.* (of a *character*) a program-visible aspect of the *character*. The only *standardized attribute* of a *character* is its *code*$_2$, but *implementations* are permitted to have additional *implementation-defined attributes*. See Section 13.1.3 (Character Attributes). "An implementation that support fonts might make font information an attribute of a character, while others might represent font information separately from characters."

aux variable *n.* a *variable* that occurs in the part of a *lambda list* that was introduced by **&aux**. Unlike all other *variables* introduced by a *lambda-list*, *aux variables* are not *parameters*.

auxiliary method *n.* a member of one of two sets of *methods* (the set of *primary methods* is the other) that form an exhaustive partition of the set of *methods* on the *method*'s *generic function*. How these sets are determined is dependent on the *method combination* type; see Section 7.6.2 (Introduction to Methods).

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

B

backquote *n.* the *standard character* that is variously called "grave accent" or "backquote" (`` ` ``). See Figure 2–5.

backslash *n.* the *standard character* that is variously called "reverse solidus" or "backslash" (`\`). See Figure 2–5.

base character *n.* a *character* of *type* **base-char**.

base string *n.* a *string* of *type* **base-string**.

before method *n.* a *method* having the *qualifier* `:before`.

bidirectional *adj.* (of a *stream*) being both an *input stream* and an *output stream*.

binary *adj.* 1. (of a *stream*) being a *stream* that has an *element type* that is a *subtype* of *type* **integer**. The most fundamental operation on a *binary input stream* is **read-byte** and on a *binary output stream* is **write-byte**. See *character*. 2. (of a *file*) having been created by opening a *binary stream*. (It is *implementation-dependent* whether this is an detectable aspect of the *file*, or whether any given *character file* can be treated as a *binary file*.)

bind *v.t.* (a *variable*) to establish a *binding* for the *variable*.

binding *n.* an association between a *name* and that which the *name* denotes. "A lexical binding is a lexical association between a name and its value." When the term *binding* is qualified by the name of a *namespace*, such as "variable" or "function," it restricts the binding to the indicated namespace, as in: "**let** establishes variable bindings." or "**let** establishes bindings of variables."

bit *n.* an *object* of *type* **bit**; that is, the *integer* `0` or the *integer* `1`.

bit array *n.* a specialized *array* that is of *type* `(array bit)`, and whose elements are of *type* **bit**.

bit vector *n.* a specialized *vector* that is of *type* **bit-vector**, and whose elements are of *type* **bit**.

bit-wise logical operation specifier *n.* an *object* which names one of the sixteen possible bit-wise logical operations that can be performed by the **boole** function, and which is the *value* of exactly one of the *constant variables* **boole-clr**, **boole-set**, **boole-1**, **boole-2**, **boole-c1**, **boole-c2**, **boole-and**, **boole-ior**, **boole-xor**, **boole-eqv**, **boole-nand**, **boole-nor**, **boole-andc1**, **boole-andc2**, **boole-orc1**, or **boole-orc2**.

block *n.* a named lexical *exit point*, *established* explicitly by **block** or implicitly by *operators* such as **loop**, **do** and **prog**, to which control and values may be transfered by using a **return-from** *form* with the name of the *block*.

block tag *n.* the *symbol* that, within the *lexical scope* of a **block** *form*, names the *block established* by that **block** *form*. See **return** or **return-from**.

boa lambda list *n.* a *lambda list* that is syntactically like an *ordinary lambda list*, but that is processed in "**b**y **o**rder of **a**rgument" style. See Section 3.4.6 (Boa Lambda Lists).

body parameter *n.* a *parameter* available in certain *lambda lists* which from the point of view of *conforming programs* is like a *rest parameter* in every way except that it is introduced by **&body** instead of **&rest**. (*Implementations* are permitted to provide extensions which distinguish *body parameters* and *rest parameters*—*e.g.*, the *forms* for *operators* which were defined using a *body parameter* might be pretty printed slightly differently than *forms* for *operators* which were defined using *rest parameters*.)

boolean *n.* an *object* of *type* **boolean**; that is, one of the following *objects*: the symbol **t** (representing *true*), or the symbol **nil** (representing *false*). See *generalized boolean*.

boolean equivalent *n.* (of an *object* O_1) any *object* O_2 that has the same truth value as O_1 when both O_1 and O_2 are viewed as *generalized booleans*.

bound *adj.*, *v.t.* 1. *adj.* having an associated denotation in a *binding*. "The variables named by a **let** are bound within its body." See *unbound*. 2. *adj.* having a local *binding* which *shadows*$_2$ another. "The variable ***print-escape*** is bound while in the **princ** function." 3. *v.t.* the past tense of *bind*.

bound declaration *n.* a *declaration* that refers to or is associated with a *variable* or *function* and that appears within the *special form* that *establishes* the *variable* or *function*, but before the body of that *special form* (specifically, at the head of that *form*'s body). (If a *bound declaration* refers to a *function binding* or a *lexical variable binding*, the *scope* of the *declaration* is exactly the *scope* of that *binding*. If the *declaration* refers to a *dynamic variable binding*, the *scope* of the *declaration* is what the *scope* of the *binding* would have been if it were lexical rather than dynamic.)

bounded *adj.* (of a *sequence* S, by an ordered pair of *bounding indices* i_{start} and i_{end}) restricted to a subrange of the *elements* of S that includes each *element* beginning with (and including) the one indexed by i_{start} and continuing up to (but not including) the one indexed by i_{end}.

bounding index *n.* (of a *sequence* with *length* n) either of a conceptual pair of *integers*, i_{start} and i_{end}, respectively called the "lower bounding index" and "upper

bounding index", such that $0 \leq i_{start} \leq i_{end} \leq n$, and which therefore delimit a subrange of the *sequence bounded* by i_{start} and i_{end}.

bounding index designator (for a *sequence*) one of two *objects* that, taken together as an ordered pair, behave as a *designator* for *bounding indices* of the *sequence*; that is, they denote *bounding indices* of the *sequence*, and are either: an *integer* (denoting itself) and **nil** (denoting the *length* of the *sequence*), or two *integers* (each denoting themselves).

break loop *n.* A variant of the normal *Lisp read-eval-print loop* that is recursively entered, usually because the ongoing *evaluation* of some other *form* has been suspended for the purpose of debugging. Often, a *break loop* provides the ability to exit in such a way as to continue the suspended computation. See the *function* **break**.

broadcast stream *n.* an *output stream* of *type* **broadcast-stream**.

built-in class *n.* a *class* that is a *generalized instance* of *class* **built-in-class**.

built-in type *n.* one of the *types* in Figure 4–2.

byte *n.* 1. adjacent bits within an *integer*. (The specific number of bits can vary from point to point in the program; see the *function* **byte**.) 2. an integer in a specified range. (The specific range can vary from point to point in the program; see the *functions* **open** and **write-byte**.)

byte specifier *n.* An *object* of *implementation-dependent* nature that is returned by the *function* **byte** and that specifies the range of bits in an *integer* to be used as a *byte* by *functions* such as **ldb**.

C

cadr [**ˈkaˌ**dər], *n.* (of an *object*) the *car* of the *cdr* of that *object*.

call *v.t.*, *n.* 1. *v.t.* (a *function* with *arguments*) to cause the *code* represented by that *function* to be *executed* in an *environment* where *bindings* for the *values* of its *parameters* have been *established* based on the *arguments*. "Calling the function **+** with the arguments `5` and `1` yields a value of `6`." 2. *n.* a *situation* in which a *function* is called.

captured initialization form *n.* an *initialization form* along with the *lexical environment* in which the *form* that defined the *initialization form* was *evaluated*. "Each newly added shared slot is set to the result of evaluating the captured initialization form for the slot that was specified in the **defclass** form for the new class."

car *n.* 1. a. (of a *cons*) the component of a *cons* corresponding to the first *argument* to **cons**; the other component is the *cdr*. "The function **rplaca** modifies the car of a cons." b. (of a *list*) the first *element* of the *list*, or **nil** if the *list* is the *empty list*. 2. the *object* that is held in the car_1. "The function **car** returns the car of a cons."

case *n.* (of a *character*) the property of being either *uppercase* or *lowercase*. Not all *characters* have *case*. "The characters `#\A` and `#\a` have case, but the character `#\$` has no case." See Section 13.1.4.3 (Characters With Case) and the *function* **both-case-p**.

case sensitivity mode *n.* one of the *symbols* `:upcase`, `:downcase`, `:preserve`, or `:invert`.

catch *n.* an *exit point* which is *established* by a **catch** *form* within the *dynamic scope* of its body, which is named by a *catch tag*, and to which control and *values* may be *thrown*.

catch tag *n.* an *object* which names an *active catch*. (If more than one *catch* is active with the same *catch tag*, it is only possible to *throw* to the innermost such *catch* because the outer one is $shadowed_2$.)

cddr [**ˈkůdɛˌdɛr**] or [**ˈkɛˌdůdɛr**], *n.* (of an *object*) the *cdr* of the *cdr* of that *object*.

cdr [**ˈkůˌdɛr**], *n.* 1. a. (of a *cons*) the component of a *cons* corresponding to the second *argument* to **cons**; the other component is the *car*. "The function **rplacd** modifies the cdr of a cons." b. (of a *list* L_1) either the *list* L_2 that contains the *elements* of L_1 that follow after the first, or else **nil** if L_1 is the *empty list*. 2. the *object* that is held in the cdr_1. "The function **cdr** returns the cdr of a cons."

cell *n. Trad.* (of an *object*) a conceptual *slot* of that *object*. The *dynamic variable* and global *function bindings* of a *symbol* are sometimes referred to as its *value cell* and *function cell*, respectively.

character *n.*, *adj.* 1. *n.* an *object* of *type* **character**; that is, an *object* that represents a unitary token in an aggregate quantity of text; see Section 13.1 (Character Concepts). 2. *adj.* a. (of a *stream*) having an *element type* that is a *subtype* of *type* **character**. The most fundamental operation on a *character input stream* is **read-char** and on a *character output stream* is **write-char**. See *binary*. b. (of a *file*) having been created by opening a *character stream*. (It is *implementation-dependent* whether this is an inspectable aspect of the *file*, or whether any given *binary file* can be treated as a *character file*.)

character code *n.* 1. one of possibly several *attributes* of a *character*. 2. a non-negative *integer* less than the *value* of **char-code-limit** that is suitable for use as a $character\ code_1$.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

character designator *n.* a *designator* for a *character*; that is, an *object* that denotes a *character* and that is one of: a *designator* for a *string* of *length* one (denoting the *character* that is its only *element*), or a *character* (denoting itself).

circular *adj.* 1. (of a *list*) a *circular list*. 2. (of an arbitrary *object*) having a *component*, *element*, $constituent_2$, or *subexpression* (as appropriate to the context) that is the *object* itself.

circular list *n.* a chain of *conses* that has no termination because some *cons* in the chain is the *cdr* of a later *cons*.

class *n.* 1. an *object* that uniquely determines the structure and behavior of a set of other *objects* called its *direct instances*, that contributes structure and behavior to a set of other *objects* called its *indirect instances*, and that acts as a *type specifier* for a set of objects called its *generalized instances*. "The class **integer** is a subclass of the class **number**." (Note that the phrase "the *class* `foo`" is often substituted for the more precise phrase "the *class* named `foo`"—in both cases, a *class object* (not a *symbol*) is denoted.) 2. (of an *object*) the uniquely determined *class* of which the *object* is a *direct instance*. See the *function* **class-of**. "The class of the object returned by **gensym** is **symbol**." (Note that with this usage a phrase such as "its *class* is `foo`" is often substituted for the more precise phrase "its *class* is the *class* named `foo`"—in both cases, a *class object* (not a *symbol*) is denoted.)

class designator *n.* a *designator* for a *class*; that is, an *object* that denotes a *class* and that is one of: a *symbol* (denoting the *class* named by that *symbol*; see the *function* **find-class**) or a *class* (denoting itself).

class precedence list *n.* a unique total ordering on a *class* and its *superclasses* that is consistent with the *local precedence orders* for the *class* and its *superclasses*. For detailed information, see Section 4.3.5 (Determining the Class Precedence List).

close *v.t.* (a *stream*) to terminate usage of the *stream* as a source or sink of data, permitting the *implementation* to reclaim its internal data structures, and to free any external resources which might have been locked by the *stream* when it was opened.

closed *adj.* (of a *stream*) having been *closed* (see close). Some (but not all) operations that are valid on *open streams* are not valid on *closed streams*. See Section 21.1.1.1.2 (Open and Closed Streams).

closure *n.* a *lexical closure*.

coalesce *v.t.* (*literal objects* that are *similar*) to consolidate the identity of those *objects*, such that they become the *same object*. See Section 3.2.1 (Compiler Terminology).

code *n.* 1. *Trad.* any representation of actions to be performed, whether conceptual or as an actual *object*, such as *forms*, *lambda expressions*, *objects* of *type function*, text in a *source file*, or instruction sequences in a *compiled file*. This is a generic term; the specific nature of the representation depends on its context. 2. (of a *character*) a *character code*.

coerce *v.t.* (an *object* to a *type*) to produce an *object* from the given *object*, without modifying that *object*, by following some set of coercion rules that must be specifically stated for any context in which this term is used. The resulting *object* is necessarily of the indicated *type*, except when that type is a *subtype* of *type* **complex**; in that case, if a *complex rational* with an imaginary part of zero would result, the result is a *rational* rather than a *complex*—see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

colon *n.* the *standard character* that is called "colon" (:). See Figure 2–5.

comma *n.* the *standard character* that is called "comma" (,). See Figure 2–5.

compilation *n.* the process of *compiling code* by the *compiler*.

compilation environment *n.* 1. An *environment* that represents information known by the *compiler* about a *form* that is being *compiled*. See Section 3.2.1 (Compiler Terminology). 2. An *object* that represents the *compilation* $environment_1$ and that is used as a second argument to a *macro function* (which supplies a *value* for any **&environment** *parameter* in the *macro function*'s definition).

compilation unit *n.* an interval during which a single unit of compilation is occurring. See the *macro* **with-compilation-unit**.

compile *v.t.* 1. (*code*) to perform semantic preprocessing of the *code*, usually optimizing one or more qualities of the code, such as run-time speed of *execution* or run-time storage usage. The minimum semantic requirements of compilation are that it must remove all macro calls and arrange for all *load time values* to be resolved prior to run time. 2. (a *function*) to produce a new *object* of *type* **compiled-function** which represents the result of *compiling* the *code* represented by the *function*. See the *function* **compile**. 3. (a *source file*) to produce a *compiled file* from a *source file*. See the *function* **compile-file**.

compile time *n.* the duration of time that the *compiler* is processing *source code*.

compile-time definition *n.* a definition in the *compilation environment*.

compiled code *n.* 1. *compiled functions*. 2. *code* that represents *compiled functions*, such as the contents of a *compiled file*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

compiled file *n.* a *file* which represents the results of *compiling* the *forms* which appeared in a corresponding *source file*, and which can be *loaded*. See the *function* **compile-file**.

compiled function *n.* an *object* of *type* **compiled-function**, which is a *function* that has been *compiled*, which contains no references to *macros* that must be expanded at run time, and which contains no unresolved references to *load time values*.

compiler *n.* a facility that is part of Lisp and that translates *code* into an *implementation-dependent* form that might be represented or *executed* efficiently. The functions **compile** and **compile-file** permit programs to invoke the *compiler*.

compiler macro *n.* an auxiliary macro definition for a globally defined *function* or *macro* which might or might not be called by any given *conforming implementation* and which must preserve the semantics of the globally defined *function* or *macro* but which might perform some additional optimizations. (Unlike a *macro*, a *compiler macro* does not extend the syntax of Common Lisp; rather, it provides an alternate implementation strategy for some existing syntax or functionality.)

compiler macro expansion *n.* 1. the process of translating a *form* into another *form* by a *compiler macro*. 2. the *form* resulting from this process.

compiler macro form *n.* a *function form* or *macro form* whose *operator* has a definition as a *compiler macro*, or a **funcall** *form* whose first *argument* is a **function** *form* whose *argument* is the *name* of a *function* that has a definition as a *compiler macro*.

compiler macro function *n.* a *function* of two arguments, a *form* and an *environment*, that implements *compiler macro expansion* by producing either a *form* to be used in place of the original argument *form* or else **nil**, indicating that the original *form* should not be replaced. See Section 3.2.2.1 (Compiler Macros).

complex *n.* an *object* of *type* **complex**.

complex float *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **float**. A *complex float* is a *complex*, but it is not a *float*.

complex part type *n.* (of a *complex*) 1. the *type* which is used to represent both the real part and the imaginary part of the *complex*. 2. the *actual complex part type* of the *complex*. 3. the *expressed complex part type* of the *complex*.

complex rational *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **rational**. A *complex rational* is a *complex*, but it is not a *rational*. No *complex rational* has an imaginary part of zero because such a number is always represented by Common Lisp as an *object* of *type* **rational**; see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

complex single float *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **single-float**. A *complex single float* is a *complex*, but it is not a *single float*.

composite stream *n.* a *stream* that is composed of one or more other *streams*. "**make-synonym-stream** creates a composite stream."

compound form *n.* a *non-empty list* which is a *form*: a *special form*, a *lambda form*, a *macro form*, or a *function form*.

compound type specifier *n.* a *type specifier* that is a *cons*; *i.e.*, a *type specifier* that is not an *atomic type specifier*. "`(vector single-float)` is a compound type specifier."

concatenated stream *n.* an *input stream* of *type* **concatenated-stream**.

condition *n.* 1. an *object* which represents a *situation*—usually, but not necessarily, during *signaling*. 2. an *object* of *type* **condition**.

condition designator *n.* one or more *objects* that, taken together, denote either an existing *condition object* or a *condition object* to be implicitly created. For details, see Section 9.1.2.1 (Condition Designators).

condition handler *n.* a *function* that might be invoked by the act of *signaling*, that receives the *condition* being signaled as its only argument, and that is permitted to *handle* the *condition* or to *decline*. See Section 9.1.4.1 (Signaling).

condition reporter *n.* a *function* that describes how a *condition* is to be printed when the *Lisp printer* is invoked while ***print-escape*** is *false*. See Section 9.1.3 (Printing Conditions).

conditional newline *n.* a point in output where a *newline* might be inserted at the discretion of the *pretty printer*. There are four kinds of *conditional newlines*, called "linear-style," "fill-style," "miser-style," and "mandatory-style." See the *function* **pprint-newline** and Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

conformance *n.* a state achieved by proper and complete adherence to the requirements of this specification. See Section 1.5 (Conformance).

conforming code *n.* *code* that is all of part of a *conforming program*.

conforming implementation *n.* an *implementation*, used to emphasize complete and correct adherance to all conformance criteria. A *conforming implementation* is

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

capable of accepting a *conforming program* as input, preparing that *program* for *execution*, and executing the prepared *program* in accordance with this specification. An *implementation* which has been extended may still be a *conforming implementation* provided that no extension interferes with the correct function of any *conforming program*.

conforming processor *n.* *ANSI* a *conforming implementation*.

conforming program *n.* a *program*, used to emphasize the fact that the *program* depends for its correctness only upon documented aspects of Common Lisp, and can therefore be expected to run correctly in any *conforming implementation*.

congruent *n.* conforming to the rules of *lambda list* congruency, as detailed in Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

cons *n.v.* 1. *n.* a compound data *object* having two components called the *car* and the *cdr*. 2. *v.* to create such an *object*. 3. *v. Idiom.* to create any *object*, or to allocate storage.

constant *n.* 1. a *constant form*. 2. a *constant variable*. 3. a *constant object*. 4. a *self-evaluating object*.

constant form *n.* any *form* for which *evaluation* always *yields* the same *value*, that neither affects nor is affected by the *environment* in which it is *evaluated* (except that it is permitted to refer to the names of *constant variables* defined in the *environment*), and that neither affects nor is affected by the state of any *object* except those *objects* that are *otherwise inaccessible parts* of *objects* created by the *form* itself. "A **car** form in which the argument is a **quote** form is a constant form."

constant object *n.* an *object* that is constrained (*e.g.*, by its context in a *program* or by the source from which it was obtained) to be *immutable*. "A literal object that has been processed by **compile-file** is a constant object."

constant variable *n.* a *variable*, the *value* of which can never change; that is, a *keyword*$_1$ or a *named constant*. "The symbols **t**, **nil**, `:direction`, and **most-positive-fixnum** are constant variables."

constituent *n.*, *adj.* 1. a. *n.* the *syntax type* of a *character* that is part of a *token*. For details, see Section 2.1.4.1 (Constituent Characters). b. *adj.* (of a *character*) having the *constituent*$_{1a}$ *syntax type*$_2$. c. *n.* a *constituent*$_{1b}$ *character*. 2. *n.* (of a *composite stream*) one of possibly several *objects* that collectively comprise the source or sink of that *stream*.

constituent trait *n.* (of a *character*) one of several classifications of a *constituent character* in a *readtable*. See Section 2.1.4.1 (Constituent Characters).

constructed stream *n.* a *stream* whose source or sink is a Lisp *object*. Note that since a *stream* is another Lisp *object*, *composite streams* are considered *constructed streams*. "A string stream is a constructed stream."

contagion *n.* a process whereby operations on *objects* of differing *types* (*e.g.*, arithmetic on mixed *types* of *numbers*) produce a result whose *type* is controlled by the dominance of one *argument*'s *type* over the *types* of the other *arguments*. See Section 12.1.1.2 (Contagion in Numeric Operations).

continuable *n.* (of an *error*) an *error* that is *correctable* by the `continue` restart.

control form *n.* 1. a *form* that establishes one or more places to which control can be transferred. 2. a *form* that transfers control.

copy *n.* 1. (of a *cons* C) a *fresh cons* with the *same car* and *cdr* as C. 2. (of a *list* L) a *fresh list* with the *same elements* as L. (Only the *list structure* is *fresh*; the *elements* are the *same*.) See the *function* **copy-list**. 3. (of an *association list* A with *elements* A_i) a *fresh list* B with *elements* B_i, each of which is **nil** if A_i is **nil**, or else a *copy* of the *cons* A_i. See the *function* **copy-alist**. 4. (of a *tree* T) a *fresh tree* with the *same leaves* as T. See the *function* **copy-tree**. 5. (of a *random state* R) a *fresh random state* that, if used as an argument to to the *function* **random** would produce the same series of "random" values as R would produce. 6. (of a *structure* S) a *fresh structure* that has the same *type* as S, and that has slot values, each of which is the *same* as the corresponding slot value of S. (Note that since the difference between a *cons*, a *list*, and a *tree* is a matter of "view" or "intention," there can be no general-purpose *function* which, based solely on the *type* of an *object*, can determine which of these distinct meanings is intended. The distinction rests solely on the basis of the text description within this document. For example, phrases like "a *copy* of the given *list*" or "copy of the *list* `x`" imply the second definition.)

correctable *adj.* (of an *error*) 1. (by a *restart* other than **abort** that has been associated with the *error*) capable of being corrected by invoking that *restart*. "The function **cerror** signals an error that is correctable by the **continue** *restart*." (Note that correctability is not a property of an *error object*, but rather a property of the *dynamic environment* that is in effect when the *error* is *signaled*. Specifically, the *restart* is "associated with" the *error condition object*. See Section 9.1.4.2.4 (Associating a Restart with a Condition).) 2. (when no specific *restart* is mentioned) $correctable_1$ by at least one *restart*. "**import** signals a correctable error of *type* **package-error** if any of the imported symbols has the same name as some distinct symbol already accessible in the package."

current input base *n.* (in a *dynamic environment*) the *radix* that is the *value* of ***read-base*** in that *environment*, and that is the default *radix* employed by the *Lisp reader* and its related *functions*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

current logical block *n.* the context of the innermost lexically enclosing use of **pprint-logical-block**.

current output base *n.* (in a *dynamic environment*) the *radix* that is the *value* of ***print-base*** in that *environment*, and that is the default *radix* employed by the *Lisp printer* and its related *functions*.

current package *n.* (in a *dynamic environment*) the *package* that is the *value* of ***package*** in that *environment*, and that is the default *package* employed by the *Lisp reader* and *Lisp printer*, and their related *functions*.

current pprint dispatch table *n.* (in a *dynamic environment*) the *pprint dispatch table* that is the *value* of ***print-pprint-dispatch*** in that *environment*, and that is the default *pprint dispatch table* employed by the *pretty printer*.

current random state *n.* (in a *dynamic environment*) the *random state* that is the *value* of ***random-state*** in that *environment*, and that is the default *random state* employed by **random**.

current readtable *n.* (in a *dynamic environment*) the *readtable* that is the *value* of ***readtable*** in that *environment*, and that affects the way in which $expressions_2$ are parsed into *objects* by the *Lisp reader*.

D

data type *n.* *Trad.* a *type*.

debug I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***debug-io***.

debugger *n.* a facility that allows the *user* to handle a *condition* interactively. For example, the *debugger* might permit interactive selection of a *restart* from among the *active restarts*, and it might perform additional *implementation-defined* services for the purposes of debugging.

declaration *n.* a *global declaration* or *local declaration*.

declaration identifier *n.* one of the *symbols* **declaration**, **dynamic-extent**, **ftype**, **function**, **ignore**, **inline**, **notinline**, **optimize**, **special**, or **type**; or a *symbol* which is the *name* of a *type*; or a *symbol* which has been *declared* to be a *declaration identifier* by using a **declaration** *declaration*.

declaration specifier *n.* an *expression* that can appear at top level of a **declare** expression or a **declaim** form, or as the argument to **proclaim**, and which has a *car* which is a *declaration identifier*, and which has a *cdr* that is data interpreted according to rules specific to the *declaration identifier*.

declare *v.* to *establish* a *declaration*. See **declare**, **declaim**, or **proclaim**.

decline *v.* (of a *handler*) to return normally without having *handled* the *condition* being *signaled*, permitting the signaling process to continue as if the *handler* had not been present.

decoded time *n.* *absolute time*, represented as an ordered series of nine *objects* which, taken together, form a description of a point in calendar time, accurate to the nearest second (except that *leap seconds* are ignored). See Section 25.1.4.1 (Decoded Time).

default method *n.* a *method* having no *parameter specializers* other than the *class* `t`. Such a *method* is always an *applicable method* but might be *shadowed*$_2$ by a more specific *method*.

defaulted initialization argument list *n.* a *list* of alternating initialization argument *names* and *values* in which unsupplied initialization arguments are defaulted, used in the protocol for initializing and reinitializing *instances* of *classes*.

define-method-combination arguments lambda list *n.* a *lambda list* used by the `:arguments` option to **define-method-combination**. See Section 3.4.10 (Define-method-combination Arguments Lambda Lists).

define-modify-macro lambda list *n.* a *lambda list* used by **define-modify-macro**. See Section 3.4.9 (Define-modify-macro Lambda Lists).

defined name *n.* a *symbol* the meaning of which is defined by Common Lisp.

defining form *n.* a *form* that has the side-effect of *establishing* a definition. "**defun** and **defparameter** are defining forms."

defsetf lambda list *n.* a *lambda list* that is like an *ordinary lambda list* except that it does not permit **&aux** and that it permits use of **&environment**. See Section 3.4.7 (Defsetf Lambda Lists).

deftype lambda list *n.* a *lambda list* that is like a *macro lambda list* except that the default *value* for unsupplied *optional parameters* and *keyword parameters* is the *symbol* `*` (rather than **nil**). See Section 3.4.8 (Deftype Lambda Lists).

denormalized *adj., ANSI, IEEE* (of a *float*) conforming to the description of "denormalized" as described by *IEEE Standard for Binary Floating-Point Arithmetic*. For example, in an *implementation* where the minimum possible exponent was `-7` but where `0.001` was a valid mantissa, the number `1.0e-10` might be representable as `0.001e-7` internally even if the *normalized* representation would call for it to be represented instead as `1.0e-10` or `0.1e-9`. By their nature, *denormalized floats* generally have less precision than *normalized floats*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

derived type *n.* a *type specifier* which is defined in terms of an expansion into another *type specifier*. **deftype** defines *derived types*, and there may be other *implementation-defined operators* which do so as well.

derived type specifier *n.* a *type specifier* for a *derived type*.

designator *n.* an *object* that denotes another *object*. In the dictionary entry for an *operator* if a *parameter* is described as a *designator* for a *type*, the description of the *operator* is written in a way that assumes that appropriate coercion to that *type* has already occurred; that is, that the *parameter* is already of the denoted *type*. For more detailed information, see Section 1.4.1.5 (Designators).

destructive *adj.* (of an *operator*) capable of modifying some program-visible aspect of one or more *objects* that are either explicit *arguments* to the *operator* or that can be obtained directly or indirectly from the *global environment* by the *operator*.

destructuring lambda list *n.* an *extended lambda list* used in **destructuring-bind** and nested within *macro lambda lists*. See Section 3.4.5 (Destructuring Lambda Lists).

different *adj.* not the *same* "The strings `"FOO"` and `"foo"` are different under **equal** but not under **equalp**."

digit *n.* (in a *radix*) a *character* that is among the possible digits (`0` to `9`, `A` to `Z`, and `a` to `z`) and that is defined to have an associated numeric weight as a digit in that *radix*. See Section 13.1.4.6 (Digits in a Radix).

dimension *n.* 1. a non-negative *integer* indicating the number of *objects* an *array* can hold along one axis. If the *array* is a *vector* with a *fill pointer*, the *fill pointer* is ignored. "The second dimension of that array is 7." 2. an axis of an array. "This array has six dimensions."

direct instance *n.* (of a *class* C) an *object* whose *class* is C itself, rather than some *subclass* of C. "The function **make-instance** always returns a direct instance of the class which is (or is named by) its first argument."

direct subclass *n.* (of a *class* C_1) a *class* C_2, such that C_1 is a *direct superclass* of C_2.

direct superclass *n.* (of a *class* C_1) a *class* C_2 which was explicitly designated as a *superclass* of C_1 in the definition of C_1.

disestablish *v.t.* to withdraw the *establishment* of an *object*, a *binding*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment*.

disjoint *n.* (of *types*) having no *elements* in common.

dispatching macro character *n.* a *macro character* that has an associated table that specifies the *function* to be called for each *character* that is seen following the *dispatching macro character*. See the *function* **make-dispatch-macro-character**.

displaced array *n.* an *array* which has no storage of its own, but which is instead indirected to the storage of another *array*, called its *target*, at a specified offset, in such a way that any attempt to *access* the *displaced array* implicitly references the *target array*.

distinct *adj.* not *identical*.

documentation string *n.* (in a defining *form*) A *literal string* which because of the context in which it appears (rather than because of some intrinsically observable aspect of the *string*) is taken as documentation. In some cases, the *documentation string* is saved in such a way that it can later be obtained by supplying either an *object*, or by supplying a *name* and a "kind" to the *function* **documentation**. "The body of code in a **defmacro** form can be preceded by a documentation string of kind **function**."

dot *n.* the *standard character* that is variously called "full stop," "period," or "dot" (.). See Figure 2–5.

dotted list *n.* a *list* which has a terminating *atom* that is not **nil**. (An *atom* by itself is not a *dotted list*, however.)

dotted pair *n.* 1. a *cons* whose *cdr* is a *non-list*. 2. any *cons*, used to emphasize the use of the *cons* as a symmetric data pair.

double float *n.* an *object* of *type* **double-float**.

double-quote *n.* the *standard character* that is variously called "quotation mark" or "double quote" ("). See Figure 2–5.

dynamic binding *n.* a *binding* in a *dynamic environment*.

dynamic environment *n.* that part of an *environment* that contains *bindings* with *dynamic extent*. A *dynamic environment* contains, among other things: *exit points* established by **unwind-protect**, and *bindings* of *dynamic variables*, *exit points* established by **catch**, *condition handlers*, and *restarts*.

dynamic extent *n.* an *extent* whose duration is bounded by points of *establishment* and *disestablishment* within the execution of a particular *form*. See *indefinite extent*. "Dynamic variable bindings have dynamic extent."

dynamic scope *n.* *indefinite scope* along with *dynamic extent*.

dynamic variable *n.* a *variable* the *binding* for which is in the *dynamic environment*. See **special**.

E

echo stream *n.* a *stream* of *type* **echo-stream**.

effective method *n.* the combination of *applicable methods* that are executed when a *generic function* is invoked with a particular sequence of *arguments*.

element *n.* 1. (of a *list*) an *object* that is the *car* of one of the *conses* that comprise the *list*. 2. (of an *array*) an *object* that is stored in the *array*. 3. (of a *sequence*) an *object* that is an *element* of the *list* or *array* that is the *sequence*. 4. (of a *type*) an *object* that is a member of the set of *objects* designated by the *type*. 5. (of an *input stream*) a *character* or *number* (as appropriate to the *element type* of the *stream*) that is among the ordered series of *objects* that can be read from the *stream* (using **read-char** or **read-byte**, as appropriate to the *stream*). 6. (of an *output stream*) a *character* or *number* (as appropriate to the *element type* of the *stream*) that is among the ordered series of *objects* that has been or will be written to the *stream* (using **write-char** or **write-byte**, as appropriate to the *stream*). 7. (of a *class*) a *generalized instance* of the *class*.

element type *n.* 1. (of an *array*) the *array element type* of the *array*. 2. (of a *stream*) the *stream element type* of the *stream*.

em *n.* *Trad.* a context-dependent unit of measure commonly used in typesetting, equal to the displayed width of of a letter "M" in the current font. (The letter "M" is traditionally chosen because it is typically represented by the widest *glyph* in the font, and other characters' widths are typically fractions of an *em*. In implementations providing non-Roman characters with wider characters than "M," it is permissible for another character to be the *implementation-defined* reference character for this measure, and for "M" to be only a fraction of an *em* wide.) In a fixed width font, a line with *n* characters is *n ems* wide; in a variable width font, *n ems* is the expected upper bound on the width of such a line.

empty list *n.* the *list* containing no *elements*. See *()*.

empty type *n.* the *type* that contains no *elements*, and that is a *subtype* of all *types* (including itself). See *nil*.

end of file *n.* 1. the point in an *input stream* beyond which there is no further data. Whether or not there is such a point on an *interactive stream* is *implementation-defined*. 2. a *situation* that occurs upon an attempt to obtain data from an *input stream* that is at the *end of file*$_1$.

environment *n.* 1. a set of *bindings*. See Section 3.1.1 (Introduction to Environments). 2. an *environment object*. "**macroexpand** takes an optional environment argument."

environment object *n.* an *object* representing a set of *lexical bindings*, used in the processing of a *form* to provide meanings for *names* within that *form*. "**macroexpand** takes an optional environment argument." (The *object* **nil** when used as an *environment object* denotes the *null lexical environment*; the *values* of *environment parameters* to *macro functions* are *objects* of *implementation-dependent* nature which represent the *environment*$_1$ in which the corresponding *macro form* is to be expanded.) See Section 3.1.1.4 (Environment Objects).

environment parameter *n.* A *parameter* in a *defining form* f for which there is no corresponding *argument*; instead, this *parameter* receives as its value an *environment object* which corresponds to the *lexical environment* in which the *defining form* f appeared.

error *n.* 1. (only in the phrase "is an error") a *situation* in which the semantics of a program are not specified, and in which the consequences are undefined. 2. a *condition* which represents an *error situation*. See Section 1.4.2 (Error Terminology). 3. an *object* of *type* **error**.

error output *n.* the *output stream* which is the *value* of the *dynamic variable* ***error-output***.

escape *n., adj.* 1. *n.* a *single escape* or a *multiple escape*. 2. *adj.* *single escape* or *multiple escape*.

establish *v.t.* to build or bring into being a *binding*, a *declaration*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment*. "**let** establishes lexical bindings."

evaluate *v.t.* (a *form* or an *implicit progn*) to *execute* the *code* represented by the *form* (or the series of *forms* making up the *implicit progn*) by applying the rules of *evaluation*, returning zero or more values.

evaluation *n.* a model whereby *forms* are *executed*, returning zero or more values. Such execution might be implemented directly in one step by an interpreter or in two steps by first *compiling* the *form* and then *executing* the *compiled code*; this choice is dependent both on context and the nature of the *implementation*, but in any case is not in general detectable by any program. The evaluation model is designed in such a way that a *conforming implementation* might legitimately have only a compiler and no interpreter, or vice versa. See Section 3.1.2 (The Evaluation Model).

evaluation environment *n.* a *run-time environment* in which macro expanders and code specified by **eval-when** to be evaluated are evaluated. All evaluations initiated by the *compiler* take place in the *evaluation environment*.

execute *v.t.* *Trad.* (*code*) to perform the imperative actions represented by the *code*.

execution time *n.* the duration of time that *compiled code* is being *executed*.

exhaustive partition *n.* (of a *type*) a set of *pairwise disjoint types* that form an *exhaustive union*.

exhaustive union *n.* (of a *type*) a set of *subtypes* of the *type*, whose union contains all *elements* of that *type*.

exit point *n.* a point in a *control form* from which (*e.g.*, **block**), through which (*e.g.*, **unwind-protect**), or to which (*e.g.*, **tagbody**) control and possibly *values* can be transferred both actively by using another *control form* and passively through the normal control and data flow of *evaluation*. "**catch** and **block** establish bindings for exit points to which **throw** and **return-from**, respectively, can transfer control and values; **tagbody** establishes a binding for an exit point with lexical extent to which **go** can transfer control; and **unwind-protect** establishes an exit point through which control might be transferred by operators such as **throw**, **return-from**, and **go**."

explicit return *n.* the act of transferring control (and possibly *values*) to a *block* by using **return-from** (or **return**).

explicit use *n.* (of a *variable* V in a *form* F) a reference to V that is directly apparent in the normal semantics of F; *i.e.*, that does not expose any undocumented details of the *macro expansion* of the *form* itself. References to V exposed by expanding *subforms* of F are, however, considered to be *explicit uses* of V.

exponent marker *n.* a character that is used in the textual notation for a *float* to separate the mantissa from the exponent. The characters defined as *exponent markers* in the *standard readtable* are shown in Figure 26–1. For more information, see Section 2.1 (Character Syntax). "The exponent marker 'd' in '3.0d7' indicates that this number is to be represented as a double float."

Marker	Meaning
`D` or `d`	**double-float**
`E` or `e`	**float** (see ***read-default-float-format***)
`F` or `f`	**single-float**
`L` or `l`	**long-float**
`S` or `s`	**short-float**

Figure 26–1. Exponent Markers

export *v.t.* (a *symbol* in a *package*) to add the *symbol* to the list of *external symbols* of the *package*.

exported *adj.* (of a *symbol* in a *package*) being an *external symbol* of the *package*.

expressed adjustability *n.* (of an *array*) a *generalized boolean* that is conceptually (but not necessarily actually) associated with the *array*, representing whether the *array* is *expressly adjustable*. See also *actual adjustability*.

expressed array element type *n.* (of an *array*) the *type* which is the *array element type* implied by a *type declaration* for the *array*, or which is the requested *array element type* at its time of creation, prior to any selection of an *upgraded array element type*. (Common Lisp does not provide a way of detecting this *type* directly at run time, but an *implementation* is permitted to make assumptions about the *array*'s contents and the operations which may be performed on the *array* when this *type* is noted during code analysis, even if those assumptions would not be valid in general for the *upgraded array element type* of the *expressed array element type*.)

expressed complex part type *n.* (of a *complex*) the *type* which is implied as the *complex part type* by a *type declaration* for the *complex*, or which is the requested *complex part type* at its time of creation, prior to any selection of an *upgraded complex part type*. (Common Lisp does not provide a way of detecting this *type* directly at run time, but an *implementation* is permitted to make assumptions about the operations which may be performed on the *complex* when this *type* is noted during code analysis, even if those assumptions would not be valid in general for the *upgraded complex part type* of the *expressed complex part type*.)

expression *n.* 1. an *object*, often used to emphasize the use of the *object* to encode or represent information in a specialized format, such as program text. "The second expression in a **let** form is a list of bindings." 2. the textual notation used to notate an *object* in a source file. "The expression `'sample` is equivalent to `(quote sample)`."

expressly adjustable *adj.* (of an *array*) being *actually adjustable* by virtue of an explicit request for this characteristic having been made at the time of its creation. All *arrays* that are *expressly adjustable* are *actually adjustable*, but not necessarily vice versa.

extended character *n.* a *character* of *type* **extended-char**: a *character* that is not a *base character*.

extended function designator *n.* a *designator* for a *function*; that is, an *object* that denotes a *function* and that is one of: a *function name* (denoting the *function* it names in the *global environment*), or a *function* (denoting itself). The consequences are undefined if a *function name* is used as an *extended function designator* but it does not have a global definition as a *function*, or if it is a *symbol* that has a global definition as a *macro* or a *special form*. See also *function designator*.

extended lambda list *n.* a list resembling an *ordinary lambda list* in form and

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

purpose, but offering additional syntax or functionality not available in an *ordinary lambda list*. "**defmacro** uses extended lambda lists."

extension *n.* a facility in an *implementation* of Common Lisp that is not specified by this standard.

extent *n.* the interval of time during which a *reference* to an *object*, a *binding*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment* is defined.

external file format *n.* an *object* of *implementation-dependent* nature which determines one of possibly several *implementation-dependent* ways in which *characters* are encoded externally in a *character file*.

external file format designator *n.* a *designator* for an *external file format*; that is, an *object* that denotes an *external file format* and that is one of: the *symbol* `:default` (denoting an *implementation-dependent* default *external file format* that can accomodate at least the *base characters*), some other *object* defined by the *implementation* to be an *external file format designator* (denoting an *implementation-defined external file format*), or some other *object* defined by the *implementation* to be an *external file format* (denoting itself).

external symbol *n.* (of a *package*) a *symbol* that is part of the 'external interface' to the *package* and that are *inherited*$_3$ by any other *package* that *uses* the *package*. When using the *Lisp reader*, if a *package prefix* is used, the *name* of an *external symbol* is separated from the *package name* by a single *package marker* while the *name* of an *internal symbol* is separated from the *package name* by a double *package marker*; see Section 2.3.4 (Symbols as Tokens).

externalizable object *n.* an *object* that can be used as a *literal object* in *code* to be processed by the *file compiler*.

F

false *n.* the *symbol* **nil**, used to represent the failure of a *predicate* test.

fbound [ˈefˌbau̇nd] *adj.* (of a *function name*) *bound* in the *function namespace*. (The *names* of *macros* and *special operators* are *fbound*, but the nature and *type* of the *object* which is their *value* is *implementation-dependent*. Further, defining a *setf expander* *F* does not cause the *setf function* `(setf F)` to become defined; as such, if there is a such a definition of a *setf expander* *F*, the *function* `(setf F)` can be *fbound* if and only if, by design or coincidence, a function binding for `(setf F)` has been independently established.) See the *functions* **fboundp** and **symbol-function**.

feature *n.* 1. an aspect or attribute of Common Lisp, of the *implementation*, or of the *environment*. 2. a *symbol* that names a *feature*$_1$. See Section 24.1.2 (Features). "The `:ansi-cl` feature is present in all conforming implementations."

feature expression *n.* A boolean combination of *features* used by the `#+` and `#-` *reader macros* in order to direct conditional *reading* of *expressions* by the *Lisp reader*. See Section 24.1.2.1 (Feature Expressions).

features list *n.* the *list* that is the *value* of ***features***.

file *n.* a named entry in a *file system*, having an *implementation-defined* nature.

file compiler *n.* any *compiler* which *compiles source code* contained in a *file*, producing a *compiled file* as output. The **compile-file** function is the only interface to such a *compiler* provided by Common Lisp, but there might be other, *implementation-defined* mechanisms for invoking the *file compiler*.

file position *n.* (in a *stream*) a non-negative *integer* that represents a position in the *stream*. Not all *streams* are able to represent the notion of *file position*; in the description of any *operator* which manipulates *file positions*, the behavior for *streams* that don't have this notion must be explicitly stated. For *binary streams*, the *file position* represents the number of preceding *bytes* in the *stream*. For *character streams*, the constraint is more relaxed: *file positions* must increase monotonically, the amount of the increase between *file positions* corresponding to any two successive characters in the *stream* is *implementation-dependent*.

file position designator *n.* (in a *stream*) a *designator* for a *file position* in that *stream*; that is, the symbol `:start` (denoting `0`, the first *file position* in that *stream*), the symbol `:end` (denoting the last *file position* in that *stream*; *i.e.*, the position following the last *element* of the *stream*), or a *file position* (denoting itself).

file stream *n.* an *object* of *type* **file-stream**.

file system *n.* a facility which permits aggregations of data to be stored in named *files* on some medium that is external to the *Lisp image* and that therefore persists from *session* to *session*.

filename *n.* a handle, not necessarily ever directly represented as an *object*, that can be used to refer to a *file* in a *file system*. *Pathnames* and *namestrings* are two kinds of *objects* that substitute for *filenames* in Common Lisp.

fill pointer *n.* (of a *vector*) an *integer* associated with a *vector* that represents the index above which no *elements* are *active*. (A *fill pointer* is a non-negative *integer* no larger than the total number of *elements* in the *vector*. Not all *vectors* have *fill pointers*.)

finite *adj.* (of a *type*) having a finite number of *elements*. "The type specifier `(integer 0 5)` denotes a finite type, but the type specifiers **integer** and `(integer 0)` do not."

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

fixnum *n.* an *integer* of *type* **fixnum**.

float *n.* an *object* of *type* **float**.

for-value *adj.* (of a *reference* to a *binding*) being a *reference* that $reads_1$ the *value* of the *binding*.

form *n.* 1. any *object* meant to be *evaluated*. 2. a *symbol*, a *compound form*, or a *self-evaluating object*. 3. (for an *operator*, as in "⟪*operator*⟫ *form*") a *compound form* having that *operator* as its first element. "A **quote** form is a constant form."

formal argument *n. Trad.* a *parameter*.

formal parameter *n. Trad.* a *parameter*.

format *v.t.* (a *format control* and *format arguments*) to perform output as if by **format**, using the *format string* and *format arguments*.

format argument *n.* an *object* which is used as data by functions such as **format** which interpret *format controls*.

format control *n.* a *format string*, or a *function* that obeys the *argument* conventions for a *function* returned by the **formatter** *macro*. See Section 22.2.1.3 (Compiling Format Strings).

format directive *n.* 1. a sequence of *characters* in a *format string* which is introduced by a *tilde*, and which is specially interpreted by *code* which processes *format strings* to mean that some special operation should be performed, possibly involving data supplied by the *format arguments* that accompanied the *format string*. See the *function* **format**. "In `"~D base 10 = ~8R"`, the character sequences '`~D`' and '`~8R`' are format directives." 2. the conceptual category of all *format* $directives_1$ which use the same dispatch character. "Both `"~3d"` and `"~3,'0D"` are valid uses of the '`~D`' format directive."

format string *n.* a *string* which can contain both ordinary text and *format directives*, and which is used in conjunction with *format arguments* to describe how text output should be formatted by certain functions, such as **format**.

free declaration *n.* a declaration that is not a *bound declaration*. See **declare**.

fresh *adj.* 1. (of an *object yielded* by a *function*) having been newly-allocated by that *function*. (The caller of a *function* that returns a *fresh object* may freely modify the *object* without fear that such modification will compromise the future correct behavior of that *function*.) 2. (of a *binding* for a *name*) newly-allocated; not shared with other *bindings* for that *name*.

freshline *n.* a conceptual operation on a *stream*, implemented by the *function* **fresh-line** and by the *format directive* ~&, which advances the display position to the beginning of the next line (as if a *newline* had been typed, or the *function* **terpri** had been called) unless the *stream* is already known to be positioned at the beginning of a line. Unlike *newline*, *freshline* is not a *character*.

funbound [ˈefunbaůnd] *n.* (of a *function name*) not *fbound*.

function *n.* 1. an *object* representing code, which can be *called* with zero or more *arguments*, and which produces zero or more *values*. 2. an *object* of *type* **function**.

function block name *n.* (of a *function name*) The *symbol* that would be used as the name of an *implicit block* which surrounds the body of a *function* having that *function name*. If the *function name* is a *symbol*, its *function block name* is the *function name* itself. If the *function name* is a *list* whose *car* is **setf** and whose *cadr* is a *symbol*, its *function block name* is the *symbol* that is the *cadr* of the *function name*. An *implementation* which supports additional kinds of *function names* must specify for each how the corresponding *function block name* is computed.

function cell *n. Trad.* (of a *symbol*) The *place* which holds the *definition* of the global *function binding*, if any, named by that *symbol*, and which is *accessed* by **symbol-function**. See *cell*.

function designator *n.* a *designator* for a *function*; that is, an *object* that denotes a *function* and that is one of: a *symbol* (denoting the *function* named by that *symbol* in the *global environment*), or a *function* (denoting itself). The consequences are undefined if a *symbol* is used as a *function designator* but it does not have a global definition as a *function*, or it has a global definition as a *macro* or a *special form*. See also *extended function designator*.

function form *n.* a *form* that is a *list* and that has a first element which is the *name* of a *function* to be called on *arguments* which are the result of *evaluating* subsequent elements of the *function form*.

function name *n.* 1. (in an *environment*) A *symbol* or a *list* (setf *symbol*) that is the *name* of a *function* in that *environment*. 2. A *symbol* or a *list* (setf *symbol*).

functional evaluation *n.* the process of extracting a *functional value* from a *function name* or a *lambda expression*. The evaluator performs *functional evaluation* implicitly when it encounters a *function name* or a *lambda expression* in the *car* of a *compound form*, or explicitly when it encounters a **function** *special form*. Neither a use of a *symbol* as a *function designator* nor a use of the *function* **symbol-function** to extract the *functional value* of a *symbol* is considered a *functional evaluation*.

functional value *n.* 1. (of a *function name* N in an *environment* E) The *value* of the *binding* named N in the *function namespace* for *environment* E; that is, the

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

contents of the *function cell* named N in *environment* E. 2. (of an *fbound symbol* S) the contents of the *symbol*'s *function cell*; that is, the *value* of the *binding* named S in the *function namespace* of the *global environment*. (A *name* that is a *macro name* in the *global environment* or is a *special operator* might or might not be *fbound*. But if S is such a *name* and is *fbound*, the specific nature of its *functional value* is *implementation-dependent*; in particular, it might or might not be a *function*.)

further compilation *n. implementation-dependent* compilation beyond *minimal compilation*. Further compilation is permitted to take place at *run time*. "Block compilation and generation of machine-specific instructions are examples of further compilation."

G

general *adj.* (of an *array*) having *element type* **t**, and consequently able to have any *object* as an *element*.

generalized boolean *n.* an *object* used as a truth value, where the symbol **nil** represents *false* and all other *objects* represent *true*. See *boolean*.

generalized instance *n.* (of a *class*) an *object* the *class* of which is either that *class* itself, or some subclass of that *class*. (Because of the correspondence between types and classes, the term "generalized instance of X" implies "object of type X" and in cases where X is a *class* (or *class name*) the reverse is also true. The former terminology emphasizes the view of X as a *class* while the latter emphasizes the view of X as a *type specifier*.)

generalized reference *n.* a reference to a location storing an *object* as if to a *variable*. (Such a reference can be either to *read* or *write* the location.) See Section 5.1 (Generalized Reference). See also *place*.

generalized synonym stream *n.* (with a *synonym stream symbol*) 1. (to a *stream*) a *synonym stream* to the *stream*, or a *composite stream* which has as a target a *generalized synonym stream* to the *stream*. 2. (to a *symbol*) a *synonym stream* to the *symbol*, or a *composite stream* which has as a target a *generalized synonym stream* to the *symbol*.

generic function *n.* a *function* whose behavior depends on the *classes* or identities of the arguments supplied to it and whose parts include, among other things, a set of *methods*, a *lambda list*, and a *method combination* type.

generic function lambda list *n.* A *lambda list* that is used to describe data flow into a *generic function*. See Section 3.4.2 (Generic Function Lambda Lists).

gensym *n. Trad.* an *uninterned symbol*. See the *function* **gensym**.

global declaration *n.* a *form* that makes certain kinds of information about code globally available; that is, a **proclaim** *form* or a **declaim** *form*.

global environment *n.* that part of an *environment* that contains *bindings* with *indefinite scope* and *indefinite extent*.

global variable *n.* a *dynamic variable* or a *constant variable*.

glyph *n.* a visual representation. "Graphic characters have associated glyphs."

go *v.* to transfer control to a *go point*. See the *special operator* **go**.

go point one of possibly several *exit points* that are *established* by **tagbody** (or other abstractions, such as **prog**, which are built from **tagbody**).

go tag *n.* the *symbol* or *integer* that, within the *lexical scope* of a **tagbody** *form*, names an *exit point established* by that **tagbody** *form*.

graphic *adj.* (of a *character*) being a "printing" or "displayable" *character* that has a standard visual representation as a single *glyph*, such as `A` or `*` or `=`. *Space* is defined to be *graphic*. Of the *standard characters*, all but *newline* are *graphic*. See *non-graphic*.

H

handle *v.* (of a *condition* being *signaled*) to perform a non-local transfer of control, terminating the ongoing *signaling* of the *condition*.

handler *n.* a *condition handler*.

hash table *n.* an *object* of *type* **hash-table**, which provides a mapping from *keys* to *values*.

home package *n.* (of a *symbol*) the *package*, if any, which is contents of the *package cell* of the *symbol*, and which dictates how the *Lisp printer* prints the *symbol* when it is not *accessible* in the *current package*. (*Symbols* which have **nil** in their *package cell* are said to have no *home package*, and also to be *apparently uninterned*.)

I

I/O customization variable *n.* one of the *stream variables* in Figure 26–2, or some other (*implementation-defined*) *stream variable* that is defined by the *implementation* to be an *I/O customization variable*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

debug-io	*error-io*	query-io*
standard-input	*standard-output*	*trace-output*

Figure 26–2. Standardized I/O Customization Variables

identical *adj.* the *same* under **eq**.

identifier *n.* 1. a *symbol* used to identify or to distinguish *names*. 2. a *string* used the same way.

immutable *adj.* not subject to change, either because no *operator* is provided which is capable of effecting such change or because some constraint exists which prohibits the use of an *operator* that might otherwise be capable of effecting such a change. Except as explicitly indicated otherwise, *implementations* are not required to detect attempts to modify *immutable objects* or *cells*; the consequences of attempting to make such modification are undefined. "Numbers are immutable."

implementation *n.* a system, mechanism, or body of *code* that implements the semantics of Common Lisp.

implementation limit *n.* a restriction imposed by an *implementation*.

implementation-defined *adj.* *implementation-dependent*, but required by this specification to be defined by each *conforming implementation* and to be documented by the corresponding implementor.

implementation-dependent *adj.* describing a behavior or aspect of Common Lisp which has been deliberately left unspecified, that might be defined in some *conforming implementations* but not in others, and whose details may differ between *implementations*. A *conforming implementation* is encouraged (but not required) to document its treatment of each item in this specification which is marked *implementation-dependent*, although in some cases such documentation might simply identify the item as "undefined."

implementation-independent *adj.* used to identify or emphasize a behavior or aspect of Common Lisp which does not vary between *conforming implementations*.

implicit block *n.* a *block* introduced by a *macro form* rather than by an explicit **block** *form*.

implicit compilation *n.* *compilation* performed during *evaluation*.

implicit progn *n.* an ordered set of adjacent *forms* appearing in another *form*, and defined by their context in that *form* to be executed as if within a **progn**.

implicit tagbody *n.* an ordered set of adjacent *forms* and/or *tags* appearing in another *form*, and defined by their context in that *form* to be executed as if within a **tagbody**.

import *v.t.* (a *symbol* into a *package*) to make the *symbol* be *present* in the *package*.

improper list *n.* a *list* which is not a *proper list*: a *circular list* or a *dotted list*.

inaccessible *adj.* not *accessible*.

indefinite extent *n.* an *extent* whose duration is unlimited. "Most Common Lisp objects have indefinite extent."

indefinite scope *n.* *scope* that is unlimited.

indicator *n.* a *property indicator*.

indirect instance *n.* (of a *class* C_1) an *object* of *class* C_2, where C_2 is a *subclass* of C_1. "An integer is an indirect instance of the class **number**."

inherit *v.t.* 1. to receive or acquire a quality, trait, or characteristic; to gain access to a feature defined elsewhere. 2. (a *class*) to acquire the structure and behavior defined by a *superclass*. 3. (a *package*) to make *symbols exported* by another *package accessible* by using **use-package**.

initial pprint dispatch table *n.* the *value* of ***print-pprint-dispatch*** at the time the *Lisp image* is started.

initial readtable *n.* the *value* of ***readtable*** at the time the *Lisp image* is started.

initialization argument list *n.* a *property list* of initialization argument *names* and *values* used in the protocol for initializing and reinitializing *instances* of *classes*. See Section 7.1 (Object Creation and Initialization).

initialization form *n.* a *form* used to supply the initial *value* for a *slot* or *variable*. "The initialization form for a slot in a **defclass** form is introduced by the keyword `:initform`."

input *adj.* (of a *stream*) supporting input operations (*i.e.*, being a "data source"). An *input stream* might also be an *output stream*, in which case it is sometimes called a *bidirectional stream*. See the *function* **input-stream-p**.

instance *n.* 1. a *direct instance*. 2. a *generalized instance*. 3. an *indirect instance*.

integer *n.* an *object* of *type* **integer**, which represents a mathematical integer.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

interactive stream *n.* a *stream* on which it makes sense to perform interactive querying. See Section 21.1.1.1.3 (Interactive Streams).

intern *v.t.* 1. (a *string* in a *package*) to look up the *string* in the *package*, returning either a *symbol* with that *name* which was already *accessible* in the *package* or a newly created *internal symbol* of the *package* with that *name*. 2. *Idiom.* generally, to observe a protocol whereby objects which are equivalent or have equivalent names under some predicate defined by the protocol are mapped to a single canonical object.

internal symbol *n.* (of a *package*) a symbol which is *accessible* in the *package*, but which is not an *external symbol* of the *package*.

internal time *n.* *time*, represented as an *integer* number of *internal time units*. *Absolute internal time* is measured as an offset from an arbitrarily chosen, *implementation-dependent* base. See Section 25.1.4.3 (Internal Time).

internal time unit *n.* a unit of time equal to $1/n$ of a second, for some *implementation-defined integer* value of n. See the *variable* **internal-time-units-per-second**.

interned *adj.* *Trad.* 1. (of a *symbol*) $accessible_3$ in any *package*. 2. (of a *symbol* in a specific *package*) *present* in that *package*.

interpreted function *n.* a *function* that is not a *compiled function*. (It is possible for there to be a *conforming implementation* which has no *interpreted functions*, but a *conforming program* must not assume that all *functions* are *compiled functions*.)

interpreted implementation *n.* an *implementation* that uses an execution strategy for *interpreted functions* that does not involve a one-time semantic analysis pre-pass, and instead uses "lazy" (and sometimes repetitious) semantic analysis of *forms* as they are encountered during execution.

interval designator *n.* (of *type* T) an ordered pair of *objects* that describe a *subtype* of T by delimiting an interval on the real number line. See Section 12.1.6 (Interval Designators).

invalid *n.*, *adj.* 1. *n.* a possible *constituent trait* of a *character* which if present signifies that the *character* cannot ever appear in a *token* except under the control of a *single escape character*. For details, see Section 2.1.4.1 (Constituent Characters). 2. *adj.* (of a *character*) being a *character* that has *syntax type constituent* in the *current readtable* and that has the *constituent trait* $invalid_1$. See Figure 2–8.

iteration form *n.* a *compound form* whose *operator* is named in Figure 26–3, or a *compound form* that has an *implementation-defined operator* and that is defined by the *implementation* to be an *iteration form*.

do	**do-external-symbols**	**dotimes**
do*	**do-symbols**	**loop**
do-all-symbols	**dolist**	

Figure 26–3. Standardized Iteration Forms

iteration variable *n.* a *variable* V, the *binding* for which was created by an *explicit use* of V in an *iteration form*.

K

key *n.* an *object* used for selection during retrieval. See *association list*, *property list*, and *hash table*. Also, see Section 17.1 (Sequence Concepts).

keyword *n.* 1. a *symbol* the *home package* of which is the KEYWORD *package*. 2. any *symbol*, usually but not necessarily in the KEYWORD *package*, that is used as an identifying marker in keyword-style argument passing. See **lambda**. 3. *Idiom.* a *lambda list keyword*.

keyword parameter *n.* A *parameter* for which a corresponding keyword *argument* is optional. (There is no such thing as a required keyword *argument*.) If the *argument* is not supplied, a default value is used. See also *supplied-p parameter*.

keyword/value pair *n.* two successive *elements* (a *keyword* and a *value*, respectively) of a *property list*.

L

lambda combination *n. Trad.* a *lambda form*.

lambda expression *n.* a *list* which can be used in place of a *function name* in certain contexts to denote a *function* by directly describing its behavior rather than indirectly by referring to the name of an *established function*; its name derives from the fact that its first element is the *symbol* `lambda`. See **lambda**.

lambda form *n.* a *form* that is a *list* and that has a first element which is a *lambda expression* representing a *function* to be called on *arguments* which are the result of *evaluating* subsequent elements of the *lambda form*.

lambda list *n.* a *list* that specifies a set of *parameters* (sometimes called *lambda variables*) and a protocol for receiving *values* for those *parameters*; that is, an *ordinary lambda list*, an *extended lambda list*, or a *modified lambda list*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

lambda list keyword *n.* a *symbol* whose *name* begins with *ampersand* and that is specially recognized in a *lambda list*. Note that no *standardized lambda list keyword* is in the KEYWORD *package*.

lambda variable *n.* a *formal parameter*, used to emphasize the *variable*'s relation to the *lambda list* that *established* it.

leaf *n.* 1. an *atom* in a $tree_1$. 2. a terminal node of a $tree_2$.

leap seconds *n.* additional one-second intervals of time that are occasionally inserted into the true calendar by official timekeepers as a correction similar to "leap years." All Common Lisp *time* representations ignore *leap seconds*; every day is assumed to be exactly 86400 seconds long.

left-parenthesis *n.* the *standard character* "(", that is variously called "left parenthesis" or "open parenthesis" See Figure 2–5.

length *n.* (of a *sequence*) the number of *elements* in the *sequence*. (Note that if the *sequence* is a *vector* with a *fill pointer*, its *length* is the same as the *fill pointer* even though the total allocated size of the *vector* might be larger.)

lexical binding *n.* a *binding* in a *lexical environment*.

lexical closure *n.* a *function* that, when invoked on *arguments*, executes the body of a *lambda expression* in the *lexical environment* that was captured at the time of the creation of the *lexical closure*, augmented by *bindings* of the *function*'s *parameters* to the corresponding *arguments*.

lexical environment *n.* that part of the *environment* that contains *bindings* whose names have *lexical scope*. A *lexical environment* contains, among other things: ordinary *bindings* of *variable names* to *values*, lexically *established bindings* of *function names* to *functions*, *macros*, *symbol macros*, *blocks*, *tags*, and *local declarations* (see **declare**).

lexical scope *n.* *scope* that is limited to a spatial or textual region within the establishing *form*. "The names of parameters to a function normally are lexically scoped."

lexical variable *n.* a *variable* the *binding* for which is in the *lexical environment*.

Lisp image *n.* a running instantiation of a Common Lisp *implementation*. A *Lisp image* is characterized by a single address space in which any *object* can directly refer to any another in conformance with this specification, and by a single, common, *global environment*. (External operating systems sometimes call this a "core image," "fork," "incarnation," "job," or "process." Note however, that the issue of a "process" in such an operating system is technically orthogonal to the issue of a *Lisp image* being defined here. Depending on the operating system, a single "process" might have multiple *Lisp images*, and multiple "processes" might reside in a single *Lisp image*. Hence, it is the idea of a fully shared address space for direct reference among all *objects* which is the defining characteristic. Note, too, that two "processes" which have a communication area that permits the sharing of some but not all *objects* are considered to be distinct *Lisp images*.)

Lisp printer *n. Trad.* the procedure that prints the character representation of an *object* onto a *stream*. (This procedure is implemented by the *function* **write**.)

Lisp read-eval-print loop *n. Trad.* an endless loop that *reads*$_2$ a *form*, *evaluates* it, and prints (*i.e.*, *writes*$_2$) the results. In many *implementations*, the default mode of interaction with Common Lisp during program development is through such a loop.

Lisp reader *n. Trad.* the procedure that parses character representations of *objects* from a *stream*, producing *objects*. (This procedure is implemented by the *function* **read**.)

list *n.* 1. a chain of *conses* in which the *car* of each *cons* is an *element* of the *list*, and the *cdr* of each *cons* is either the next link in the chain or a terminating *atom*. See also *proper list*, *dotted list*, or *circular list*. 2. the *type* that is the union of **null** and **cons**.

list designator *n.* a *designator* for a *list* of *objects*; that is, an *object* that denotes a *list* and that is one of: a *non-nil atom* (denoting a *singleton list* whose *element* is that *non-nil atom*) or a *proper list* (denoting itself).

list structure *n.* (of a *list*) the set of *conses* that make up the *list*. Note that while the *car*$_{1b}$ component of each such *cons* is part of the *list structure*, the *objects* that are *elements* of the *list* (*i.e.*, the *objects* that are the *cars*$_2$ of each *cons* in the *list*) are not themselves part of its *list structure*, even if they are *conses*, except in the (*circular*$_2$) case where the *list* actually contains one of its *tails* as an *element*. (The *list structure* of a *list* is sometimes redundantly referred to as its "top-level list structure" in order to emphasize that any *conses* that are *elements* of the *list* are not involved.)

literal *adj.* (of an *object*) referenced directly in a program rather than being computed by the program; that is, appearing as data in a **quote** *form*, or, if the *object* is a *self-evaluating object*, appearing as unquoted data. "In the form `(cons "one" '("two"))`, the expressions `"one"`, `("two")`, and `"two"` are literal objects."

load *v.t.* (a *file*) to cause the *code* contained in the *file* to be *executed*. See the *function* **load**.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

load time *n.* the duration of time that the loader is *loading compiled code*.

load time value *n.* an *object* referred to in *code* by a **load-time-value** *form*. The *value* of such a *form* is some specific *object* which can only be computed in the run-time *environment*. In the case of *file compilation*, the *value* is computed once as part of the process of *loading* the *compiled file*, and not again. See the *special operator* **load-time-value**.

loader *n.* a facility that is part of Lisp and that *loads* a *file*. See the *function* **load**.

local declaration *n.* an *expression* which may appear only in specially designated positions of certain *forms*, and which provides information about the code contained within the containing *form*; that is, a **declare** *expression*.

local precedence order *n.* (of a *class*) a *list* consisting of the *class* followed by its *direct superclasses* in the order mentioned in the defining *form* for the *class*.

local slot *n.* (of a *class*) a *slot accessible* in only one *instance*, namely the *instance* in which the *slot* is allocated.

logical block *n.* a conceptual grouping of related output used by the *pretty printer*. See the *macro* **pprint-logical-block** and Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

logical host *n.* an *object* of *implementation-dependent* nature that is used as the representation of a "host" in a *logical pathname*, and that has an associated set of translation rules for converting *logical pathnames* belonging to that host into *physical pathnames*. See Section 19.3 (Logical Pathnames).

logical host designator *n.* a *designator* for a *logical host*; that is, an *object* that denotes a *logical host* and that is one of: a *string* (denoting the *logical host* that it names), or a *logical host* (denoting itself). (Note that because the representation of a *logical host* is *implementation-dependent*, it is possible that an *implementation* might represent a *logical host* as the *string* that names it.)

logical pathname *n.* an *object* of *type* **logical-pathname**.

long float *n.* an *object* of *type* **long-float**.

loop keyword *n. Trad.* a symbol that is a specially recognized part of the syntax of an extended **loop** *form*. Such symbols are recognized by their *name* (using **string=**), not by their identity; as such, they may be in any package. A *loop keyword* is not a *keyword*.

lowercase *adj.* (of a *character*) being among *standard characters* corresponding to the small letters a through z, or being some other *implementation-defined character* that is defined by the *implementation* to be *lowercase*. See Section 13.1.4.3 (Characters With Case).

M

macro *n.* 1. a *macro form* 2. a *macro function*. 3. a *macro name*.

macro character *n.* a *character* which, when encountered by the *Lisp reader* in its main dispatch loop, introduces a *reader* $macro_1$. (*Macro characters* have nothing to do with *macros*.)

macro expansion *n.* 1. the process of translating a *macro form* into another *form*. 2. the *form* resulting from this process.

macro form *n.* a *form* that stands for another *form* (*e.g.*, for the purposes of abstraction, information hiding, or syntactic convenience); that is, either a *compound form* whose first element is a *macro name*, or a *form* that is a *symbol* that names a *symbol macro*.

macro function *n.* a *function* of two arguments, a *form* and an *environment*, that implements *macro expansion* by producing a *form* to be evaluated in place of the original argument *form*.

macro lambda list *n.* an *extended lambda list* used in *forms* that *establish macro* definitions, such as **defmacro** and **macrolet**. See Section 3.4.4 (Macro Lambda Lists).

macro name *n.* a *name* for which **macro-function** returns *true* and which when used as the first element of a *compound form* identifies that *form* as a *macro form*.

macroexpand hook *n.* the *function* that is the *value* of ***macroexpand-hook***.

mapping *n.* 1. a type of iteration in which a *function* is successively applied to *objects* taken from corresponding entries in collections such as *sequences* or *hash tables*. 2. *Math.* a relation between two sets in which each element of the first set (the "domain") is assigned one element of the second set (the "range").

metaclass *n.* 1. a *class* whose instances are *classes*. 2. (of an *object*) the *class* of the *class* of the *object*.

Metaobject Protocol *n.* one of many possible descriptions of how a *conforming implementation* might implement various aspects of the object system. This description is beyond the scope of this document, and no *conforming implementation* is required to adhere to it except as noted explicitly in this specification. Nevertheless,

its existence helps to establish normative practice, and implementors with no reason to diverge from it are encouraged to consider making their *implementation* adhere to it where possible. It is described in detail in *The Art of the Metaobject Protocol*.

method *n.* an *object* that is part of a *generic function* and which provides information about how that *generic function* should behave when its *arguments* are *objects* of certain *classes* or with certain identities.

method combination *n.* 1. generally, the composition of a set of *methods* to produce an *effective method* for a *generic function*. 2. an object of *type* **method-combination**, which represents the details of how the *method* $combination_1$ for one or more specific *generic functions* is to be performed.

method-defining form *n.* a *form* that defines a *method* for a *generic function*, whether explicitly or implicitly. See Section 7.6.1 (Introduction to Generic Functions).

method-defining operator *n.* an *operator* corresponding to a *method-defining form*. See Figure 7–1.

minimal compilation *n.* actions the *compiler* must take at compile time. See Section 3.2.2 (Compilation Semantics).

modified lambda list *n.* a list resembling an *ordinary lambda list* in form and purpose, but which deviates in syntax or functionality from the definition of an *ordinary lambda list*. See *ordinary lambda list*. "**deftype** uses a modified lambda list."

most recent *adj.* innermost; that is, having been *established* (and not yet *disestablished*) more recently than any other of its kind.

multiple escape *n.*, *adj.* 1. *n.* the *syntax type* of a *character* that is used in pairs to indicate that the enclosed *characters* are to be treated as $alphabetic_2$ *characters* with their *case* preserved. For details, see Section 2.1.4.5 (Multiple Escape Characters). 2. *adj.* (of a *character*) having the *multiple escape syntax type*. 3. *n.* a *multiple* $escape_2$ *character*. (In the *standard readtable*, *vertical-bar* is a *multiple escape character*.)

multiple values *n.* 1. more than one *value*. "The function **truncate** returns multiple values." 2. a variable number of *values*, possibly including zero or one. "The function **values** returns multiple values." 3. a fixed number of values other than one. "The macro **multiple-value-bind** is among the few operators in Common Lisp which can detect and manipulate multiple values."

N

name *n.*, *v.t.* 1. *n.* an *identifier* by which an *object*, a *binding*, or an *exit point* is referred to by association using a *binding*. 2. *v.t.* to give a *name* to. 3. *n.* (of an *object* having a name component) the *object* which is that component. "The string which is a symbol's name is returned by **symbol-name**." 4. *n.* (of a *pathname*) a. the name component, returned by **pathname-name**. b. the entire namestring, returned by **namestring**. 5. *n.* (of a *character*) a *string* that names the *character* and that has *length* greater than one. (All *non-graphic characters* are required to have *names* unless they have some *implementation-defined attribute* which is not *null*. Whether or not other *characters* have *names* is *implementation-dependent*.)

named constant *n.* a *variable* that is defined by Common Lisp, by the *implementation*, or by user code (see the *macro* **defconstant**) to always *yield* the same *value* when *evaluated*. "The value of a named constant may not be changed by assignment or by binding."

namespace *n.* 1. *bindings* whose denotations are restricted to a particular kind. "The bindings of names to tags is the tag namespace." 2. any *mapping* whose domain is a set of *names*. "A package defines a namespace."

namestring *n.* a *string* that represents a *filename* using either the *standardized* notation for naming *logical pathnames* described in Section 19.3.1 (Syntax of Logical Pathname Namestrings), or some *implementation-defined* notation for naming a *physical pathname*.

newline *n.* the *standard character* ⟨*Newline*⟩, notated for the *Lisp reader* as `#\Newline`.

next method *n.* the next *method* to be invoked with respect to a given *method* for a particular set of arguments or argument *classes*. See Section 7.6.6.1.3 (Applying method combination to the sorted list of applicable methods).

nickname *n.* (of a *package*) one of possibly several *names* that can be used to refer to the *package* but that is not the primary *name* of the *package*.

nil *n.* the *object* that is at once the *symbol* named `"NIL"` in the `COMMON-LISP` *package*, the *empty list*, the *boolean* (or *generalized boolean*) representing *false*, and the *name* of the *empty type*.

non-atomic *adj.* being other than an *atom*; *i.e.*, being a *cons*.

non-constant variable *n.* a *variable* that is not a *constant variable*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

non-correctable *adj.* (of an *error*) not intentionally *correctable*. (Because of the dynamic nature of *restarts*, it is neither possible nor generally useful to completely prohibit an *error* from being *correctable*. This term is used in order to express an intent that no special effort should be made by *code* signaling an *error* to make that *error correctable*; however, there is no actual requirement on *conforming programs* or *conforming implementations* imposed by this term.)

non-empty *adj.* having at least one *element*.

non-generic function *n.* a *function* that is not a *generic function*.

non-graphic *adj.* (of a *character*) not *graphic*. See Section 13.1.4.1 (Graphic Characters).

non-list *n.*, *adj.* other than a *list*; *i.e.*, a *non-nil atom*.

non-local exit *n.* a transfer of control (and sometimes *values*) to an *exit point* for reasons other than a *normal return*. "The operators **go**, **throw**, and **return-from** cause a non-local exit."

non-nil *n.*, *adj.* not **nil**. Technically, any *object* which is not **nil** can be referred to as *true*, but that would tend to imply a unique view of the *object* as a *generalized boolean*. Referring to such an *object* as *non-nil* avoids this implication.

non-null lexical environment *n.* a *lexical environment* that has additional information not present in the *global environment*, such as one or more *bindings*.

non-simple *adj.* not *simple*.

non-terminating *adj.* (of a *macro character*) being such that it is treated as a constituent *character* when it appears in the middle of an extended token. See Section 2.2 (Reader Algorithm).

non-top-level form *n.* a *form* that, by virtue of its position as a *subform* of another *form*, is not a *top level form*. See Section 3.2.3.1 (Processing of Top Level Forms).

normal return *n.* the natural transfer of control and *values* which occurs after the complete *execution* of a *form*.

normalized *adj.*, *ANSI*, *IEEE* (of a *float*) conforming to the description of "normalized" as described by *IEEE Standard for Binary Floating-Point Arithmetic*. See *denormalized*.

null *adj.*, *n.* 1. *adj.* a. (of a *list*) having no *elements*: empty. See *empty list*. b. (of a *string*) having a *length* of zero. (It is common, both within this document and in observed spoken behavior, to refer to an empty string by an apparent definite reference, as in "the *null string*" even though no attempt is made to *intern*$_2$ null strings. The phrase "a *null string*" is technically more correct, but is generally considered awkward by most Lisp programmers. As such, the phrase "the *null string*" should be treated as an indefinite reference in all cases except for anaphoric references.) c. (of an *implementation-defined attribute* of a *character*) An *object* to which the value of that *attribute* defaults if no specific value was requested. 2. *n.* an *object* of *type* **null** (the only such *object* being **nil**).

null lexical environment *n.* the *lexical environment* which has no *bindings*.

number *n.* an *object* of *type* **number**.

numeric *adj.* (of a *character*) being one of the *standard characters* `0` through *9*, or being some other *graphic character* defined by the *implementation* to be *numeric*.

O

object *n.* 1. any Lisp datum. "The function **cons** creates an object which refers to two other objects." 2. (immediately following the name of a *type*) an *object* which is of that *type*, used to emphasize that the *object* is not just a *name* for an object of that *type* but really an *element* of the *type* in cases where *objects* of that *type* (such as **function** or **class**) are commonly referred to by *name*. "The function **symbol-function** takes a function name and returns a function object."

object-traversing *adj.* operating in succession on components of an *object*. "The operators **mapcar**, **maphash**, **with-package-iterator** and **count** perform object-traversing operations."

open *adj.*, *v.t.* (a *file*) 1. *v.t.* to create and return a *stream* to the *file*. 2. *adj.* (of a *stream*) having been *opened*$_1$, but not yet *closed*.

operator *n.* 1. a *function*, *macro*, or *special operator*. 2. a *symbol* that names such a *function*, *macro*, or *special operator*. 3. (in a **function** *special form*) the *cadr* of the **function** *special form*, which might be either an *operator*$_2$ or a *lambda expression*. 4. (of a *compound form*) the *car* of the *compound form*, which might be either an *operator*$_2$ or a *lambda expression*, and which is never `(setf` *symbol*`)`.

optimize quality *n.* one of several aspects of a program that might be optimizable by certain compilers. Since optimizing one such quality might conflict with optimizing another, relative priorities for qualities can be established in an **optimize** *declaration*. The *standardized optimize qualities* are `compilation-speed` (speed of the compilation process), `debug` (ease of debugging), `safety` (run-time error checking), `space` (both code size and run-time space), and `speed` (of the object code). *Implementations* may define additional *optimize qualities*.

optional parameter *n.* A *parameter* for which a corresponding positional *argument* is optional. If the *argument* is not supplied, a default value is used. See also *supplied-p parameter*.

ordinary function *n.* a *function* that is not a *generic function*.

ordinary lambda list *n.* the kind of *lambda list* used by **lambda**. See *modified lambda list* and *extended lambda list*. "**defun** uses an ordinary lambda list."

otherwise inaccessible part *n.* (of an *object*, O_1) an *object*, O_2, which would be made *inaccessible* if O_1 were made *inaccessible*. (Every *object* is an *otherwise inaccessible part* of itself.)

output *adj.* (of a *stream*) supporting output operations (*i.e.*, being a "data sink"). An *output stream* might also be an *input stream*, in which case it is sometimes called a *bidirectional stream*. See the *function* **output-stream-p**.

P

package *n.* an *object* of *type* **package**.

package cell *n.* *Trad.* (of a *symbol*) The *place* in a *symbol* that holds one of possibly several *packages* in which the *symbol* is *interned*, called the *home package*, or which holds **nil** if no such *package* exists or is known. See the *function* **symbol-package**.

package designator *n.* a *designator* for a *package*; that is, an *object* that denotes a *package* and that is one of: a *string designator* (denoting the *package* that has the *string* that it designates as its *name* or as one of its *nicknames*), or a *package* (denoting itself).

package marker *n.* a character which is used in the textual notation for a symbol to separate the package name from the symbol name, and which is *colon* in the *standard readtable*. See Section 2.1 (Character Syntax).

package prefix *n.* a notation preceding the *name* of a *symbol* in text that is processed by the *Lisp reader*, which uses a *package name* followed by one or more *package markers*, and which indicates that the symbol is looked up in the indicated *package*.

package registry *n.* A mapping of *names* to *package objects*. It is possible for there to be a *package object* which is not in this mapping; such a *package* is called an *unregistered package*. *Operators* such as **find-package** consult this mapping in order to find a *package* from its *name*. *Operators* such as **do-all-symbols**, **find-all-symbols**, and **list-all-packages** operate only on *packages* that exist in the *package registry*.

pairwise *adv.* (of an adjective on a set) applying individually to all possible pairings of elements of the set. "The types A, B, and C are pairwise disjoint if A and B are disjoint, B and C are disjoint, and A and C are disjoint."

parallel *adj. Trad.* (of *binding* or *assignment*) done in the style of **psetq**, **let**, or **do**; that is, first evaluating all of the *forms* that produce *values*, and only then *assigning* or *binding* the *variables* (or *places*). Note that this does not imply traditional computational "parallelism" since the *forms* that produce *values* are evaluated *sequentially*. See *sequential*.

parameter *n.* 1. (of a *function*) a *variable* in the definition of a *function* which takes on the *value* of a corresponding *argument* (or of a *list* of corresponding arguments) to that *function* when it is called, or which in some cases is given a default value because there is no corresponding *argument*. 2. (of a *format directive*) an *object* received as data flow by a *format directive* due to a prefix notation within the *format string* at the *format directive*'s point of use. See Section 22.3 (Formatted Output). "In `"~3,'0D"`, the number `3` and the character `#\0` are parameters to the `~D` format directive."

parameter specializer *n.* 1. (of a *method*) an *expression* which constrains the *method* to be applicable only to *argument* sequences in which the corresponding *argument* matches the *parameter specializer*. 2. a *class*, or a *list* `(eql` *object*`)`.

parameter specializer name *n.* 1. (of a *method* definition) an expression used in code to name a *parameter specializer*. See Section 7.6.2 (Introduction to Methods). 2. a *class*, a *symbol* naming a *class*, or a *list* `(eql` *form*`)`.

pathname *n.* an *object* of *type* **pathname**, which is a structured representation of the name of a *file*. A *pathname* has six components: a "host," a "device," a "directory," a "name," a "type," and a "version."

pathname designator *n.* a *designator* for a *pathname*; that is, an *object* that denotes a *pathname* and that is one of: a *pathname namestring* (denoting the corresponding *pathname*), a *stream associated with a file* (denoting the *pathname* used to open the *file*; this may be, but is not required to be, the actual name of the *file*), or a *pathname* (denoting itself). See Section 21.1.1.1.2 (Open and Closed Streams).

physical pathname *n.* a *pathname* that is not a *logical pathname*.

place *n.* 1. a *form* which is suitable for use as a *generalized reference*. 2. the conceptual location referred to by such a $place_1$.

plist [ˈpē͵list] *n.* a *property list*.

portable *adj.* (of *code*) required to produce equivalent results and observable side effects in all *conforming implementations*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

potential copy *n.* (of an *object* O_1 subject to constriants) an *object* O_2 that if the specified constraints are satisfied by O_1 without any modification might or might not be *identical* to O_1, or else that must be a *fresh object* that resembles a *copy* of O_1 except that it has been modified as necessary to satisfy the constraints.

potential number *n.* A textual notation that might be parsed by the *Lisp reader* in some *conforming implementation* as a *number* but is not required to be parsed as a *number*. No *object* is a *potential number*—either an *object* is a *number* or it is not. See Section 2.3.1.1 (Potential Numbers as Tokens).

pprint dispatch table *n.* an *object* that can be the *value* of ***print-pprint-dispatch*** and hence can control how *objects* are printed when ***print-pretty*** is *true*. See Section 22.2.1.4 (Pretty Print Dispatch Tables).

predicate *n.* a *function* that returns a *generalized boolean* as its first value.

present *n.* 1. (of a *feature* in a *Lisp image*) a state of being that is in effect if and only if the *symbol* naming the *feature* is an *element* of the *features list*. 2. (of a *symbol* in a *package*) being accessible in that *package* directly, rather than being inherited from another *package*.

pretty print *v.t.* (an *object*) to invoke the *pretty printer* on the *object*.

pretty printer *n.* the procedure that prints the character representation of an *object* onto a *stream* when the *value* of ***print-pretty*** is *true*, and that uses layout techniques (*e.g.*, indentation) that tend to highlight the structure of the *object* in a way that makes it easier for human readers to parse visually. See the *variable* ***print-pprint-dispatch*** and Section 22.2 (The Lisp Pretty Printer).

pretty printing stream *n.* a *stream* that does pretty printing. Such streams are created by the *function* **pprint-logical-block** as a link between the output stream and the logical block.

primary method *n.* a member of one of two sets of *methods* (the set of *auxiliary methods* is the other) that form an exhaustive partition of the set of *methods* on the *method*'s *generic function*. How these sets are determined is dependent on the *method combination* type; see Section 7.6.2 (Introduction to Methods).

primary value *n.* (of *values* resulting from the *evaluation* of a *form*) the first *value*, if any, or else **nil** if there are no *values*. "The primary value returned by **truncate** is an integer quotient, truncated toward zero."

principal *adj.* (of a value returned by a Common Lisp *function* that implements a mathematically irrational or transcendental function defined in the complex domain) of possibly many (sometimes an infinite number of) correct values for the mathematical function, being the particular *value* which the corresponding Common Lisp *function* has been defined to return.

print name *n. Trad.* (usually of a *symbol*) a $name_3$.

printer control variable *n.* a *variable* whose specific purpose is to control some action of the *Lisp printer*; that is, one of the *variables* in Figure 22–1, or else some *implementation-defined variable* which is defined by the *implementation* to be a *printer control variable*.

printer escaping *n.* The combined state of the *printer control variables* ***print-escape*** and ***print-readably***. If the value of either ***print-readably*** or ***print-escape*** is *true*, then ***printer escaping*** is "enabled"; otherwise (if the values of both ***print-readably*** and ***print-escape*** are *false*), then *printer escaping* is "disabled".

printing *adj.* (of a *character*) being a *graphic character* other than *space*.

process *v.t.* (a *form* by the *compiler*) to perform *minimal compilation*, determining the time of evaluation for a *form*, and possibly *evaluating* that *form* (if required).

processor *n.*, *ANSI* an *implementation*.

proclaim *v.t.* (a *proclamation*) to *establish* that *proclamation*.

proclamation *n.* a *global declaration*.

prog tag *n. Trad.* a *go tag*.

program *n. Trad.* Common Lisp *code*.

programmer *n.* an active entity, typically a human, that writes a *program*, and that might or might not also be a *user* of the *program*.

programmer code *n.* *code* that is supplied by the programmer; that is, *code* that is not *system code*.

proper list *n.* A *list* terminated by the *empty list*. (The *empty list* is a *proper list*.) See *improper list*.

proper name *n.* (of a *class*) a *symbol* that *names* the *class* whose *name* is that *symbol*. See the *functions* **class-name** and **find-class**.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

proper sequence *n.* a *sequence* which is not an *improper list*; that is, a *vector* or a *proper list*.

proper subtype *n.* (of a *type*) a *subtype* of the *type* which is not the *same type* as the *type* (*i.e.*, its *elements* are a "proper subset" of the *type*).

property *n.* (of a *property list*) 1. a conceptual pairing of a *property indicator* and its associated *property value* on a *property list*. 2. a *property value*.

property indicator *n.* (of a *property list*) the *name* part of a *property*, used as a *key* when looking up a *property value* on a *property list*.

property list *n.* 1. a *list* containing an even number of *elements* that are alternating *names* (sometimes called *indicators* or *keys*) and *values* (sometimes called *properties*). When there is more than one *name* and *value* pair with the *identical name* in a *property list*, the first such pair determines the *property*. 2. (of a *symbol*) the component of the *symbol* containing a *property list*.

property value *n.* (of a *property indicator* on a *property list*) the *object* associated with the *property indicator* on the *property list*.

purports to conform *v.* makes a good-faith claim of conformance. This term expresses intention to conform, regardless of whether the goal of that intention is realized in practice. For example, language implementations have been known to have bugs, and while an *implementation* of this specification with bugs might not be a *conforming implementation*, it can still *purport to conform*. This is an important distinction in certain specific cases; *e.g.*, see the *variable* ***features***.

Q

qualified method *n.* a *method* that has one or more *qualifiers*.

qualifier *n.* (of a *method* for a *generic function*) one of possibly several *objects* used to annotate the *method* in a way that identifies its role in the *method combination*. The *method combination type* determines how many *qualifiers* are permitted for each *method*, which *qualifiers* are permitted, and the semantics of those *qualifiers*.

query I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***query-io***.

quoted object *n.* an *object* which is the second element of a **quote** *form*.

R

radix *n.* an *integer* between 2 and 36, inclusive, which can be used to designate a base with respect to which certain kinds of numeric input or output are performed. (There are *n* valid digit characters for any given *radix n*, and those digits are the first *n* digits in the sequence `0`, `1`, ..., `9`, `A`, `B`, ..., `Z`, which have the weights `0`, `1`, ..., `9`, `10`, `11`, ..., `35`, respectively. Case is not significant in parsing numbers of radix greater than `10`, so "9b8a" and "9B8A" denote the same *radix* `16` number.)

random state *n.* an *object* of *type* **random-state**.

rank *n.* a non-negative *integer* indicating the number of *dimensions* of an *array*.

ratio *n.* an *object* of *type* **ratio**.

ratio marker *n.* a character which is used in the textual notation for a *ratio* to separate the numerator from the denominator, and which is *slash* in the *standard readtable*. See Section 2.1 (Character Syntax).

rational *n.* an *object* of *type* **rational**.

read *v.t.* 1. (a *binding* or *slot* or component) to obtain the *value* of the *binding* or *slot*. 2. (an *object* from a *stream*) to parse an *object* from its representation on the *stream*.

readably *adv.* (of a manner of printing an *object* O_1) in such a way as to permit the *Lisp Reader* to later *parse* the printed output into an *object* O_2 that is *similar* to O_1.

reader *n.* 1. a *function* that *reads*$_1$ a *variable* or *slot*. 2. the *Lisp reader*.

reader macro *n.* 1. a textual notation introduced by dispatch on one or two *characters* that defines special-purpose syntax for use by the *Lisp reader*, and that is implemented by a *reader macro function*. See Section 2.2 (Reader Algorithm). 2. the *character* or *characters* that introduce a *reader macro*$_1$; that is, a *macro character* or the conceptual pairing of a *dispatching macro character* and the *character* that follows it. (A *reader macro* is not a kind of *macro*.)

reader macro function *n.* a *function designator* that denotes a *function* that implements a *reader macro*$_2$. See the *functions* **set-macro-character** and **set-dispatch-macro-character**.

readtable *n.* an *object* of *type* **readtable**.

readtable case *n.* an attribute of a *readtable* whose value is a *case sensitivity mode*, and that selects the manner in which *characters* in a *symbol*'s *name* are to be treated by the *Lisp reader* and the *Lisp printer*. See Section 23.1.2 (Effect of Readtable Case on the Lisp Reader) and Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer).

readtable designator *n.* a *designator* for a *readtable*; that is, an *object* that denotes a *readtable* and that is one of: **nil** (denoting the *standard readtable*), or a *readtable* (denoting itself).

recognizable subtype *n.* (of a *type*) a *subtype* of the *type* which can be reliably detected to be such by the *implementation*. See the *function* **subtypep**.

reference *n.*, *v.t.* 1. *n.* an act or occurrence of referring to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment*. 2. *v.t.* to refer to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment*, usually by *name*.

registered package *n.* a *package object* that is installed in the *package registry*. (Every *registered package* has a *name* that is a *string*, as well as zero or more *string* nicknames. All *packages* that are initially specified by Common Lisp or created by **make-package** or **defpackage** are *registered packages*. *Registered packages* can be turned into *unregistered packages* by **delete-package**.)

relative *adj.* 1. (of a *time*) representing an offset from an *absolute time* in the units appropriate to that time. For example, a *relative internal time* is the difference between two *absolute internal times*, and is measured in *internal time units*. 2. (of a *pathname*) representing a position in a directory hierarchy by motion from a position other than the root, which might therefore vary. "The notation `#P"../foo.text"` denotes a relative pathname if the host file system is Unix." See *absolute*.

repertoire *n.*, *ISO* a *subtype* of **character**. See Section 13.1.2.2 (Character Repertoires).

report *n.* (of a *condition*) to *call* the *function* **print-object** on the *condition* in an *environment* where the *value* of ***print-escape*** is *false*.

report message *n.* the text that is output by a *condition reporter*.

required parameter *n.* A *parameter* for which a corresponding positional *argument* must be supplied when *calling* the *function*.

rest list *n.* (of a *function* having a *rest parameter*) The *list* to which the *rest parameter* is *bound* on some particular *call* to the *function*.

rest parameter *n.* A *parameter* which was introduced by **&rest**.

restart *n.* an *object* of *type* **restart**.

restart designator *n.* a *designator* for a *restart*; that is, an *object* that denotes a *restart* and that is one of: a *non-nil symbol* (denoting the most recently established *active restart* whose *name* is that *symbol*), or a *restart* (denoting itself).

restart function *n.* a *function* that invokes a *restart*, as if by **invoke-restart**. The primary purpose of a *restart function* is to provide an alternate interface. By convention, a *restart function* usually has the same name as the *restart* which it invokes. Figure 26–4 shows a list of the *standardized restart functions*.

abort	**muffle-warning**	**use-value**
continue	**store-value**	

Figure 26–4. Standardized Restart Functions

return *v.t.* (of *values*) 1. (from a *block*) to transfer control and *values* from the *block*; that is, to cause the *block* to *yield* the *values* immediately without doing any further evaluation of the *forms* in its body. 2. (from a *form*) to *yield* the *values*.

return value *n. Trad.* a $value_1$

right-parenthesis *n.* the *standard character* ")", that is variously called "right parenthesis" or "close parenthesis" See Figure 2–5.

run time *n.* 1. *load time* 2. *execution time*

run-time compiler *n.* refers to the **compile** function or to *implicit compilation*, for which the compilation and run-time *environments* are maintained in the same *Lisp image*.

run-time definition *n.* a definition in the *run-time environment*.

run-time environment *n.* the *environment* in which a program is *executed*.

S

safe *adj.* 1. (of *code*) processed in a *lexical environment* where the the highest **safety** level (3) was in effect. See **optimize**. 2. (of a *call*) a *safe call*.

safe call *n.* a *call* in which the *call*, the *function* being *called*, and the point of *functional evaluation* are all $safe_1$ *code*. For more detailed information, see Section 3.5.1.1 (Safe and Unsafe Calls).

same *adj.* 1. (of *objects* under a specified *predicate*) indistinguishable by that *predicate*. "The symbol `car`, the string `"car"`, and the string `"CAR"` are the *same* under **string-equal**". 2. (of *objects* if no predicate is implied by context) indistinguishable by **eql**. Note that **eq** might be capable of distinguishing some *numbers* and *characters* which **eql** cannot distinguish, but the nature of such, if any, is *implementation-dependent*. Since **eq** is used only rarely in this specification, **eql** is the default predicate when none is mentioned explicitly. "The conses returned by two successive calls to **cons** are never the same." 3. (of *types*) having the same set of *elements*; that is, each *type* is a *subtype* of the others. "The types specified by `(integer 0 1)`, `(unsigned-byte 1)`, and `bit` are the same."

satisfy the test *v.* (of an *object* being considered by a *sequence function*) 1. (for a one *argument* test) to be in a state such that the *function* which is the *predicate argument* to the *sequence function* returns *true* when given a single *argument* that is the result of calling the *sequence function*'s *key argument* on the *object* being considered. See Section 17.2.2 (Satisfying a One-Argument Test). 2. (for a two *argument* test) to be in a state such that the two-place *predicate* which is the *sequence function*'s *test argument* returns *true* when given a first *argument* that is the *object* being considered, and when given a second *argument* that is the result of calling the *sequence function*'s *key argument* on an *element* of the *sequence function*'s *sequence argument* which is being tested for equality; or to be in a state such that the *test-not function* returns *false* given the same *arguments*. See Section 17.2.1 (Satisfying a Two-Argument Test).

scope *n.* the structural or textual region of code in which *references* to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment* (usually by *name*) can occur.

script *n. ISO* one of possibly several sets that form an *exhaustive partition* of the type **character**. See Section 13.1.2.1 (Character Scripts).

secondary value *n.* (of *values* resulting from the *evaluation* of a *form*) the second *value*, if any, or else **nil** if there are fewer than two *values*. "The secondary value returned by **truncate** is a remainder."

section *n.* a partitioning of output by a *conditional newline* on a *pretty printing stream*. See Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

self-evaluating object *n.* an *object* that is neither a *symbol* nor a *cons*. If a *self-evaluating object* is *evaluated*, it *yields* itself as its only *value*. "Strings are self-evaluating objects."

semi-standard *adj.* (of a language feature) not required to be implemented by any *conforming implementation*, but nevertheless recommended as the canonical approach in situations where an *implementation* does plan to support such a feature. The presence of *semi-standard* aspects in the language is intended to lessen portability problems and reduce the risk of gratuitous divergence among *implementations* that might stand in the way of future standardization.

semicolon *n.* the *standard character* that is called "semicolon" (`;`). See Figure 2–5.

sequence *n.* 1. an ordered collection of elements 2. a *vector* or a *list*.

sequence function *n.* one of the *functions* in Figure 17–1, or an *implementation-defined function* that operates on one or more *sequences*. and that is defined by the *implementation* to be a *sequence function*.

sequential *adj. Trad.* (of *binding* or *assignment*) done in the style of **setq**, **let***, or **do***; that is, interleaving the evaluation of the *forms* that produce *values* with the *assignments* or *bindings* of the *variables* (or *places*). See *parallel*.

sequentially *adv.* in a *sequential* way.

serious condition *n.* a *condition* of *type* **serious-condition**, which represents a *situation* that is generally sufficiently severe that entry into the *debugger* should be expected if the *condition* is *signaled* but not *handled*.

session *n.* the conceptual aggregation of events in a *Lisp image* from the time it is started to the time it is terminated.

set *v.t. Trad.* (any *variable* or a *symbol* that is the *name* of a *dynamic variable*) to *assign* the *variable*.

setf expander *n.* a function used by **setf** to compute the *setf expansion* of a *place*.

setf expansion *n.* a set of five *expressions*$_1$ that, taken together, describe how to store into a *place* and which *subforms* of the macro call associated with the *place* are evaluated. See Section 5.1.1.2 (Setf Expansions).

setf function *n.* a *function* whose *name* is `(setf` *symbol*`)`.

setf function name *n.* (of a *symbol* S) the *list* `(setf` S`)`.

shadow *v.t.* 1. to override the meaning of. "That binding of `X` shadows an outer one." 2. to hide the presence of. "That **macrolet** of `F` shadows the outer **flet** of `F`." 3. to replace. "That package shadows the symbol `cl:car` with its own symbol `car`."

shadowing symbol *n.* (in a *package*) an *element* of the *package*'s *shadowing symbols list*.

shadowing symbols list *n.* (of a *package*) a *list*, associated with the *package*, of *symbols* that are to be exempted from 'symbol conflict errors' detected when packages are *used*. See the *function* **package-shadowing-symbols**.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

shared slot *n.* (of a *class*) a *slot accessible* in more than one *instance* of a *class*; specifically, such a *slot* is *accessible* in all *direct instances* of the *class* and in those *indirect instances* whose *class* does not *shadow*$_1$ the *slot*.

sharpsign *n.* the *standard character* that is variously called "number sign," "sharp," or "sharp sign" (`#`). See Figure 2–5.

short float *n.* an *object* of *type* **short-float**.

sign *n.* one of the *standard characters* "`+`" or "`-`".

signal *v.* to announce, using a standard protocol, that a particular situation, represented by a *condition*, has been detected. See Section 9.1 (Condition System Concepts).

signature *n.* (of a *method*) a description of the *parameters* and *parameter specializers* for the *method* which determines the *method*'s applicability for a given set of required *arguments*, and which also describes the *argument* conventions for its other, non-required *arguments*.

similar *adj.* (of two *objects*) defined to be equivalent under the *similarity* relationship.

similarity *n.* a two-place conceptual equivalence predicate, which is independent of the *Lisp image* so that two *objects* in different *Lisp images* can be understood to be equivalent under this predicate. See Section 3.2.4 (Literal Objects in Compiled Files).

simple *adj.* 1. (of an *array*) being of *type* **simple-array**. 2. (of a *character*) having no *implementation-defined attributes*, or else having *implementation-defined attributes* each of which has the *null* value for that *attribute*.

simple array *n.* an *array* of *type* **simple-array**.

simple bit array *n.* a *bit array* that is a *simple array*; that is, an *object* of *type* `(simple-array bit)`.

simple bit vector *n.* a *bit vector* of *type* **simple-bit-vector**.

simple condition *n.* a *condition* of *type* **simple-condition**.

simple general vector *n.* a *simple vector*.

simple string *n.* a *string* of *type* **simple-string**.

simple vector *n.* a *vector* of *type* **simple-vector**, sometimes called a "*simple general vector*." Not all *vectors* that are *simple* are *simple vectors*—only those that have *element type* `t`.

single escape *n.*, *adj.* 1. *n.* the *syntax type* of a *character* that indicates that the next *character* is to be treated as an *alphabetic*$_2$ *character* with its *case* preserved. For details, see Section 2.1.4.6 (Single Escape Character). 2. *adj.* (of a *character*) having the *single escape syntax type*. 3. *n.* a *single escape*$_2$ *character*. (In the *standard readtable*, *slash* is the only *single escape*.)

single float *n.* an *object* of *type* **single-float**.

single-quote *n.* the *standard character* that is variously called "apostrophe," "acute accent," "quote," or "single quote" (`'`). See Figure 2–5.

singleton *adj.* (of a *sequence*) having only one *element*. "`(list 'hello)` returns a singleton list."

situation *n.* the *evaluation* of a *form* in a specific *environment*.

slash *n.* the *standard character* that is variously called "solidus" or "slash" (`/`). See Figure 2–5.

slot *n.* a component of an *object* that can store a *value*.

slot specifier *n.* a representation of a *slot* that includes the *name* of the *slot* and zero or more *slot* options. A *slot* option pertains only to a single *slot*.

source code *n.* *code* representing *objects* suitable for *evaluation* (*e.g.*, *objects* created by **read**, by *macro expansion*, or by *compiler macro expansion*).

source file *n.* a *file* which contains a textual representation of *source code*, that can be edited, *loaded*, or *compiled*.

space *n.* the *standard character* ⟨*Space*⟩, notated for the *Lisp reader* as `#\Space`.

special form *n.* a *list*, other than a *macro form*, which is a *form* with special syntax or special *evaluation* rules or both, possibly manipulating the *evaluation environment* or control flow or both. The first element of a *special form* is a *special operator*.

special operator *n.* one of a fixed set of *symbols*, enumerated in Figure 3–2, that may appear in the *car* of a *form* in order to identify the *form* as a *special form*.

special variable *n.* *Trad.* a *dynamic variable*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

specialize *v.t.* (a *generic function*) to define a *method* for the *generic function*, or in other words, to refine the behavior of the *generic function* by giving it a specific meaning for a particular set of *classes* or *arguments*.

specialized *adj.* 1. (of a *generic function*) having *methods* which *specialize* the *generic function*. 2. (of an *array*) having an *actual array element type* that is a *proper subtype* of the *type* `t`; see Section 15.1.1 (Array Elements). "`(make-array 5 :element-type 'bit)` makes an array of length five that is specialized for bits."

specialized lambda list *n.* an *extended lambda list* used in *forms* that *establish method* definitions, such as **defmethod**. See Section 3.4.3 (Specialized Lambda Lists).

spreadable argument list designator *n.* a *designator* for a *list* of *objects*; that is, an *object* that denotes a *list* and that is a *non-null list* $L1$ of length n, whose last element is a *list* $L2$ of length m (denoting a list $L3$ of length $m+n-1$ whose *elements* are $L1_i$ for $i < n-1$ followed by $L2_j$ for $j < m$). "The list (1 2 (3 4 5)) is a spreadable argument list designator for the list (1 2 3 4 5)."

stack allocate *v.t.* *Trad.* to allocate in a non-permanent way, such as on a stack. Stack-allocation is an optimization technique used in some *implementations* for allocating certain kinds of *objects* that have *dynamic extent*. Such *objects* are allocated on the stack rather than in the heap so that their storage can be freed as part of unwinding the stack rather than taking up space in the heap until the next garbage collection. What *types* (if any) can have *dynamic extent* can vary from *implementation* to *implementation*. No *implementation* is ever required to perform stack-allocation.

stack-allocated *adj.* *Trad.* having been *stack allocated*.

standard character *n.* a *character* of *type* **standard-char**, which is one of a fixed set of 96 such *characters* required to be present in all *conforming implementations*. See Section 2.1.3 (Standard Characters).

standard class *n.* a *class* that is a *generalized instance* of *class* **standard-class**.

standard generic function a *function* of *type* **standard-generic-function**.

standard input *n.* the *input stream* which is the *value* of the *dynamic variable* ***standard-input***.

standard method combination *n.* the *method combination* named **standard**.

standard object *n.* an *object* that is a *generalized instance* of *class* **standard-object**.

standard output *n.* the *output stream* which is the *value* of the *dynamic variable* ***standard-output***.

standard pprint dispatch table *n.* A *pprint dispatch table* that is *different* from the *initial pprint dispatch table*, that implements *pretty printing* as described in this specification, and that, unlike other *pprint dispatch tables*, must never be modified by any program. (Although the definite reference "the *standard pprint dispatch table*" is generally used within this document, it is actually *implementation-dependent* whether a single *object* fills the role of the *standard pprint dispatch table*, or whether there might be multiple such objects, any one of which could be used on any given occasion where "the *standard pprint dispatch table*" is called for. As such, this phrase should be seen as an indefinite reference in all cases except for anaphoric references.)

standard readtable *n.* A *readtable* that is *different* from the *initial readtable*, that implements the *expression* syntax defined in this specification, and that, unlike other *readtables*, must never be modified by any program. (Although the definite reference "the *standard readtable*" is generally used within this document, it is actually *implementation-dependent* whether a single *object* fills the role of the *standard readtable*, or whether there might be multiple such objects, any one of which could be used on any given occasion where "the *standard readtable*" is called for. As such, this phrase should be seen as an indefinite reference in all cases except for anaphoric references.)

standard syntax *n.* the syntax represented by the *standard readtable* and used as a reference syntax throughout this document. See Section 2.1 (Character Syntax).

standardized *adj.* (of a *name*, *object*, or definition) having been defined by Common Lisp. "All standardized variables that are required to hold bidirectional streams have "`-io*`" in their name."

startup environment *n.* the *global environment* of the running *Lisp image* from which the *compiler* was invoked.

step *v.t.*, *n.* 1. *v.t.* (an iteration *variable*) to *assign* the *variable* a new *value* at the end of an iteration, in preparation for a new iteration. 2. *n.* the *code* that identifies how the next value in an iteration is to be computed. 3. *v.t.* (*code*) to specially execute the *code*, pausing at intervals to allow user confirmation or intervention, usually for debugging.

stream *n.* an *object* that can be used with an input or output function to identify an appropriate source or sink of *characters* or *bytes* for that operation.

stream associated with a file *n.* a *file stream*, or a *synonym stream* the *target* of which is a *stream associated with a file*. Such a *stream* cannot be created with **make-two-way-stream**, **make-echo-stream**,

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

make-broadcast-stream, **make-concatenated-stream**, **make-string-input-stream**, or **make-string-output-stream**.

stream designator *n.* a *designator* for a *stream*; that is, an *object* that denotes a *stream* and that is one of: **t** (denoting the *value* of ***terminal-io***), **nil** (denoting the *value* of ***standard-input*** for *input stream designators* or denoting the *value* of ***standard-output*** for *output stream designators*), or a *stream* (denoting itself).

stream element type *n.* (of a *stream*) the *type* of data for which the *stream* is specialized.

stream variable *n.* a *variable* whose *value* must be a *stream*.

stream variable designator *n.* a *designator* for a *stream variable*; that is, a *symbol* that denotes a *stream variable* and that is one of: **t** (denoting ***terminal-io***), **nil** (denoting ***standard-input*** for *input stream variable designators* or denoting ***standard-output*** for *output stream variable designators*), or some other *symbol* (denoting itself).

string *n.* a specialized *vector* that is of *type* **string**, and whose elements are of *type* **character** or a *subtype* of *type* **character**.

string designator *n.* a *designator* for a *string*; that is, an *object* that denotes a *string* and that is one of: a *character* (denoting a *singleton string* that has the *character* as its only *element*), a *symbol* (denoting the *string* that is its *name*), or a *string* (denoting itself). The intent is that this term be consistent with the behavior of **string**; *implementations* that extend **string** must extend the meaning of this term in a compatible way.

string equal *adj.* the *same* under **string-equal**.

string stream *n.* a *stream* of *type* **string-stream**.

structure *n.* an *object* of *type* **structure-object**.

structure class *n.* a *class* that is a *generalized instance* of *class* **structure-class**.

structure name *n.* a *name* defined with **defstruct**. Usually, such a *type* is also a *structure class*, but there may be *implementation-dependent* situations in which this is not so, if the `:type` option to **defstruct** is used.

style warning *n.* a *condition* of *type* **style-warning**.

subclass *n.* a *class* that *inherits* from another *class*, called a *superclass*. (No *class* is a *subclass* of itself.)

subexpression *n.* (of an *expression*) an *expression* that is contained within the *expression*. (In fact, the state of being a *subexpression* is not an attribute of the *subexpression*, but really an attribute of the containing *expression* since the *same object* can at once be a *subexpression* in one context, and not in another.)

subform *n.* (of a *form*) an *expression* that is a *subexpression* of the *form*, and which by virtue of its position in that *form* is also a *form*. "`(f x)` and `x`, but not `exit`, are subforms of `(return-from exit (f x))`."

subrepertoire *n.* a subset of a *repertoire*.

subtype *n.* a *type* whose membership is the same as or a proper subset of the membership of another *type*, called a *supertype*. (Every *type* is a *subtype* of itself.)

superclass *n.* a *class* from which another *class* (called a *subclass*) *inherits*. (No *class* is a *superclass* of itself.) See *subclass*.

supertype *n.* a *type* whose membership is the same as or a proper superset of the membership of another *type*, called a *subtype*. (Every *type* is a *supertype* of itself.) See *subtype*.

supplied-p parameter *n.* a *parameter* which recieves its *generalized boolean* value implicitly due to the presence or absence of an *argument* corresponding to another *parameter* (such as an *optional parameter* or a *rest parameter*). See Section 3.4.1 (Ordinary Lambda Lists).

symbol *n.* an *object* of *type* **symbol**.

symbol macro *n.* a *symbol* that stands for another *form*. See the *macro* **symbol-macrolet**.

synonym stream *n.* 1. a *stream* of *type* **synonym-stream**, which is consequently a *stream* that is an alias for another *stream*, which is the *value* of a *dynamic variable* whose *name* is the *synonym stream symbol* of the *synonym stream*. See the *function* **make-synonym-stream**. 2. (to a *stream*) a *synonym stream* which has the *stream* as the *value* of its *synonym stream symbol*. 3. (to a *symbol*) a *synonym stream* which has the *symbol* as its *synonym stream symbol*.

synonym stream symbol *n.* (of a *synonym stream*) the *symbol* which names the *dynamic variable* which has as its *value* another *stream* for which the *synonym stream* is an alias.

syntax type *n.* (of a *character*) one of several classifications, enumerated in Figure 2–6, that are used for dispatch during parsing by the *Lisp reader*. See Section 2.1.4 (Character Syntax Types).

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

system class *n.* a *class* that may be of *type* **built-in-class** in a *conforming implementation* and hence cannot be inherited by *classes* defined by *conforming programs*.

system code *n.* *code* supplied by the *implementation* to implement this specification (*e.g.*, the definition of **mapcar**) or generated automatically in support of this specification (*e.g.*, during method combination); that is, *code* that is not *programmer code*.

T

t *n.* 1. a. the *boolean* representing true. b. the canonical *generalized boolean* representing true. (Although any *object* other than **nil** is considered *true* as a *generalized boolean*, `t` is generally used when there is no special reason to prefer one such *object* over another.) 2. the *name* of the *type* to which all *objects* belong—the *supertype* of all *types* (including itself). 3. the *name* of the *superclass* of all *classes* except itself.

tag *n.* 1. a *catch tag*. 2. a *go tag*.

tail *n.* (of a *list*) an *object* that is the *same* as either some *cons* which makes up that *list* or the *atom* (if any) which terminates the *list*. "The empty list is a tail of every proper list."

target *n.* 1. (of a *constructed stream*) a *constituent* of the *constructed stream*. "The target of a synonym stream is the value of its synonym stream symbol." 2. (of a *displaced array*) the *array* to which the *displaced array* is displaced. (In the case of a chain of *constructed streams* or *displaced arrays*, the unqualified term "*target*" always refers to the immediate *target* of the first item in the chain, not the immediate target of the last item.)

terminal I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***terminal-io***.

terminating *n.* (of a *macro character*) being such that, if it appears while parsing a token, it terminates that token. See Section 2.2 (Reader Algorithm).

tertiary value *n.* (of *values* resulting from the *evaluation* of a *form*) the third *value*, if any, or else **nil** if there are fewer than three *values*.

throw *v.* to transfer control and *values* to a *catch*. See the *special operator* **throw**.

tilde *n.* the *standard character* that is called "tilde" (~). See Figure 2–5.

time a representation of a point (*absolute time*) or an interval (*relative time*) on a time line. See *decoded time*, *internal time*, and *universal time*.

time zone *n.* a *rational* multiple of `1/3600` between `-24` (inclusive) and `24` (inclusive) that represents a time zone as a number of hours offset from Greenwich Mean Time. Time zone values increase with motion to the west, so Massachusetts, U.S.A. is in time zone `5`, California, U.S.A. is time zone `8`, and Moscow, Russia is time zone *-3*. (When "daylight savings time" is separately represented as an *argument* or *return value*, the *time zone* that accompanies it does not depend on whether daylight savings time is in effect.)

token *n.* a textual representation for a *number* or a *symbol*. See Section 2.3 (Interpretation of Tokens).

top level form *n.* a *form* which is processed specially by **compile-file** for the purposes of enabling *compile time evaluation* of that *form*. *Top level forms* include those *forms* which are not *subforms* of any other *form*, and certain other cases. See Section 3.2.3.1 (Processing of Top Level Forms).

trace output *n.* the *output stream* which is the *value* of the *dynamic variable* ***trace-output***.

tree *n.* 1. a binary recursive data structure made up of *conses* and *atoms*: the *conses* are themselves also *trees* (sometimes called "subtrees" or "branches"), and the *atoms* are terminal nodes (sometimes called *leaves*). Typically, the *leaves* represent data while the branches establish some relationship among that data. 2. in general, any recursive data structure that has some notion of "branches" and *leaves*.

tree structure *n.* (of a $tree_1$) the set of *conses* that make up the *tree*. Note that while the car_{1b} component of each such *cons* is part of the *tree structure*, the *objects* that are the $cars_2$ of each *cons* in the *tree* are not themselves part of its *tree structure* unless they are also *conses*.

true *n.* any *object* that is not *false* and that is used to represent the success of a *predicate* test. See t_1.

truename *n.* 1. the canonical *filename* of a *file* in the *file system*. See Section 20.1.3 (Truenames). 2. a *pathname* representing a $truename_1$.

two-way stream *n.* a *stream* of *type* **two-way-stream**, which is a *bidirectional composite stream* that receives its input from an associated *input stream* and sends its output to an associated *output stream*.

type *n.* 1. a set of *objects*, usually with common structure, behavior, or purpose. (Note that the expression "X is of type S_a" naturally implies that "X is of type S_b" if S_a is a *subtype* of S_b.) 2. (immediately following the name of a *type*) a *subtype* of that *type*. "The type **vector** is an array type."

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

type declaration *n.* a *declaration* that asserts that every reference to a specified *binding* within the scope of the *declaration* results in some *object* of the specified *type*.

type equivalent *adj.* (of two *types* X and Y) having the same *elements*; that is, X is a *subtype* of Y and Y is a *subtype* of X.

type expand *n.* to fully expand a *type specifier*, removing any references to *derived types*. (Common Lisp provides no program interface to cause this to occur, but the semantics of Common Lisp are such that every *implementation* must be able to do this internally, and some situations involving *type specifiers* are most easily described in terms of a fully expanded *type specifier*.)

type specifier *n.* an *expression* that denotes a *type*. "The symbol `random-state`, the list `(integer 3 5)`, the list `(and list (not null))`, and the class named `standard-class` are type specifiers."

U

unbound *adj.* not having an associated denotation in a *binding*. See *bound*.

unbound variable *n.* a *name* that is syntactically plausible as the name of a *variable* but which is not *bound* in the *variable namespace*.

undefined function *n.* a *name* that is syntactically plausible as the name of a *function* but which is not *bound* in the *function namespace*.

unintern *v.t.* (a *symbol* in a *package*) to make the *symbol* not be *present* in that *package*. (The *symbol* might continue to be *accessible* by inheritance.)

uninterned *adj.* (of a *symbol*) not *accessible* in any *package*; *i.e.*, not $interned_1$.

universal time *n.* *time*, represented as a non-negative *integer* number of seconds. *Absolute universal time* is measured as an offset from the beginning of the year 1900 (ignoring *leap seconds*). See Section 25.1.4.2 (Universal Time).

unqualified method *n.* a *method* with no *qualifiers*.

unregistered package *n.* a *package object* that is not present in the *package registry*. An *unregistered package* has no *name*; *i.e.*, its *name* is **nil**. See the *function* **delete-package**.

unsafe *adj.* (of *code*) not *safe*. (Note that, unless explicitly specified otherwise, if a particular kind of error checking is guaranteed only in a *safe* context, the same checking might or might not occur in that context if it were *unsafe*; describing a context as *unsafe* means that certain kinds of error checking are not reliably enabled but does not guarantee that error checking is definitely disabled.)

unsafe call *n.* a *call* that is not a *safe call*. For more detailed information, see Section 3.5.1.1 (Safe and Unsafe Calls).

upgrade *v.t.* (a declared *type* to an actual *type*) 1. (when creating an *array*) to substitute an *actual array element type* for an *expressed array element type* when choosing an appropriately *specialized array* representation. See the *function* **upgraded-array-element-type**. 2. (when creating a *complex*) to substitute an *actual complex part type* for an *expressed complex part type* when choosing an appropriately *specialized complex* representation. See the *function* **upgraded-complex-part-type**.

upgraded array element type *n.* (of a *type*) a *type* that is a *supertype* of the *type* and that is used instead of the *type* whenever the *type* is used as an *array element type* for object creation or type discrimination. See Section 15.1.2.1 (Array Upgrading).

upgraded complex part type *n.* (of a *type*) a *type* that is a *supertype* of the *type* and that is used instead of the *type* whenever the *type* is used as a *complex part type* for object creation or type discrimination. See the *function* **upgraded-complex-part-type**.

uppercase *adj.* (of a *character*) being among *standard characters* corresponding to the capital letters `A` through `Z`, or being some other *implementation-defined character* that is defined by the *implementation* to be *uppercase*. See Section 13.1.4.3 (Characters With Case).

use *v.t.* (a *package* P_1) to *inherit* the *external symbols* of P_1. (If a package P_2 uses P_1, the *external symbols* of P_1 become *internal symbols* of P_2 unless they are explicitly *exported*.) "The package `CL-USER` uses the package `CL`."

use list *n.* (of a *package*) a (possibly empty) *list* associated with each *package* which determines what other *packages* are currently being *used* by that *package*.

user *n.* an active entity, typically a human, that invokes or interacts with a *program* at run time, but that is not necessarily a *programmer*.

V

valid array dimension *n.* a *fixnum* suitable for use as an *array dimension*. Such a *fixnum* must be greater than or equal to zero, and less than the *value* of **array-dimension-limit**. When multiple *array dimensions* are to be used together to specify a multi-dimensional *array*, there is also an implied constraint that the product of all of the *dimensions* be less than the *value* of **array-total-size-limit**.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

valid array index *n.* (of an *array*) a *fixnum* suitable for use as one of possibly several indices needed to name an *element* of the *array* according to a multi-dimensional Cartesian coordinate system. Such a *fixnum* must be greater than or equal to zero, and must be less than the corresponding *dimension*$_1$ of the *array*. (Unless otherwise explicitly specified, the phrase "a *list* of *valid array indices*" further implies that the *length* of the *list* must be the same as the *rank* of the *array*.) "For a `2` by `3` array, valid array indices for the first dimension are `0` and `1`, and valid array indices for the second dimension are `0`, `1` and `2`."

valid array row-major index *n.* (of an *array*, which might have any number of *dimensions*$_2$) a single *fixnum* suitable for use in naming any *element* of the *array*, by viewing the array's storage as a linear series of *elements* in row-major order. Such a *fixnum* must be greater than or equal to zero, and less than the *array total size* of the *array*.

valid fill pointer *n.* (of an *array*) a *fixnum* suitable for use as a *fill pointer* for the *array*. Such a *fixnum* must be greater than or equal to zero, and less than or equal to the *array total size* of the *array*.

valid logical pathname host *n.* a *string* that has been defined as the name of a *logical host*. See the *function* **load-logical-pathname-translations**.

valid pathname device *n.* a *string*, **nil**, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid pathname device*.

valid pathname directory *n.* a *string*, a *list* of *strings*, **nil**, `:wild`, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid directory component*.

valid pathname host *n.* a *valid physical pathname host* or a *valid logical pathname host*.

valid pathname name *n.* a *string*, **nil**, `:wild`, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid pathname name*.

valid pathname type *n.* a *string*, **nil**, `:wild`, `:unspecific`.

valid pathname version *n.* a non-negative *integer*, or one of `:wild`, `:newest`, `:unspecific`, or **nil**. The symbols `:oldest`, `:previous`, and `:installed` are *semi-standard* special version symbols.

valid physical pathname host *n.* any of a *string*, a *list* of *strings*, or the symbol `:unspecific`, that is recognized by the implementation as the name of a host.

valid sequence index *n.* (of a *sequence*) an *integer* suitable for use to name an *element* of the *sequence*. Such an *integer* must be greater than or equal to zero, and must be less than the *length* of the *sequence*. (If the *sequence* is an *array*, the *valid sequence index* is further constrained to be a *fixnum*.)

value *n.* 1. a. one of possibly several *objects* that are the result of an *evaluation*. b. (in a situation where exactly one value is expected from the *evaluation* of a *form*) the *primary value* returned by the *form*. c. (of *forms* in an *implicit progn*) one of possibly several *objects* that result from the *evaluation* of the last *form*, or **nil** if there are no *forms*. 2. an *object* associated with a *name* in a *binding*. 3. (of a *symbol*) the *value* of the *dynamic variable* named by that symbol. 4. an *object* associated with a *key* in an *association list*, a *property list*, or a *hash table*.

value cell *n. Trad.* (of a *symbol*) The *place* which holds the *value*, if any, of the *dynamic variable* named by that *symbol*, and which is *accessed* by **symbol-value**. See *cell*.

variable *n.* a *binding* in the "variable" *namespace*. See Section 3.1.2.1.1 (Symbols as Forms).

vector *n.* a one-dimensional *array*.

vertical-bar *n.* the *standard character* that is called "vertical bar" (|). See Figure 2–5.

W

whitespace *n.* 1. one or more *characters* that are either the *graphic character* `#\Space` or else *non-graphic* characters such as `#\Newline` that only move the print position. 2. a. *n.* the *syntax type* of a *character* that is a *token* separator. For details, see Section 2.1.4.7 (Whitespace Characters). b. *adj.* (of a *character*) having the *whitespace*$_{2a}$ *syntax type*$_2$. c. *n.* a *whitespace*$_{2b}$ *character*.

wild *adj.* 1. (of a *namestring*) using an *implementation-defined* syntax for naming files, which might "match" any of possibly several possible *filenames*, and which can therefore be used to refer to the aggregate of the *files* named by those *filenames*. 2. (of a *pathname*) a structured representation of a name which might "match" any of possibly several *pathnames*, and which can therefore be used to refer to the aggregate of the *files* named by those *pathnames*. The set of *wild pathnames* includes, but is not restricted to, *pathnames* which have a component which is `:wild`, or which have a directory component which contains `:wild` or `:wild-inferors`. See the *function* **wild-pathname-p**.

write *v.t.* 1. (a *binding* or *slot* or component) to change the *value* of the *binding* or *slot*. 2. (an *object* to a *stream*) to output a representation of the *object* to the *stream*.

writer *n.* a *function* that *writes*$_1$ a *variable* or *slot*.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

Y

yield *v.t.* (*values*) to produce the *values* as the result of *evaluation*. "The form `(+ 2 3)` yields `5`."