Version 15.17R, X3J13/94-101R.
Fri 12-Aug-1994 6:35pm EDT

Programming Language—Common Lisp

4. Types and Classes

4.1 Introduction

A *type* is a (possibly infinite) set of *objects*. An *object* can belong to more than one *type*. *Types* are never explicitly represented as *objects* by Common Lisp. Instead, they are referred to indirectly by the use of *type specifiers*, which are *objects* that denote *types*.

New *types* can be defined using **deftype**, **defstruct**, **defclass**, and **define-condition**.

The *function* **typep**, a set membership test, is used to determine whether a given *object* is of a given *type*. The function **subtypep**, a subset test, is used to determine whether a given *type* is a *subtype* of another given *type*. The function **type-of** returns a particular *type* to which a given *object* belongs, even though that *object* must belong to one or more other *types* as well. (For example, every *object* is of *type* **t**, but **type-of** always returns a *type specifier* for a *type* more specific than **t**.)

Objects, not *variables*, have *types*. Normally, any *variable* can have any *object* as its *value*. It is possible to declare that a *variable* takes on only values of a given *type* by making an explicit *type declaration*. *Types* are arranged in a directed acyclic graph, except for the presence of equivalences.

Declarations can be made about *types* using **declare**, **proclaim**, **declaim**, or **the**. For more information about *declarations*, see Section 3.3 (Declarations).

Among the fundamental *objects* of the object system are *classes*. A *class* determines the structure and behavior of a set of other *objects*, which are called its *instances*. Every *object* is a *direct instance* of a *class*. The *class* of an *object* determines the set of operations that can be performed on the *object*. For more information, see Section 4.3 (Classes).

It is possible to write *functions* that have behavior *specialized* to the class of the *objects* which are their *arguments*. For more information, see Section 7.6 (Generic Functions and Methods).

The *class* of the *class* of an *object* is called its ***metaclass***. For more information about *metaclasses*, see Section 7.4 (Meta-Objects).

4.2 Types

4.2.1 Data Type Definition

Information about *type* usage is located in the sections specified in Figure 4–1. Figure 4–7 lists some *classes* that are particularly relevant to the object system. Figure 9–1 lists the defined *condition types*.

Section	Data Type
Section 4.3 (Classes)	Object System types
Section 7.5 (Slots)	Object System types
Chapter 7 (Objects)	Object System types
Section 7.6 (Generic Functions and Methods)	Object System types
Section 9.1 (Condition System Concepts)	Condition System types
Chapter 4 (Types and Classes)	Miscellaneous types
Chapter 2 (Syntax)	All types—read and print syntax
Section 22.1 (The Lisp Printer)	All types—print syntax
Section 3.2 (Compilation)	All types—compilation issues

Figure 4–1. Cross-References to Data Type Information

4.2.2 Type Relationships

- The *types* **cons**, **symbol**, **array**, **number**, **character**, **hash-table**, **function**, **readtable**, **package**, **pathname**, **stream**, **random-state**, **condition**, **restart**, and any single other *type* created by **defstruct**, **define-condition**, or **defclass** are *pairwise disjoint*, except for type relations explicitly established by specifying *superclasses* in **defclass** or **define-condition** or the `:include` option of **destruct**.

- Any two *types* created by **defstruct** are *disjoint* unless one is a *supertype* of the other by virtue of the **defstruct** `:include` option.

- Any two *distinct classes* created by **defclass** or **define-condition** are *disjoint* unless they have a common *subclass* or one *class* is a *subclass* of the other.

- An implementation may be extended to add other *subtype* relationships between the specified *types*, as long as they do not violate the type relationships and disjointness requirements specified here. An implementation may define additional *types* that are *subtypes* or *supertypes* of any specified *types*, as long as each additional *type* is a *subtype* of *type* **t** and a *supertype* of *type* **nil** and the disjointness requirements are not violated.

At the discretion of the implementation, either **standard-object** or **structure-object** might appear in any class precedence list for a *system class* that does not already specify either **standard-object** or **structure-object**. If it does, it must precede the *class* **t** and follow all other *standardized classes*.

4.2.3 Type Specifiers

Type specifiers can be *symbols*, *classes*, or *lists*. Figure 4–2 lists *symbols* that are *standardized atomic type specifiers*, and Figure 4–3 lists *standardized compound type specifier names*. For syntax information, see the dictionary entry for the corresponding *type specifier*. It is possible to define new *type specifiers* using **defclass**, **define-condition**, **defstruct**, or **deftype**.

arithmetic-error	**function**	**simple-condition**
array	**generic-function**	**simple-error**
atom	**hash-table**	**simple-string**
base-char	**integer**	**simple-type-error**
base-string	**keyword**	**simple-vector**
bignum	**list**	**simple-warning**
bit	**logical-pathname**	**single-float**
bit-vector	**long-float**	**standard-char**
broadcast-stream	**method**	**standard-class**
built-in-class	**method-combination**	**standard-generic-function**
cell-error	**nil**	**standard-method**
character	**null**	**standard-object**
class	**number**	**storage-condition**
compiled-function	**package**	**stream**
complex	**package-error**	**stream-error**
concatenated-stream	**parse-error**	**string**
condition	**pathname**	**string-stream**
cons	**print-not-readable**	**structure-class**
control-error	**program-error**	**structure-object**
division-by-zero	**random-state**	**style-warning**
double-float	**ratio**	**symbol**
echo-stream	**rational**	**synonym-stream**
end-of-file	**reader-error**	**t**
error	**readtable**	**two-way-stream**
extended-char	**real**	**type-error**
file-error	**restart**	**unbound-slot**
file-stream	**sequence**	**unbound-variable**
fixnum	**serious-condition**	**undefined-function**
float	**short-float**	**unsigned-byte**
floating-point-inexact	**signed-byte**	**vector**
floating-point-invalid-operation	**simple-array**	**warning**
floating-point-overflow	**simple-base-string**	
floating-point-underflow	**simple-bit-vector**	

Figure 4–2. Standardized Atomic Type Specifiers

If a *type specifier* is a *list*, the *car* of the *list* is a *symbol*, and the rest of the *list* is subsidiary *type* information. Such a *type specifier* is called a ***compound type specifier***. Except as explicitly stated otherwise, the subsidiary items can be unspecified. The unspecified subsidiary items are indicated by writing *. For example, to completely specify a *vector*, the *type* of the elements and the length of the *vector* must be present.

```
(vector double-float 100)
```

The following leaves the length unspecified:

```
(vector double-float *)
```

The following leaves the element type unspecified:

```
(vector * 100)
```

Suppose that two *type specifiers* are the same except that the first has a * where the second has a more explicit specification. Then the second denotes a *subtype* of the *type* denoted by the first.

If a *list* has one or more unspecified items at the end, those items can be dropped. If dropping all occurrences of * results in a *singleton list*, then the parentheses can be dropped as well (the list can be replaced by the *symbol* in its *car*). For example, `(vector double-float *)` can be abbreviated to `(vector double-float)`, and `(vector * *)` can be abbreviated to `(vector)` and then to `vector`.

and	**long-float**	**simple-base-string**
array	**member**	**simple-bit-vector**
base-string	**mod**	**simple-string**
bit-vector	**not**	**simple-vector**
complex	**or**	**single-float**
cons	**rational**	**string**
double-float	**real**	**unsigned-byte**
eql	**satisfies**	**values**
float	**short-float**	**vector**
function	**signed-byte**	
integer	**simple-array**	

Figure 4–3. Standardized Compound Type Specifier Names

Figure 4–4 show the *defined names* that can be used as *compound type specifier names* but that cannot be used as *atomic type specifiers*.

and	**mod**	**satisfies**
eql	**not**	**values**
member	**or**	

Figure 4–4. Standardized Compound-Only Type Specifier Names

New *type specifiers* can come into existence in two ways.

- Defining a structure by using **defstruct** without using the `:type` specifier or defining a *class* by using **defclass** or **define-condition** automatically causes the name of the structure or class to be a new *type specifier symbol*.
- **deftype** can be used to define ***derived type specifiers***, which act as 'abbreviations' for other *type specifiers*.

A *class object* can be used as a *type specifier*. When used this way, it denotes the set of all members of that *class*.

Figure 4–5 shows some *defined names* relating to *types* and *declarations*.

coerce	**defstruct**	**subtypep**
declaim	**deftype**	**the**
declare	**ftype**	**type**
defclass	**locally**	**type-of**
define-condition	**proclaim**	**typep**

Figure 4–5. Defined names relating to types and declarations.

Figure 4–6 shows all *defined names* that are *type specifier names*, whether for *atomic type specifiers* or *compound type specifiers*; this list is the union of the lists in Figure 4–2 and Figure 4–3.

and	**function**	**simple-array**
arithmetic-error	**generic-function**	**simple-base-string**
array	**hash-table**	**simple-bit-vector**
atom	**integer**	**simple-condition**
base-char	**keyword**	**simple-error**
base-string	**list**	**simple-string**
bignum	**logical-pathname**	**simple-type-error**
bit	**long-float**	**simple-vector**
bit-vector	**member**	**simple-warning**
broadcast-stream	**method**	**single-float**
built-in-class	**method-combination**	**standard-char**
cell-error	**mod**	**standard-class**
character	**nil**	**standard-generic-function**
class	**not**	**standard-method**
compiled-function	**null**	**standard-object**
complex	**number**	**storage-condition**
concatenated-stream	**or**	**stream**
condition	**package**	**stream-error**
cons	**package-error**	**string**
control-error	**parse-error**	**string-stream**
division-by-zero	**pathname**	**structure-class**
double-float	**print-not-readable**	**structure-object**
echo-stream	**program-error**	**style-warning**
end-of-file	**random-state**	**symbol**
eql	**ratio**	**synonym-stream**
error	**rational**	**t**
extended-char	**reader-error**	**two-way-stream**
file-error	**readtable**	**type-error**
file-stream	**real**	**unbound-slot**
fixnum	**restart**	**unbound-variable**
float	**satisfies**	**undefined-function**
floating-point-inexact	**sequence**	**unsigned-byte**
floating-point-invalid-operation	**serious-condition**	**values**
floating-point-overflow	**short-float**	**vector**
floating-point-underflow	**signed-byte**	**warning**

Figure 4–6. Standardized Type Specifier Names

4.3 Classes

While the object system is general enough to describe all *standardized classes* (including, for example, **number**, **hash-table**, and **symbol**), Figure 4–7 contains a list of *classes* that are especially relevant to understanding the object system.

built-in-class	**method-combination**	**standard-object**
class	**standard-class**	**structure-class**
generic-function	**standard-generic-function**	**structure-object**
method	**standard-method**	

Figure 4–7. Object System Classes

4.3.1 Introduction to Classes

A ***class*** is an *object* that determines the structure and behavior of a set of other *objects*, which are called its ***instances***.

A *class* can inherit structure and behavior from other *classes*. A *class* whose definition refers to other *classes* for the purpose of inheriting from them is said to be a *subclass* of each of those *classes*. The *classes* that are designated for purposes of inheritance are said to be *superclasses* of the inheriting *class*.

A *class* can have a *name*. The *function* **class-name** takes a *class object* and returns its *name*. The *name* of an anonymous *class* is **nil**. A *symbol* can *name* a *class*. The *function* **find-class** takes a *symbol* and returns the *class* that the *symbol* names. A *class* has a *proper name* if the *name* is a *symbol* and if the *name* of the *class* names that *class*. That is, a *class* C has the *proper name* S if $S =$ `(class-name` C`)` and $C =$ `(find-class` S`)`. Notice that it is possible for `(find-class` S_1`)` $=$ `(find-class` S_2`)` and $S_1 \neq S_2$. If $C =$ `(find-class` S`)`, we say that C is the *class named* S.

A *class* C_1 is a ***direct superclass*** of a *class* C_2 if C_2 explicitly designates C_1 as a *superclass* in its definition. In this case C_2 is a ***direct subclass*** of C_1. A *class* C_n is a ***superclass*** of a *class* C_1 if there exists a series of *classes* $C_2, \ldots, C_{n-1}$ such that C_{i+1} is a *direct superclass* of C_i for $1 \leq i < n$. In this case, C_1 is a ***subclass*** of C_n. A *class* is considered neither a *superclass* nor a *subclass* of itself. That is, if C_1 is a *superclass* of C_2, then $C_1 \neq C_2$. The set of *classes* consisting of some given *class* C along with all of its *superclasses* is called "C and its superclasses."

Each *class* has a ***class precedence list***, which is a total ordering on the set of the given *class* and its *superclasses*. The total ordering is expressed as a list ordered from most specific to least specific. The *class precedence list* is used in several ways. In general, more specific *classes* can ***shadow***$_1$ features that would otherwise be inherited from less specific *classes*. The *method* selection and combination process uses the *class precedence list* to order *methods* from most specific to least specific.

When a *class* is defined, the order in which its direct *superclasses* are mentioned in the defining form is important. Each *class* has a ***local precedence order***, which is a *list* consisting of the *class* followed by its *direct superclasses* in the order mentioned in the defining *form*.

A *class precedence list* is always consistent with the *local precedence order* of each *class* in the list. The *classes* in each *local precedence order* appear within the *class precedence list* in the same order. If the *local precedence orders* are inconsistent with each other, no *class precedence list* can be constructed, and an error is signaled. The *class precedence list* and its computation is discussed in Section 4.3.5 (Determining the Class Precedence List).

classes are organized into a directed acyclic graph. There are two distinguished *classes*, named **t** and **standard-object**. The *class* named **t** has no *superclasses*. It is a *superclass* of every *class* except itself. The *class* named **standard-object** is an *instance* of the *class* **standard-class** and is a *superclass* of every *class* that is an *instance* of the *class* **standard-class** except itself.

There is a mapping from the object system *class* space into the *type* space. Many of the standard *types* specified in this document have a corresponding *class* that has the same *name* as the *type*. Some *types* do not have a corresponding *class*. The integration of the *type* and *class* systems is discussed in Section 4.3.7 (Integrating Types and Classes).

Classes are represented by *objects* that are themselves *instances* of *classes*. The *class* of the *class* of an *object* is termed the ***metaclass*** of that *object*. When no misinterpretation is possible, the term *metaclass* is used to refer to a *class* that has *instances* that are themselves *classes*. The *metaclass* determines the form of inheritance used by the *classes* that are its *instances* and the representation of the *instances* of those *classes*. The object system provides a default *metaclass*, **standard-class**, that is appropriate for most programs.

Except where otherwise specified, all *classes* mentioned in this standard are *instances* of the *class* **standard-class**, all *generic functions* are *instances* of the *class* **standard-generic-function**, and all *methods* are *instances* of the *class* **standard-method**.

4.3.1.1 Standard Metaclasses

The object system provides a number of predefined *metaclasses*. These include the *classes* **standard-class**, **built-in-class**, and **structure-class**:

- The *class* **standard-class** is the default *class* of *classes* defined by **defclass**.

- The *class* **built-in-class** is the *class* whose *instances* are *classes* that have special implementations with restricted capabilities. Any *class* that corresponds to a standard *type* might be an *instance* of **built-in-class**. The predefined *type* specifiers that are required to have corresponding *classes* are listed in Figure 4–8. It is *implementation-dependent* whether each of these *classes* is implemented as a *built-in class*.

- All *classes* defined by means of **defstruct** are *instances* of the *class* **structure-class**.

4.3.2 Defining Classes

The macro **defclass** is used to define a new named *class*.

The definition of a *class* includes:

- The *name* of the new *class*. For newly-defined *classes* this *name* is a *proper name*.
- The list of the direct *superclasses* of the new *class*.
- A set of ***slot specifiers***. Each *slot specifier* includes the *name* of the *slot* and zero or more *slot* options. A *slot* option pertains only to a single *slot*. If a *class* definition contains two *slot specifiers* with the same *name*, an error is signaled.
- A set of *class* options. Each *class* option pertains to the *class* as a whole.

The *slot* options and *class* options of the **defclass** form provide mechanisms for the following:

- Supplying a default initial value *form* for a given *slot*.
- Requesting that *methods* for *generic functions* be automatically generated for reading or writing *slots*.
- Controlling whether a given *slot* is shared by all *instances* of the *class* or whether each *instance* of the *class* has its own *slot*.
- Supplying a set of initialization arguments and initialization argument defaults to be used in *instance* creation.
- Indicating that the *metaclass* is to be other than the default. The `:metaclass` option is reserved for future use; an implementation can be extended to make use of the `:metaclass` option.
- Indicating the expected *type* for the value stored in the *slot*.
- Indicating the *documentation string* for the *slot*.

4.3.3 Creating Instances of Classes

The generic function **make-instance** creates and returns a new *instance* of a *class*. The object system provides several mechanisms for specifying how a new *instance* is to be initialized. For example, it is possible to specify the initial values for *slots* in newly created *instances* either by giving arguments to **make-instance** or by providing default initial values. Further initialization activities can be performed by *methods* written for *generic functions* that are part of the initialization protocol. The complete initialization protocol is described in Section 7.1 (Object Creation and Initialization).

4.3.4 Inheritance

A *class* can inherit *methods*, *slots*, and some **defclass** options from its *superclasses*. Other sections describe the inheritance of *methods*, the inheritance of *slots* and *slot* options, and the inheritance of *class* options.

4.3.4.1 Examples of Inheritance

```
(defclass C1 ()
    ((S1 :initform 5.4 :type number)
     (S2 :allocation :class)))

(defclass C2 (C1)
    ((S1 :initform 5 :type integer)
     (S2 :allocation :instance)
     (S3 :accessor C2-S3)))
```

Instances of the class `C1` have a *local slot* named `S1`, whose default initial value is 5.4 and whose *value* should always be a *number*. The class `C1` also has a *shared slot* named `S2`.

There is a *local slot* named `S1` in *instances* of `C2`. The default initial value of `S1` is 5. The value of `S1` should always be of type `(and integer number)`. There are also *local slots* named `S2` and `S3` in *instances* of `C2`. The class `C2` has a *method* for `C2-S3` for reading the value of slot `S3`; there is also a *method* for `(setf C2-S3)` that writes the value of `S3`.

4.3.4.2 Inheritance of Class Options

The `:default-initargs` class option is inherited. The set of defaulted initialization arguments for a *class* is the union of the sets of initialization arguments supplied in the `:default-initargs` class options of the *class* and its *superclasses*. When more than one default initial value *form* is supplied for a given initialization argument, the default initial value *form* that is used is the one supplied by the *class* that is most specific according to the *class precedence list*.

If a given `:default-initargs` class option specifies an initialization argument of the same *name* more than once, an error of *type* **program-error** is signaled.

4.3.5 Determining the Class Precedence List

The **defclass** form for a *class* provides a total ordering on that *class* and its direct *superclasses*. This ordering is called the ***local precedence order***. It is an ordered list of the *class* and its direct *superclasses*. The ***class precedence list*** for a class C is a total ordering on C and its *superclasses* that is consistent with the *local precedence orders* for each of C and its *superclasses*.

A *class* precedes its direct *superclasses*, and a direct *superclass* precedes all other direct *superclasses* specified to its right in the *superclasses* list of the **defclass** form. For every class C, define

$$R_C = \{(C, C_1), (C_1, C_2), \ldots, (C_{n-1}, C_n)\}$$

where $C_1, \ldots, C_n$ are the direct *superclasses* of C in the order in which they are mentioned in the **defclass** form. These ordered pairs generate the total ordering on the class C and its direct *superclasses*.

Let S_C be the set of C and its *superclasses*. Let R be

$$R = \bigcup_{c \in S_C} R_c$$

.

The set R might or might not generate a partial ordering, depending on whether the R_c, $c \in S_C$, are consistent; it is assumed that they are consistent and that R generates a partial ordering. When the R_c are not consistent, it is said that R is inconsistent.

To compute the *class precedence list* for C, topologically sort the elements of S_C with respect to the partial ordering generated by R. When the topological sort must select a *class* from a set of two or more *classes*, none of which are preceded by other *classes* with respect to R, the *class* selected is chosen deterministically, as described below.

If R is inconsistent, an error is signaled.

4.3.5.1 Topological Sorting

Topological sorting proceeds by finding a class C in S_C such that no other *class* precedes that element according to the elements in R. The class C is placed first in the result. Remove C from S_C, and remove all pairs of the form (C, D), $D \in S_C$, from R. Repeat the process, adding *classes* with no predecessors to the end of the result. Stop when no element can be found that has no predecessor.

If S_C is not empty and the process has stopped, the set R is inconsistent. If every *class* in the finite set of *classes* is preceded by another, then R contains a loop. That is, there is a chain of classes $C_1, \ldots, C_n$ such that C_i precedes C_{i+1}, $1 \leq i < n$, and C_n precedes C_1.

Sometimes there are several *classes* from S_C with no predecessors. In this case select the one that has a direct *subclass* rightmost in the *class precedence list* computed so far. (If there is no such candidate *class*, R does not generate a partial ordering—the R_c, $c \in S_C$, are inconsistent.)

In more precise terms, let $\{N_1, \ldots, N_m\}$, $m \geq 2$, be the *classes* from S_C with no predecessors. Let $(C_1 \ldots C_n)$, $n \geq 1$, be the *class precedence list* constructed so far. C_1 is the most specific *class*, and C_n is the least specific. Let $1 \leq j \leq n$ be the largest number such that there exists an i where $1 \leq i \leq m$ and N_i is a direct *superclass* of C_j; N_i is placed next.

The effect of this rule for selecting from a set of *classes* with no predecessors is that the *classes* in a simple *superclass* chain are adjacent in the *class precedence list* and that *classes* in each relatively separated subgraph are adjacent in the *class precedence list*. For example, let T_1 and T_2 be subgraphs whose only element in common is the class J. Suppose that no superclass of J appears in either T_1 or T_2, and that J is in the superclass chain of every class in both T_1 and T_2. Let C_1 be the bottom of T_1; and let C_2 be the bottom of T_2. Suppose C is a *class* whose direct *superclasses* are C_1 and C_2 in that order, then the *class precedence list* for C starts with C and is followed by all *classes* in T_1 except J. All the *classes* of T_2 are next. The *class* J and its *superclasses* appear last.

4.3.5.2 Examples of Class Precedence List Determination

This example determines a *class precedence list* for the class `pie`. The following *classes* are defined:

```
(defclass pie (apple cinnamon) ())

(defclass apple (fruit) ())

(defclass cinnamon (spice) ())

(defclass fruit (food) ())

(defclass spice (food) ())

(defclass food () ())
```

The set S_{pie} = {`pie, apple, cinnamon, fruit, spice, food, standard-object, t`}. The set R = {`(pie, apple), (apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `pie` is not preceded by anything, so it comes first; the result so far is `(pie)`. Remove `pie` from S and pairs mentioning `pie` from R to get S = {`apple, cinnamon, fruit, spice, food, standard-object, t`} and R = {`(apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `apple` is not preceded by anything, so it is next; the result is `(pie apple)`. Removing `apple` and the relevant pairs results in S = {`cinnamon, fruit, spice, food, standard-object, t`} and R = {`(cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The classes `cinnamon` and `fruit` are not preceded by anything, so the one with a direct *subclass* rightmost in the *class precedence list* computed so far goes next. The class `apple` is a direct

subclass of `fruit`, and the class `pie` is a direct *subclass* of `cinnamon`. Because `apple` appears to the right of `pie` in the *class precedence list*, `fruit` goes next, and the result so far is `(pie apple fruit)`. $S = \{$`cinnamon, spice, food, standard-object, t`$\}$; $R = \{$`(cinnamon, spice), (spice, food),`
`(food, standard-object), (standard-object, t)`$\}$.

The class `cinnamon` is next, giving the result so far as `(pie apple fruit cinnamon)`. At this point $S = \{$`spice, food, standard-object, t`$\}$; $R = \{$`(spice, food), (food, standard-object), (standard-object, t)`$\}$.

The classes `spice`, `food`, **standard-object**, and `t` are added in that order, and the *class precedence list* is `(pie apple fruit cinnamon spice food standard-object t)`.

It is possible to write a set of *class* definitions that cannot be ordered. For example:

```
(defclass new-class (fruit apple) ())

(defclass apple (fruit) ())
```

The class `fruit` must precede `apple` because the local ordering of *superclasses* must be preserved. The class `apple` must precede `fruit` because a *class* always precedes its own *superclasses*. When this situation occurs, an error is signaled, as happens here when the system tries to compute the *class precedence list* of `new-class`.

The following might appear to be a conflicting set of definitions:

```
(defclass pie (apple cinnamon) ())

(defclass pastry (cinnamon apple) ())

(defclass apple () ())

(defclass cinnamon () ())
```

The *class precedence list* for `pie` is `(pie apple cinnamon standard-object t)`.

The *class precedence list* for `pastry` is `(pastry cinnamon apple standard-object t)`.

It is not a problem for `apple` to precede `cinnamon` in the ordering of the *superclasses* of `pie` but not in the ordering for `pastry`. However, it is not possible to build a new *class* that has both `pie` and `pastry` as *superclasses*.

4.3.6 Redefining Classes

A *class* that is a *direct instance* of **standard-class** can be redefined if the new *class* is also a *direct instance* of **standard-class**. Redefining a *class* modifies the existing *class object* to reflect the new *class* definition; it does not create a new *class object* for the *class*. Any *method object* created by a `:reader`, `:writer`, or `:accessor` option specified by the old **defclass** form is removed from the corresponding *generic function*. *Methods* specified by the new **defclass** form are added.

When the class C is redefined, changes are propagated to its *instances* and to *instances* of any of its *subclasses*. Updating such an *instance* occurs at an *implementation-dependent* time, but no later than the next time a *slot* of that *instance* is read or written. Updating an *instance* does not change its identity as defined by the *function* **eq**. The updating process may change the *slots* of that particular *instance*, but it does not create a new *instance*. Whether updating an *instance* consumes storage is *implementation-dependent*.

Note that redefining a *class* may cause *slots* to be added or deleted. If a *class* is redefined in a way that changes the set of *local slots accessible* in *instances*, the *instances* are updated. It is *implementation-dependent* whether *instances* are updated if a *class* is redefined in a way that does not change the set of *local slots accessible* in *instances*.

The value of a *slot* that is specified as shared both in the old *class* and in the new *class* is retained. If such a *shared slot* was unbound in the old *class*, it is unbound in the new *class*. *Slots* that were local in the old *class* and that are shared in the new *class* are initialized. Newly added *shared slots* are initialized.

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*. If there was no *initialization form*, the *slot* is unbound.

If a *class* is redefined in such a way that the set of *local slots accessible* in an *instance* of the *class* is changed, a two-step process of updating the *instances* of the *class* takes place. The process may be explicitly started by invoking the generic function **make-instances-obsolete**. This two-step process can happen in other circumstances in some implementations. For example, in some implementations this two-step process is triggered if the order of *slots* in storage is changed.

The first step modifies the structure of the *instance* by adding new *local slots* and discarding *local slots* that are not defined in the new version of the *class*. The second step initializes the newly-added *local slots* and performs any other user-defined actions. These two steps are further specified in the next two sections.

4.3.6.1 Modifying the Structure of Instances

The first step modifies the structure of *instances* of the redefined *class* to conform to its new *class* definition. *Local slots* specified by the new *class* definition that are not specified as either local or shared by the old *class* are added, and *slots* not specified as either local or shared by the new *class* definition that are specified as local by the old *class* are discarded. The *names* of these added and discarded *slots* are passed as arguments to **update-instance-for-redefined-class** as described in the next section.

The values of *local slots* specified by both the new and old *classes* are retained. If such a *local slot* was unbound, it remains unbound.

The value of a *slot* that is specified as shared in the old *class* and as local in the new *class* is retained. If such a *shared slot* was unbound, the *local slot* is unbound.

4.3.6.2 Initializing Newly Added Local Slots

The second step initializes the newly added *local slots* and performs any other user-defined actions. This step is implemented by the generic function **update-instance-for-redefined-class**, which is called after completion of the first step of modifying the structure of the *instance*.

The generic function **update-instance-for-redefined-class** takes four required arguments: the *instance* being updated after it has undergone the first step, a list of the names of *local slots* that were added, a list of the names of *local slots* that were discarded, and a property list containing the *slot* names and values of *slots* that were discarded and had values. Included among the discarded *slots* are *slots* that were local in the old *class* and that are shared in the new *class*.

The generic function **update-instance-for-redefined-class** also takes any number of initialization arguments. When it is called by the system to update an *instance* whose *class* has been redefined, no initialization arguments are provided.

There is a system-supplied primary *method* for **update-instance-for-redefined-class** whose *parameter specializer* for its *instance* argument is the *class* **standard-object**. First this *method* checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (For more information, see Section 7.1.2 (Declaring the Validity of Initialization Arguments).) Then it calls the generic function **shared-initialize** with the following arguments: the *instance*, the list of *names* of the newly added *slots*, and the initialization arguments it received.

4.3.6.3 Customizing Class Redefinition

Methods for **update-instance-for-redefined-class** may be defined to specify actions to be taken when an *instance* is updated. If only *after methods* for **update-instance-for-redefined-class** are defined, they will be run after the system-supplied primary *method* for initialization and therefore will not interfere with the default behavior of **update-instance-for-redefined-class**. Because no initialization arguments are passed to **update-instance-for-redefined-class** when it is called by the system, the *initialization forms* for *slots* that are filled by *before methods* for **update-instance-for-redefined-class** will not be evaluated by **shared-initialize**.

Methods for **shared-initialize** may be defined to customize *class* redefinition. For more information, see Section 7.1.5 (Shared-Initialize).

4.3.7 Integrating Types and Classes

The object system maps the space of *classes* into the space of *types*. Every *class* that has a proper name has a corresponding *type* with the same *name*.

The proper name of every *class* is a valid *type specifier*. In addition, every *class object* is a valid *type specifier*. Thus the expression `(typep object class)` evaluates to *true* if the *class* of *object* is *class* itself or a *subclass* of *class*. The evaluation of the expression `(subtypep class1 class2)` returns the values *true* and *true* if `class1` is a subclass of `class2` or if they are the same *class*; otherwise it returns the values *false* and *true*. If I is an *instance* of some *class* C named S and

C is an *instance* of **standard-class**, the evaluation of the expression `(type-of` I`)` returns S if S is the *proper name* of C; otherwise, it returns C.

Because the names of *classes* and *class objects* are *type specifiers*, they may be used in the special form **the** and in type declarations.

Many but not all of the predefined *type specifiers* have a corresponding *class* with the same proper name as the *type*. These type specifiers are listed in Figure 4–8. For example, the *type* **array** has a corresponding *class* named **array**. No *type specifier* that is a list, such as `(vector double-float 100)`, has a corresponding *class*. The *operator* **deftype** does not create any *classes*.

Each *class* that corresponds to a predefined *type specifier* can be implemented in one of three ways, at the discretion of each implementation. It can be a *standard class*, a *structure class*, or a *system class*.

A *built-in class* is one whose *generalized instances* have restricted capabilities or special representations. Attempting to use **defclass** to define *subclasses* of a **built-in-class** signals an error. Calling **make-instance** to create a *generalized instance* of a *built-in class* signals an error. Calling **slot-value** on a *generalized instance* of a *built-in class* signals an error. Redefining a *built-in class* or using **change-class** to change the *class* of an *object* to or from a *built-in class* signals an error. However, *built-in classes* can be used as *parameter specializers* in *methods*.

It is possible to determine whether a *class* is a *built-in class* by checking the *metaclass*. A *standard class* is an *instance* of the *class* **standard-class**, a *built-in class* is an *instance* of the *class* **built-in-class**, and a *structure class* is an *instance* of the *class* **structure-class**.

Each *structure type* created by **defstruct** without using the `:type` option has a corresponding *class*. This *class* is a *generalized instance* of the *class* **structure-class**. The `:include` option of **defstruct** creates a direct *subclass* of the *class* that corresponds to the included *structure type*.

It is *implementation-dependent* whether *slots* are involved in the operation of *functions* defined in this specification on *instances* of *classes* defined in this specification, except when *slots* are explicitly defined by this specification.

If in a particular *implementation* a *class* defined in this specification has *slots* that are not defined by this specfication, the names of these *slots* must not be *external symbols* of *packages* defined in this specification nor otherwise *accessible* in the `CL-USER` *package*.

The purpose of specifying that many of the standard *type specifiers* have a corresponding *class* is to enable users to write *methods* that discriminate on these *types*. *Method* selection requires that a *class precedence list* can be determined for each *class*.

The hierarchical relationships among the *type specifiers* are mirrored by relationships among the *classes* corresponding to those *types*.

Figure 4–8 lists the set of *classes* that correspond to predefined *type specifiers*.

arithmetic-error	**generic-function**	**simple-error**
array	**hash-table**	**simple-type-error**
bit-vector	**integer**	**simple-warning**
broadcast-stream	**list**	**standard-class**
built-in-class	**logical-pathname**	**standard-generic-function**
cell-error	**method**	**standard-method**
character	**method-combination**	**standard-object**
class	**null**	**storage-condition**
complex	**number**	**stream**
concatenated-stream	**package**	**stream-error**
condition	**package-error**	**string**
cons	**parse-error**	**string-stream**
control-error	**pathname**	**structure-class**
division-by-zero	**print-not-readable**	**structure-object**
echo-stream	**program-error**	**style-warning**
end-of-file	**random-state**	**symbol**
error	**ratio**	**synonym-stream**
file-error	**rational**	**t**
file-stream	**reader-error**	**two-way-stream**
float	**readtable**	**type-error**
floating-point-inexact	**real**	**unbound-slot**
floating-point-invalid-operation	**restart**	**unbound-variable**
floating-point-overflow	**sequence**	**undefined-function**
floating-point-underflow	**serious-condition**	**vector**
function	**simple-condition**	**warning**

Figure 4–8. Classes that correspond to pre-defined type specifiers

The *class precedence list* information specified in the entries for each of these *classes* are those that are required by the object system.

Individual implementations may be extended to define other type specifiers to have a corresponding *class*. Individual implementations may be extended to add other *subclass* relationships and to add other *elements* to the *class precedence lists* as long as they do not violate the type relationships and disjointness requirements specified by this standard. A standard *class* defined with no direct *superclasses* is guaranteed to be disjoint from all of the *classes* in the table, except for the class named **t**.

nil *Type*

Supertypes:

all *types*

Description:

The *type* **nil** contains no *objects* and so is also called the *empty type*. The *type* **nil** is a *subtype* of every *type*. No *object* is of *type* **nil**.

Notes:

The *type* containing the *object* **nil** is the *type* **null**, not the *type* **nil**.

boolean *Type*

Supertypes:

boolean, **symbol**, **t**

Description:

The *type* **boolean** contains the *symbols* **t** and **nil**, which represent true and false, respectively.

See Also:

t (*constant variable*), **nil** (*constant variable*), **if**, **not**, **complement**

Notes:

Conditional operations, such as **if**, permit the use of *generalized booleans*, not just *booleans*; any *non-nil* value, not just **t**, counts as true for a *generalized boolean*. However, as a matter of convention, the *symbol* **t** is considered the canonical value to use even for a *generalized boolean* when no better choice presents itself.

function

function *System Class*

Class Precedence List:

function, **t**

Description:

A *function* is an *object* that represents code to be executed when an appropriate number of arguments is supplied. A *function* is produced by the **function** *special form*, the *function* **coerce**, or the *function* **compile**. A *function* can be directly invoked by using it as the first argument to **funcall**, **apply**, or **multiple-value-call**.

Compound Type Specifier Kind:

Specializing.

Compound Type Specifier Syntax:

(function [*arg-typespec* [*value-typespec*]])

arg-typespec::=({*typespec*}*
[&optional {*typespec*}*]
[&rest *typespec*]
[&key {(*keyword typespec*)}*])

Compound Type Specifier Arguments:

typespec—a *type specifier*.

value-typespec—a *type specifier*.

Compound Type Specifier Description:

The list form of the **function** *type-specifier* can be used only for declaration and not for discrimination. Every element of this *type* is a *function* that accepts arguments of the types specified by the *argj-types* and returns values that are members of the *types* specified by *value-type*. The **&optional**, **&rest**, **&key**, and **&allow-other-keys** markers can appear in the list of argument types. The *type specifier* provided with **&rest** is the *type* of each actual argument, not the *type* of the corresponding variable.

The **&key** parameters should be supplied as lists of the form (*keyword type*). The *keyword* must be a valid keyword-name symbol as must be supplied in the actual arguments of a call. This is usually a *symbol* in the KEYWORD *package* but can be any *symbol*. When **&key** is given in a **function** *type specifier lambda list*, the *keyword parameters* given are exhaustive unless **&allow-other-keys** is also present. **&allow-other-keys** is an indication that other keyword arguments might actually be supplied and, if supplied, can be used. For example, the *type* of the *function* **make-list** could be declared as follows:

```
(function ((integer 0) &key (:initial-element t)) list)
```

The *value-type* can be a **values** *type specifier* in order to indicate the *types* of *multiple values*.

Consider a declaration of the following form:

```
(ftype (function (arg0-type arg1-type ...) val-type) f))
```

Any *form* `(f arg0 arg1 ...)` within the scope of that declaration is equivalent to the following:

```
(the val-type (f (the arg0-type arg0) (the arg1-type arg1) ...))
```

That is, the consequences are undefined if any of the arguments are not of the specified *types* or the result is not of the specified *type*. In particular, if any argument is not of the correct *type*, the result is not guaranteed to be of the specified *type*.

Thus, an **ftype** declaration for a *function* describes *calls* to the *function*, not the actual definition of the *function*.

Consider a declaration of the following form:

```
(type (function (arg0-type arg1-type ...) val-type) fn-valued-variable)
```

This declaration has the interpretation that, within the scope of the declaration, the consequences are unspecified if the value of `fn-valued-variable` is called with arguments not of the specified *types*; the value resulting from a valid call will be of type `val-type`.

As with variable type declarations, nested declarations imply intersections of *types*, as follows:

- Consider the following two declarations of **ftype**:

  ```
  (ftype (function (arg0-type1 arg1-type1 ...) val-type1) f))
  ```

  and

  ```
  (ftype (function (arg0-type2 arg1-type2 ...) val-type2) f))
  ```

  If both these declarations are in effect, then within the shared scope of the declarations, calls to `f` can be treated as if `f` were declared as follows:

  ```
  (ftype (function ((and arg0-type1 arg0-type2) (and arg1-type1 arg1-type2 ...) ...)
                   (and val-type1 val-type2))
         f))
  ```

  It is permitted to ignore one or all of the **ftype** declarations in force.

- If two (or more) type declarations are in effect for a variable, and they are both `function` declarations, the declarations combine similarly.

compiled-function *Type*

Supertypes:

compiled-function, **function**, **t**

Description:

Any *function* may be considered by an *implementation* to be a a *compiled function* if it contains no references to *macros* that must be expanded at run time, and it contains no unresolved references to *load time values*. See Section 3.2.2 (Compilation Semantics).

Functions whose definitions appear lexically within a *file* that has been *compiled* with **compile-file** and then *loaded* with **load** are of *type* **compiled-function**. *Functions* produced by the **compile** function are of *type* **compiled-function**. Other *functions* might also be of *type* **compiled-function**.

generic-function *System Class*

Class Precedence List:

generic-function, **function**, **t**

Description:

A ***generic function*** is a *function* whose behavior depends on the *classes* or identities of the *arguments* supplied to it. A generic function object contains a set of *methods*, a *lambda list*, a *method combination type*, and other information. The *methods* define the class-specific behavior and operations of the *generic function*; a *method* is said to *specialize* a *generic function*. When invoked, a *generic function* executes a subset of its *methods* based on the *classes* or identities of its *arguments*.

A *generic function* can be used in the same ways that an ordinary *function* can be used; specifically, a *generic function* can be used as an argument to **funcall** and **apply**, and can be given a global or a local name.

standard-generic-function *System Class*

Class Precedence List:

standard-generic-function, **generic-function**, **function**, **t**

Description:

The *class* **standard-generic-function** is the default *class* of *generic functions established* by **defmethod**, **ensure-generic-function**, **defgeneric**, and **defclass** *forms*.

class *System Class*

Class Precedence List:

class, **standard-object**, **t**

Description:

The *type* **class** represents *objects* that determine the structure and behavior of their *instances*. Associated with an *object* of *type* **class** is information describing its place in the directed acyclic graph of *classes*, its *slots*, and its options.

built-in-class *System Class*

Class Precedence List:

built-in-class, **class**, **standard-object**, **t**

Description:

A *built-in class* is a *class* whose *instances* have restricted capabilities or special representations. Attempting to use **defclass** to define *subclasses* of a *built-in class* signals an error of *type* **error**. Calling **make-instance** to create an *instance* of a *built-in class* signals an error of *type* **error**. Calling **slot-value** on an *instance* of a *built-in class* signals an error of *type* **error**. Redefining a *built-in class* or using **change-class** to change the *class* of an *instance* to or from a *built-in class* signals an error of *type* **error**. However, *built-in classes* can be used as *parameter specializers* in *methods*.

structure-class *System Class*

Class Precedence List:

structure-class, **class**, **standard-object**, **t**

Description:

All *classes* defined by means of **defstruct** are *instances* of the *class* **structure-class**.

standard-class *System Class*

Class Precedence List:

standard-class, **class**, **standard-object**, **t**

Description:

The *class* **standard-class** is the default *class* of *classes* defined by **defclass**.

method *System Class*

Class Precedence List:

method, **t**

Description:

A *method* is an *object* that represents a modular part of the behavior of a *generic function*.

A *method* contains *code* to implement the *method*'s behavior, a sequence of *parameter specializers* that specify when the given *method* is applicable, and a sequence of *qualifiers* that is used by the method combination facility to distinguish among *methods*. Each required parameter of each *method* has an associated *parameter specializer*, and the *method* will be invoked only on arguments that satisfy its *parameter specializers*.

The method combination facility controls the selection of *methods*, the order in which they are run, and the values that are returned by the generic function. The object system offers a default method combination type and provides a facility for declaring new types of method combination.

See Also:

Section 7.6 (Generic Functions and Methods)

standard-method *System Class*

Class Precedence List:

standard-method, **method**, **standard-object**, **t**

Description:

The *class* **standard-method** is the default *class* of *methods* defined by the **defmethod** and **defgeneric** *forms*.

structure-object *Class*

Class Precedence List:

structure-object, **t**

Description:

The *class* **structure-object** is an *instance* of **structure-class** and is a *superclass* of every *class* that is an *instance* of **structure-class** except itself, and is a *superclass* of every *class* that is defined by **defstruct**.

See Also:

defstruct, Section 2.4.8.13 (Sharpsign S), Section 22.1.3.12 (Printing Structures)

standard-object *Class*

Class Precedence List:

standard-object, **t**

Description:

The *class* **standard-object** is an *instance* of **standard-class** and is a *superclass* of every *class* that is an *instance* of **standard-class** except itself.

method-combination *System Class*

Class Precedence List:

method-combination, **t**

Description:

Every *method combination object* is an *indirect instance* of the *class* **method-combination**. A *method combination object* represents the information about the *method combination* being used by a *generic function*. A *method combination object* contains information about both the type of *method combination* and the arguments being used with that *type*.

t *System Class*

Class Precedence List:

t

Description:

The set of all *objects*. The *type* **t** is a *supertype* of every *type*, including itself. Every *object* is of *type* **t**.

satisfies *Type Specifier*

Compound Type Specifier Kind:

Predicating.

Compound Type Specifier Syntax:

(satisfies *predicate-name*)

Compound Type Specifier Arguments:

predicate-name—a *symbol*.

Compound Type Specifier Description:

This denotes the set of all *objects* that satisfy the *predicate* *predicate-name*, which must be a *symbol* whose global *function* definition is a one-argument predicate. A name is required for *predicate-name*; *lambda expressions* are not allowed. For example, the *type specifier* `(and integer (satisfies evenp))` denotes the set of all even integers. The form `(typep x '(satisfies p))` is equivalent to `(if (p x) t nil)`.

The argument is required. The *symbol* * can be the argument, but it denotes itself (the *symbol* *), and does not represent an unspecified value.

The symbol **satisfies** is not valid as a *type specifier*.

member *Type Specifier*

Compound Type Specifier Kind:

Combining.

Compound Type Specifier Syntax:

(member {*object*}*)

Compound Type Specifier Arguments:

object—an *object*.

Compound Type Specifier Description:

This denotes the set containing the named *objects*. An *object* is of this *type* if and only if it is **eql** to one of the specified *objects*.

The *type specifiers* (member) and **nil** are equivalent. * can be among the *objects*, but if so it denotes itself (the symbol *) and does not represent an unspecified value. The symbol **member** is not valid as a *type specifier*; and, specifically, it is not an abbreviation for either (member) or (member *).

See Also:

the *type* **eql**

not *Type Specifier*

Compound Type Specifier Kind:

Combining.

Compound Type Specifier Syntax:

(not *typespec*)

Compound Type Specifier Arguments:

typespec—a *type specifier*.

Compound Type Specifier Description:

This denotes the set of all *objects* that are not of the *type* *typespec*.

The argument is required, and cannot be *****.

The symbol **not** is not valid as a *type specifier*.

and *Type Specifier*

Compound Type Specifier Kind:

Combining.

Compound Type Specifier Syntax:

`(and` {*typespec*}*`)`

Compound Type Specifier Arguments:

typespec—a *type specifier*.

Compound Type Specifier Description:

This denotes the set of all *objects* of the *type* determined by the intersection of the *typespecs*.

***** is not permitted as an argument.

The *type specifiers* `(and)` and **t** are equivalent. The symbol **and** is not valid as a *type specifier*, and, specifically, it is not an abbreviation for `(and)`.

or *Type Specifier*

Compound Type Specifier Kind:

Combining.

Compound Type Specifier Syntax:

`(or` {*typespec*}*`)`

Compound Type Specifier Arguments:

typespec—a *type specifier*.

Compound Type Specifier Description:

This denotes the set of all *objects* of the *type* determined by the union of the *typespecs*. For example, the *type* **list** by definition is the same as `(or null cons)`. Also, the value returned by **position** is an *object* of *type* `(or null (integer 0 *))`; *i.e.*, either **nil** or a non-negative *integer*.

***** is not permitted as an argument.

The *type specifiers* `(or)` and **nil** are equivalent. The symbol **or** is not valid as a *type specifier*; and, specifically, it is not an abbreviation for `(or)`.

values *Type Specifier*

Compound Type Specifier Kind:

Specializing.

Compound Type Specifier Syntax:

`(values` ↓*value-typespec*`)`

value-typespec::={*typespec*}* `[&optional` {*typespec*}*`]` `[&rest` *typespec*`]` `[`**&allow-other-keys**`]`

Compound Type Specifier Arguments:

typespec—a *type specifier*.

Compound Type Specifier Description:

This *type specifier* can be used only as the *value-type* in a **function** *type specifier* or a **the** *special form*. It is used to specify individual *types* when *multiple values* are involved. The **&optional** and **&rest** markers can appear in the *value-type* list; they indicate the parameter list of a *function* that, when given to **multiple-value-call** along with the values, would correctly receive those values.

The symbol ***** may not be among the *value-types*.

The symbol **values** is not valid as a *type specifier*; and, specifically, it is not an abbreviation for `(values)`.

eql *Type Specifier*

Compound Type Specifier Kind:

Combining.

Compound Type Specifier Syntax:

(eql *object*)

Compound Type Specifier Arguments:

object—an *object*.

Compound Type Specifier Description:

Represents the *type* of all *x* for which (eql *object x*) is true.

The argument *object* is required. The *object* can be *, but if so it denotes itself (the symbol *) and does not represent an unspecified value. The *symbol* **eql** is not valid as an *atomic type specifier*.

coerce *Function*

Syntax:

coerce *object result-type* → *result*

Arguments and Values:

object—an *object*.

result-type—a *type specifier*.

result—an *object*, of *type result-type* except in situations described in Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

Description:

Coerces the *object* to *type result-type*.

If *object* is already of *type result-type*, the *object* itself is returned, regardless of whether it would have been possible in general to coerce an *object* of some other *type* to *result-type*.

Otherwise, the *object* is *coerced* to *type result-type* according to the following rules:

sequence

If the *result-type* is a *recognizable subtype* of **list**, and the *object* is a *sequence*, then the *result* is a *list* that has the *same elements* as *object*.

If the *result-type* is a *recognizable subtype* of **vector**, and the *object* is a *sequence*, then the *result* is a *vector* that has the *same elements* as *object*. If *result-type* is a specialized *type*, the *result* has an *actual array element type* that is the result of *upgrading* the element type part of that *specialized type*. If no element type is specified, the element type defaults to `t`. If the *implementation* cannot determine the element type, an error is signaled.

character

If the *result-type* is **character** and the *object* is a *character designator*, the *result* is the *character* it denotes.

complex

If the *result-type* is **complex** and the *object* is a *real*, then the *result* is obtained by constructing a *complex* whose real part is the *object* and whose imaginary part is the result of *coercing* an *integer* zero to the *type* of the *object* (using **coerce**). (If the real part is a *rational*, however, then the result must be represented as a *rational* rather than a *complex*; see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals). So, for example, `(coerce 3 'complex)` is permissible, but will return `3`, which is not a *complex*.)

float

If the *result-type* is any of **float**, **short-float**, **single-float**, **double-float**, **long-float**, and the *object* is a *real*, then the *result* is a *float* of *type* *result-type* which is equal in sign and magnitude to the *object* to whatever degree of representational precision is permitted by that *float* representation. (If the *result-type* is **float** and *object* is not already a *float*, then the *result* is a *single float*.)

function

If the *result-type* is **function**, and *object* is any *function name* that is *fbound* but that is globally defined neither as a *macro name* nor as a *special operator*, then the *result* is the *functional value* of *object*.

If the *result-type* is **function**, and *object* is a *lambda expression*, then the *result* is a *closure* of *object* in the *null lexical environment*.

t

Any *object* can be *coerced* to an *object* of *type* `t`. In this case, the *object* is simply returned.

Examples:

```
(coerce '(a b c) 'vector) → #(A B C)
(coerce 'a 'character) → #\A
(coerce 4.56 'complex) → #C(4.56 0.0)
(coerce 4.5s0 'complex) → #C(4.5s0 0.0s0)
(coerce 7/2 'complex) → 7/2
(coerce 0 'short-float) → 0.0s0
(coerce 3.5L0 'float) → 3.5L0
(coerce 7/2 'float) → 3.5
(coerce (cons 1 2) t) → (1 . 2)
```

All the following *forms* should signal an error:

```
(coerce '(a b c) '(vector * 4))
(coerce #(a b c) '(vector * 4))
(coerce '(a b c) '(vector * 2))
(coerce #(a b c) '(vector * 2))
(coerce "foo" '(string 2))
(coerce #(#\a #\b #\c) '(string 2))
(coerce '(0 1) '(simple-bit-vector 3))
```

Exceptional Situations:

If a coercion is not possible, an error of *type* **type-error** is signaled.

`(coerce x 'nil)` always signals an error of *type* **type-error**.

An error of *type* **error** is signaled if the *result-type* is **function** but *object* is a *symbol* that is not *fbound* or if the *symbol* names a *macro* or a *special operator*.

An error of *type* **type-error** should be signaled if *result-type* specifies the number of elements and *object* is of a different length.

See Also:

rational, **floor**, **char-code**, **char-int**

Notes:

Coercions from *floats* to *rationals* and from *ratios* to *integers* are not provided because of rounding problems.

```
(coerce x 't) ≡ (identity x) ≡ x
```

deftype *Macro*

Syntax:

deftype *name lambda-list* ⟦ {*declaration*}* | *documentation* ⟧ {*form*}* → *name*

Arguments and Values:

name—a *symbol*.

lambda-list—a *deftype lambda list*.

declaration—a **declare** *expression*; not evaluated.

documentation—a *string*; not evaluated.

form—a *form*.

Description:

deftype defines a *derived type specifier* named *name*.

The meaning of the new *type specifier* is given in terms of a function which expands the *type specifier* into another *type specifier*, which itself will be expanded if it contains references to another *derived type specifier*.

The newly defined *type specifier* may be referenced as a list of the form (*name* arg_1 arg_2 ...). The number of arguments must be appropriate to the *lambda-list*. If the new *type specifier* takes no arguments, or if all of its arguments are optional, the *type specifier* may be used as an *atomic type specifier*.

The *argument expressions* to the *type specifier*, arg_1 ... arg_n, are not *evaluated*. Instead, these *literal objects* become the *objects* to which corresponding *parameters* become *bound*.

The body of the **deftype** *form* (but not the *lambda-list*) is implicitly enclosed in a *block* named *name*, and is evaluated as an *implicit progn*, returning a new *type specifier*.

The *lexical environment* of the body is the one which was current at the time the **deftype** form was evaluated, augmented by the *variables* in the *lambda-list*.

Recursive expansion of the *type specifier* returned as the expansion must terminate, including the expansion of *type specifiers* which are nested within the expansion.

The consequences are undefined if the result of fully expanding a *type specifier* contains any circular structure, except within the *objects* referred to by **member** and **eql** *type specifiers*.

Documentation is attached to *name* as a *documentation string* of kind **type**.

If a **deftype** *form* appears as a *top level form*, the *compiler* must ensure that the *name* is recognized in subsequent *type* declarations. The *programmer* must ensure that the body of a **deftype** form can be *evaluated* at compile time if the *name* is referenced in subsequent *type* declarations.

If the expansion of a *type specifier* is not defined fully at compile time (perhaps because it expands into an unknown *type specifier* or a **satisfies** of a named *function* that isn't defined in the compile-time environment), an *implementation* may ignore any references to this *type* in declarations and/or signal a warning.

Examples:

```
(defun equidimensional (a)
  (or (< (array-rank a) 2)
      (apply #'= (array-dimensions a)))) → EQUIDIMENSIONAL
(deftype square-matrix (&optional type size)
  `(and (array ,type (,size ,size))
        (satisfies equidimensional))) → SQUARE-MATRIX
```

See Also:

declare, **defmacro**, **documentation**, Section 4.2.3 (Type Specifiers), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

subtypep *Function*

Syntax:

subtypep *type-1 type-2* `&optional` *environment* → *subtype-p, valid-p*

Arguments and Values:

type-1—a *type specifier*.

type-2—a *type specifier*.

environment—an *environment object*. The default is **nil**, denoting the *null lexical environment* and the current *global environment*.

subtype-p—a *generalized boolean*.

valid-p—a *generalized boolean*.

Description:

If *type-1* is a *recognizable subtype* of *type-2*, the first *value* is *true*. Otherwise, the first *value* is *false*, indicating that either *type-1* is not a *subtype* of *type-2*, or else *type-1* is a *subtype* of *type-2* but is not a *recognizable subtype*.

A second *value* is also returned indicating the 'certainty' of the first *value*. If this value is *true*, then the first value is an accurate indication of the *subtype* relationship. (The second *value* is always *true* when the first *value* is *true*.)

Figure 4–9 summarizes the possible combinations of *values* that might result.

Value 1	Value 2	Meaning
true	*true*	*type-1* is definitely a *subtype* of *type-2*.
false	*true*	*type-1* is definitely not a *subtype* of *type-2*.
false	*false*	**subtypep** could not determine the relationship, so *type-1* might or might not be a *subtype* of *type-2*.

Figure 4–9. Result possibilities for subtypep

subtypep is permitted to return the *values* *false* and *false* only when at least one argument involves one of these *type specifiers*: **and**, **eql**, the list form of **function**, **member**, **not**, **or**, **satisfies**, or **values**. (A *type specifier* 'involves' such a *symbol* if, after being *type expanded*, it contains that *symbol* in a position that would call for its meaning as a *type specifier* to be used.) One consequence of this is that if neither *type-1* nor *type-2* involves any of these *type specifiers*, then **subtypep** is obliged to determine the relationship accurately. In particular, **subtypep** returns the *values* *true* and *true* if the arguments are **equal** and do not involve any of these *type specifiers*.

subtypep never returns a second value of **nil** when both *type-1* and *type-2* involve only the names in Figure 4–2, or names of *types* defined by **defstruct**, **define-condition**, or **defclass**, or *derived types* that expand into only those names. While *type specifiers* listed in Figure 4–2 and names of **defclass** and **defstruct** can in some cases be implemented as *derived types*, **subtypep** regards them as primitive.

The relationships between *types* reflected by **subtypep** are those specific to the particular implementation. For example, if an implementation supports only a single type of floating-point numbers, in that implementation `(subtypep 'float 'long-float)` returns the *values* *true* and *true* (since the two *types* are identical).

For all *T1* and *T2* other than `*`, `(array` *T1*`)` and `(array` *T2*`)` are two different *type specifiers* that always refer to the same sets of things if and only if they refer to *arrays* of exactly the same specialized representation, *i.e.*, if `(upgraded-array-element-type '`*T1*`)` and `(upgraded-array-element-type '`*T2*`)` return two different *type specifiers* that always refer to the same sets of *objects*. This is another way of saying that `` `(array `` *type-specifier*`)` and `` `(array ,(upgraded-array-element-type ' ``*type-specifier*`))` refer to the same set of specialized *array* representations. For all *T1* and *T2* other than `*`, the intersection of `(array` *T1*`)` and `(array` *T2*`)` is the empty set if and only if they refer to *arrays* of different, distinct specialized representations.

Therefore,

```
(subtypep '(array T1) '(array T2)) → true
```

if and only if

```
(upgraded-array-element-type 'T1)  and
(upgraded-array-element-type 'T2)
```

subtypep

return two different *type specifiers* that always refer to the same sets of *objects*.

For all type-specifiers *T1* and *T2* other than `*`,

```
(subtypep '(complex T1) '(complex T2)) → true, true
```

if:

1. `T1` is a *subtype* of `T2`, or
2. `(upgraded-complex-part-type 'T1)` and `(upgraded-complex-part-type 'T2)` return two different *type specifiers* that always refer to the same sets of *objects*; in this case, `(complex T1)` and `(complex T2)` both refer to the same specialized representation.

The *values* are *false* and *true* otherwise.

The form

```
(subtypep '(complex single-float) '(complex float))
```

must return *true* in all implementations, but

```
(subtypep '(array single-float) '(array float))
```

returns *true* only in implementations that do not have a specialized *array* representation for *single floats* distinct from that for other *floats*.

Examples:

```
(subtypep 'compiled-function 'function) → true, true
(subtypep 'null 'list) → true, true
(subtypep 'null 'symbol) → true, true
(subtypep 'integer 'string) → false, true
(subtypep '(satisfies dummy) nil) → false, implementation-dependent
(subtypep '(integer 1 3) '(integer 1 4)) → true, true
(subtypep '(integer (0) (0)) 'nil) → true, true
(subtypep 'nil '(integer (0) (0))) → true, true
(subtypep '(integer (0) (0)) '(member)) → true, true ;or false, false
(subtypep '(member) 'nil) → true, true ;or false, false
(subtypep 'nil '(member)) → true, true ;or false, false
```

Let `<aet-x>` and `<aet-y>` be two distinct *type specifiers* that do not always refer to the same sets of *objects* in a given implementation, but for which **make-array**, will return an *object* of the same *array type*.

Thus, in each case,

```
 (subtypep (array-element-type (make-array 0 :element-type '<aet-x>))
           (array-element-type (make-array 0 :element-type '<aet-y>)))
→ true, true
```

```
 (subtypep (array-element-type (make-array 0 :element-type '<aet-y>))
           (array-element-type (make-array 0 :element-type '<aet-x>)))
```
→ *true*, *true*

If `(array <aet-x>)` and `(array <aet-y>)` are different names for exactly the same set of *objects*, these names should always refer to the same sets of *objects*. That implies that the following set of tests are also true:

```
(subtypep '(array <aet-x>) '(array <aet-y>)) → true, true
(subtypep '(array <aet-y>) '(array <aet-x>)) → true, true
```

See Also:

Section 4.2 (Types)

Notes:

The small differences between the **subtypep** specification for the **array** and **complex** types are necessary because there is no creation function for *complexes* which allows the specification of the resultant part type independently of the actual types of the parts. Thus in the case of the *type* **complex**, the actual type of the parts is referred to, although a *number* can be a member of more than one *type*. For example, `17` is of *type* `(mod 18)` as well as *type* `(mod 256)` and *type* **integer**; and `2.3f5` is of *type* **single-float** as well as *type* **float**.

type-of *Function*

Syntax:

type-of *object* → *typespec*

Arguments and Values:

object—an *object*.

typespec—a *type specifier*.

Description:

Returns a *type specifier*, *typespec*, for a *type* that has the *object* as an *element*. The *typespec* satisfies the following:

1. For any *object* that is an *element* of some *built-in type*:

   a. the *type* returned is a *recognizable subtype* of that *built-in type*.

   b. the *type* returned does not involve `and`, `eql`, `member`, `not`, `or`, `satisfies`, or `values`.

type-of

2. For all *objects*, `(typep object (type-of object))` returns *true*. Implicit in this is that *type specifiers* which are not valid for use with **typep**, such as the *list* form of the **function** *type specifier*, are never returned by **type-of**.

3. The *type* returned by **type-of** is always a *recognizable subtype* of the *class* returned by **class-of**. That is,

   `(subtypep (type-of object) (class-of object))` → *true*, *true*

4. For *objects* of metaclass **structure-class** or **standard-class**, and for *conditions*, **type-of** returns the *proper name* of the *class* returned by **class-of** if it has a *proper name*, and otherwise returns the *class* itself. In particular, for *objects* created by the constructor function of a structure defined with **defstruct** without a `:type` option, **type-of** returns the structure name; and for *objects* created by **make-condition**, the *typespec* is the *name* of the *condition type*.

5. For each of the *types* **short-float**, **single-float**, **double-float**, or **long-float** of which the *object* is an *element*, the *typespec* is a *recognizable subtype* of that *type*.

Examples:

```
 (type-of 'a) → SYMBOL
 (type-of '(1 . 2))
→ CONS
or→ (CONS FIXNUM FIXNUM)
 (type-of #c(0 1))
→ COMPLEX
or→ (COMPLEX INTEGER)
 (defstruct temp-struct x y z) → TEMP-STRUCT
 (type-of (make-temp-struct)) → TEMP-STRUCT
 (type-of "abc")
→ STRING
or→ (STRING 3)
 (subtypep (type-of "abc") 'string) → true, true
 (type-of (expt 2 40))
→ BIGNUM
or→ INTEGER
or→ (INTEGER 1099511627776 1099511627776)
or→ SYSTEM::TWO-WORD-BIGNUM
or→ FIXNUM
 (subtypep (type-of 112312) 'integer) → true, true
 (defvar *foo* (make-array 5 :element-type t)) → *FOO*
 (class-name (class-of *foo*)) → VECTOR
```

```
 (type-of *foo*)
→ VECTOR
or→ (VECTOR T 5)
```

See Also:

array-element-type, **class-of**, **defstruct**, **typecase**, **typep**, Section 4.2 (Types)

Notes:

Implementors are encouraged to arrange for **type-of** to return a portable value.

typep *Function*

Syntax:

typep *object type-specifier* `&optional` *environment* → *generalized-boolean*

Arguments and Values:

object—an *object*.

type-specifier—any *type specifier* except **values**, or a *type specifier* list whose first element is either **function** or **values**.

environment—an *environment object*. The default is **nil**, denoting the *null lexical environment* and the and current *global environment*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *object* is of the *type* specified by *type-specifier*; otherwise, returns *false*.

A *type-specifier* of the form `(satisfies fn)` is handled by applying the function `fn` to *object*.

`(typep` *object* `'(array` *type-specifier*`))`, where *type-specifier* is not `*`, returns *true* if and only if *object* is an *array* that could be the result of supplying *type-specifier* as the `:element-type` argument to **make-array**. `(array *)` refers to all *arrays* regardless of element type, while `(array` *type-specifier*`)` refers only to those *arrays* that can result from giving *type-specifier* as the `:element-type` argument to **make-array**. A similar interpretation applies to `(simple-array` *type-specifier*`)` and `(vector` *type-specifier*`)`. See Section 15.1.2.1 (Array Upgrading).

`(typep` *object* `'(complex` *type-specifier*`))` returns *true* for all *complex* numbers that can result from giving *numbers* of type *type-specifier* to the *function* **complex**, plus all other *complex* numbers of the same specialized representation. Both the real and the imaginary parts of any such *complex* number must satisfy:

```
 (typep realpart 'type-specifier)
```

typep

```
(typep imagpart 'type-specifier)
```

See the *function* **upgraded-complex-part-type**.

Examples:

```
(typep 12 'integer) → true
(typep (1+ most-positive-fixnum) 'fixnum) → false
(typep nil t) → true
(typep nil nil) → false
(typep 1 '(mod 2)) → true
(typep #c(1 1) '(complex (eql 1))) → true
;; To understand this next example, you might need to refer to
;; Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).
(typep #c(0 0) '(complex (eql 0))) → false
```

Let $\mathtt{A}_x$ and $\mathtt{A}_y$ be two *type specifiers* that denote different *types*, but for which

```
(upgraded-array-element-type 'Ax)
```

and

```
(upgraded-array-element-type 'Ay)
```

denote the same *type*. Notice that

```
(typep (make-array 0 :element-type 'Ax) '(array Ax)) → true
(typep (make-array 0 :element-type 'Ay) '(array Ay)) → true
(typep (make-array 0 :element-type 'Ax) '(array Ay)) → true
(typep (make-array 0 :element-type 'Ay) '(array Ax)) → true
```

Exceptional Situations:

An error of *type* **error** is signaled if *type-specifier* is `values`, or a *type specifier* list whose first element is either **function** or **values**.

The consequences are undefined if the *type-specifier* is not a *type specifier*.

See Also:

type-of, **upgraded-array-element-type**, **upgraded-complex-part-type**, Section 4.2.3 (Type Specifiers)

Notes:

Implementations are encouraged to recognize and optimize the case of `(typep x (the class y))`, since it does not involve any need for expansion of **deftype** information at runtime.

type-error *Condition Type*

Class Precedence List:

type-error, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **type-error** represents a situation in which an *object* is not of the expected type. The "offending datum" and "expected type" are initialized by the initialization arguments named `:datum` and `:expected-type` to **make-condition**, and are *accessed* by the functions **type-error-datum** and **type-error-expected-type**.

See Also:

type-error-datum, **type-error-expected-type**

type-error-datum, type-error-expected-type *Function*

Syntax:

type-error-datum *condition* → *datum*

type-error-expected-type *condition* → *expected-type*

Arguments and Values:

condition—a *condition* of *type* **type-error**.

datum—an *object*.

expected-type—a *type specifier*.

Description:

type-error-datum returns the offending datum in the *situation* represented by the *condition*.

type-error-expected-type returns the expected type of the offending datum in the *situation* represented by the *condition*.

Examples:

```
(defun fix-digits (condition)
  (check-type condition type-error)
  (let* ((digits '(zero one two three four
                  five six seven eight nine))
```

```
        (val (position (type-error-datum condition) digits)))
    (if (and val (subtypep 'fixnum (type-error-expected-type condition)))
        (store-value 7))))

(defun foo (x)
  (handler-bind ((type-error #'fix-digits))
    (check-type x number)
    (+ x 3)))

(foo 'seven)
→ 10
```

See Also:

type-error, Chapter 9 (Conditions)

simple-type-error *Condition Type*

Class Precedence List:

simple-type-error, **simple-condition**, **type-error**, **error**, **serious-condition**, **condition**, **t**

Description:

Conditions of *type* **simple-type-error** are like *conditions* of *type* **type-error**, except that they provide an alternate mechanism for specifying how the *condition* is to be *reported*; see the *type* **simple-condition**.

See Also:

simple-condition, **simple-condition-format-control**, **simple-condition-format-arguments**, **type-error-datum**, **type-error-expected-type**