Programming Language ISLISP
ISLISP Working Draft 23.0

This document was created Thu 17-Mar-2007 00:10am JST.

Contents

This page intentionally left blank.

Introduction

The programming language ISLisp is a member of the Lisp family.

The following factors influenced the establishment of design goals for ISLisp:

1. A desire of the international Lisp community to standardize on those features of Lisp upon which there is widespread agreement.
2. The existence of the incompatible dialects Common-Lisp, EuLisp, Le-Lisp, and Scheme (mentioned in alphabetical order).
3. A desire to affirm Lisp as an industrial language.

This led to the following design goals for ISLisp:

1. ISLisp shall be compatible with existing Lisp dialects where feasible.
2. ISLisp shall have as a primary goal to provide basic functionality.
3. ISLisp shall be object-oriented.
4. ISLisp shall be designed with extensibility in mind.
5. ISLisp shall give priority to industrial needs over academic needs.
6. ISLisp shall promote efficient implementations and applications.

Programming Language ISLISP

1 Scope

This document specifies syntax and semantics of the computer programming language ISLISP by specifying requirements for a conforming ISLISP processor and a conforming ISLISP text.

This document does not specify:

(a) the size or complexity of an ISLISP text that exceeds the capacity of any specific data processing system or the capacity of a particular processor, nor the actions to be taken when the corresponding limits are exceeded;

(b) the minimal requirements of a data processing system that is capable of supporting an implementation of a processor for ISLISP;

(c) the method of preparation of an ISLISP text for execution and the method of activation of this ISLISP text, prepared for execution;

(d) the typographical presentation of an ISLISP text published for human reading.

(e) extensions that might or might not be provided by the implementation.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC TR 10034:1990, *Guidelines for the preparation of conformity clauses in programming language standards.*
- IEEE standard 754-1985. *Standard for binary floating point arithmetic.*

3 Compliance of ISLISP processors and text

An ISLISP processor complying with the requirements of this document shall

(a) accept and implement all features of ISLISP specified in this document.

(b) reject any text that contains any textual usage which this document explicitly defines to be a violation (see §9).

(c) be accompanied by a document that provides the definitions of all implementation-defined features.

(d) be accompanied by a document that separately describes any features accepted by the processor that are not specified in this document; these extensions shall be described as being "extensions to ISLisp as specified by ISLISP Working Draft 23.0."

A complying ISLisp text shall not rely on implementation-dependent features. However, a complying ISLisp text may rely on implementation-defined features required by this document.

A complying ISLisp text shall not attempt to create a lexical variable binding for any named constant defined in this document. It is a violation if any such attempt is made.

4 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

4.1
abstract class
class that by definition has no direct instances

4.2
activation
computation of a function

Note: Every activation has an activation point, an activation period, and an activation end. The activator, which is a function application form prepared for execution, starts the activation at the activation point.

4.3
accessor
association of a reader and a writer for a slot of an instance

4.4
argument position
occurrence of a text unit as an element in a form excluding the first one

4.5
binding
concept that has both a syntactic and a semantic aspect, where

- syntactically, "binding" describes the relation between an identifier and a binding ISLISP form, and
- semantically, "binding" describes the relation between a variable, its denoting identifier, and an object (or, the relation between a variable and a location)

Note 1: The property of being bound can be checked textually by relating defining and applied identifier occurrences.

Note 2: Semantically, the binding relation might be imagined to be materialized in some entity, the binding. Such a binding entity is constructed at run time and destroyed later, or might have indefinite extent.

4.6
class
object that determines the structure and behavior of a set of other objects called its instances

Note: The behavior is the set of operations that can be performed on an instance.

4.7
condition
object that represents a situation that has been (or might be) detected by a running program

4.8
definition point
textual point of an ISLISP text that is the start of an identifier's representation of an ISLISP object

4.9
direct instance
instance of a class but not an instance of one of its subclasses

Note: Every ISLISP object is direct instance of exactly one class, which is called "its class". The set of all direct instances together with their behavior constitute a class.

4.10
dynamic
having an effect that is determined only through program execution and that cannot, in general, be determined statically

4.11
dynamic variable
variable whose associated binding is determined by the most recently executed active block that established it, rather than statically by a lexically apparent block according to the lexical principle

4.12
evaluation
computation of a form prepared for execution which results in a value and/or a side-effect

4.13
execution
sequence of (sometimes nested) activations

4.14
extension
implementation-defined modification to the requirements of this document that does not invalidate any ISLISP text complying with this document (except by prohibiting the use of one or more particular spellings of identifiers), does not alter the set of actions which are required to signal errors, and does not alter the status of any feature designated as implementation dependent

4.15
form
single, syntactically valid unit of program text, capable of being prepared for execution

4.16
function
ISLISP object that is called with arguments, performs a computation (possibly having side-effects), and returns a value

4.17
generic function
function whose application behavior is determined by the classes of the values of its arguments and which consists – in general – of several methods

4.18
identifier
lexical element (lexeme) which designates an ISLISP object

Note: In the data structure representation of ISLISP texts, identifiers are denoted by symbols.

4.19
immutable binding
binding in which the relation between an identifier and the object represented by this identifier cannot be changed

Note: It is a violation if there is attempt to change an immutable binding (error-id. *immutable-binding*).

4.20
immutable object
object which is not subject to change, either because no operator is provided that 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

Note: Except as explicitly indicated otherwise, a conforming processor is not required to detect attempts to modify immutable objects; the consequences are undefined if an attempt is made to modify an immutable object.

4.21
implementation defined
feature, possibly differing between different ISLISP processors, but completely defined for every processor

4.22
implementation dependent
feature, possibly differing between different ISLISP processors, but not necessarily defined for any particular processor

Note: A conforming ISLISP text must not depend upon implementation-dependent features.

4.23
inheritance
relation between a class and its superclass which maps structure and behavior of the superclass onto the class

Note: ISLISP supports a restricted form of multiple inheritance; *i.e.,* a class may have several direct superclasses at once.

4.24
instance
⟨class⟩ either a direct instance of a class or an instance of one of its subclasses

4.25
literal
object whose representation occurs directly in a program as a constant value

4.26
metaclass
class whose instances are themselves classes

4.27
method
case of a generic function for a particular parameter profile, which defines the class-specific behavior and operations of the generic function

4.28
object
anything that can be created, destroyed, manipulated, compared, stored, input, or output by the ISLISP processor

Note 1: In particular, functions are ISLISP objects.

Note 2: Objects that can be passed as arguments to functions, can be returned as values, can be bound to variables, and can be part of structures, are called *first-class objects*.

4.29
operator
first element of a compound form, which is either a reserved name that identifies the form as a special form, or the name of a macro, or a lambda expression, or else an identifier in the function namespace

4.30
operator position
occurrence of a text unit as the first element in a form

4.31
parameter profile
parameter list of a method, where each formal parameter is accompanied by its class name

Note: If a parameter is not accompanied by a class name, it belongs to the most general class.

4.32
place
location where objects can be stored and retrieved later

Note: Places are designated by forms which are permitted as the first argument of `setf`. If used this way an object is stored in the place. If the form is not used as first argument of `setf` the stored object is retrieved. The cases are listed in the description of `setf`.

4.33
process
execution of an ISLISP text prepared for execution

4.34
processor
system or mechanism, that accepts an ISLISP text (or an equivalent data structure) as input, prepares it for execution, and executes the result to produce values and side-effects

4.35
program
aggregation of expressions to be evaluated, the specific nature of which depends on context

Note: Within this document, the term "program" is used only in an abstract way; there is no specific syntactic construct that delineates a program.

4.36
scope
⟨identifier⟩ the textual part of a program where the meaning of that identifier is defined; *i.e.,* there exists an ISLISP object designated by this identifier

4.37
slot
named component of an instance which can be accessed using the slot accessors

Note: The structure of an instance is defined by the set of its slots.

4.38
text
text that complies with the requirements of this document (*i.e.,* with the syntax and static semantics of ISLISP)

Note: An ISLISP text consists of a sequence of toplevel forms.

4.39
toplevel form
any form that either is not nested in any other form or is nested only in `progn` forms

4.40
toplevel scope
scope in which a complete ISLISP text unit is processed

4.41
writer
method associated with a slot of a class, whose task is to bind a value with a slot of an instance of that class

5 Notation and conventions

For a clear definition of, and a distinction between, syntactic and semantic concepts, several levels of description abstraction are used in the following.

There is a correspondence from ISLISP textual units to their ISLISP data structure representations. Throughout this document the text and the corresponding ISLISP objects (data structures) are addressed simultaneously. ISLISP text can be seen as an external specification of ISLISP data structures. To distinguish between the two representations different concepts are used. When textual representation is discussed, textual elements (such as *identifiers*, *literals*, and *compound forms*) are used; when ISLISP objects are discussed, *objects* (such as *symbols* and *lists*) are used.

The constituents of ISLISP text are called ***forms***. A ***form*** can be an *identifier*, a *literal*, or a *compound form*. A *compound form* can be a *function application form*, a *macro form*, a *special form*, or a *defining form*.

An *identifier* is represented by a *symbol*. A *compound form* is represented by a non-null *list*. A *literal* is represented by neither a symbol nor a list, and so is neither an *identifier* nor a *compound form*; for example, a number is a literal.

An object is ***prepared for execution***; this might include transformation or compilation, including macro expansion. The method of preparation for execution and its result are not defined in this document (with exception of the violations to be detected). After successful preparation for execution the result is ready for ***execution***. The combination of preparation for execution and subsequent execution implements ISLISP's ***evaluation model***. The term "evaluation" is used because ISLISP is an expression language—each form has a value which is used to compute the value of the containing form. The results obtained when an entity is prepared for execution are designated throughout this document by the construction "prepared entity"; *e.g.,* "prepared form," "prepared special form."

Example: A "`cond` special form" becomes a "prepared `cond`" by preparation for execution.

In the examples, the metasymbol "⇒" designates the result of an actual evaluation. For example:

$$\texttt{(+ 3 4)} \Rightarrow \texttt{7}$$

The metasymbol "→" identifies the class that results from the evaluation of a form having a given pattern. For example:

$$\texttt{(+ } i_1 \ \ i_2\texttt{)} \rightarrow \texttt{<}integer\texttt{>}$$

Given a form pattern (usually defined by its constant parts, the function name or special operator), → relates it to the class to which the result of the evaluation of all matching forms belong.

Form patterns or forms which are equivalent are related by ≡.

The following notational conventions for form patterns are used:

(f-name *argument**) → *result-class* **f kind**

In this notation, words written in *italics* are non-terminal (pattern variables). f-name is always terminal: Specific function names, special operators, defining form names, or generic function names are always presented.

An underlined term (like the *name* in a defining form) in this notation, indicates an expression that is not evaluated. If a form might or might not be evaluated (like one of the *then-form* or *else-form* in an if), this is indicated explicitly in the text.

Class names are uniformly denoted as follows: <*class-name*>. For example, <list> is the name of a class; this is usually spoken aloud as "list class."

Notes, appearing as **Note:** note-text, in this document have no effect on the language. They are for better understanding by the human reader.

Regarding the pattern variables and the extensions of above, the following conventions are also adopted:

$term^+$	denotes one or more occurrences of *term*;
$term^*$	denotes zero or more occurrences of *term*;
[*term*]	denotes at most one occurrence of *term*, commonly one says that *term* is optional;
$\{term_1\ term_2 \ldots\}$	denotes grouping of *terms*.
$term_1 \mid term_2 \mid \ldots$	denotes grouping of alternative *terms*.

The following naming conventions are used to denote forms whose arguments obey the respective class restrictions:

$array$, $array_1$, … $array_j$, …	<basic-array>
$cons$, $cons_1$, … $cons_j$, …	<cons>
$list$, $list_1$, … $list_j$, …	<list>
obj, obj_1, … obj_j, …	<object>
$sequence$, $sequence_1$, … $sequence_j$, …	<basic-vector> or <list> (see §25)
$stream$, $stream_1$, … $stream_j$, …	<stream>
$string$, $string_1$, … $string_j$, …	<string>
$char$, $char_1$, … $char_j$, …	<character>
$function$, $function_1$, … $function_j$, …	<function>

class, $class_1$, … $class_j$, …	<class>
symbol, $symbol_1$, … $symbol_j$, …	<symbol>
x, x_1, … x_j, …	<number>
z, z_1, … z_j, …	<integer>

In this document the conventions detailed below are used, except where noted:

- `-p` Predicates—sometimes called "boolean functions"—usually have names that end in a `-p`. Usually every class <*name*> has a characteristic function, whose name is built as *name*-p if *name* is hyphenated (`generic-function-p`), or *name*p if *name* is not hyphenated (`symbolp`). Note that not all functions whose names end with "p" are predicates.
- `create-` Usually a built-in class <*name*> has a constructor function, which is called `create-`*name*.
- `def` This is used as the prefix of the defining operators.
- `set-` Within this document, any functions named `set-`*name* are writers for a place, for which there is a corresponding reader named *name*.

For any kind of entity in the language, the phrase "*entity-kind name*" refers to the entity of kind *entity-kind* denoted by *name*. For example, the phrases "function *name*," "constant *name*," or "class *name*" respectively mean the function, constant, or class denoted by *name*.

6 Lexemes

An ISLisp text is built up from lexemes. Lexemes are built up from at least the following characters (see §20):

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 + - < > / * & = . ? _ ! $ % : @ [ ] ^ { } ~ #
```

Additional characters are implementation defined.

The following characters are individual lexemes (see §16 and §21.1):

```
( ) , ' `
```

The following character tuples (where n is a sequence of digits) are individual lexemes (see §12.7, §16, and §22.1):

```
#' #( ,@ #B #b #O #o #X #x #na #nA
```

The textual representations of symbols (see §18), numbers (see §19), characters (see §20), and strings (see §24) are lexemes.

\ (single escape) and | (multiple escape) are special characters. They may occur in some lexemes (identifiers and string literals).

Other lexemes are separated by delimiters. Delimiters are separators along with the following characters:

( ) ‘ , ’

The effect of delimiting is disestablished inside a string (see §24) or inside a corresponding pair of multiple escape characters (see §18) or for the character immediately following `#\`.

6.1 Separators

Separators are as follows: blank, comments, newline, and an implementation-defined set of characters, (*e.g.,* tabs). Separators have no meaning and can be replaced by each other without changing the meaning of the ISLISP text.

6.2 Comments

The character semicolon (;) is the ***comment begin*** character. That is, the semicolon and all the characters up to and including the end-of-line form a comment.

A character sequence beginning with `#|` and ending with `|#` is a comment. Such comments may be nested.

Being a separator, a comment cannot occur inside a lexeme.

7 Textual representation

The textual representation of an object is machine independent. The following are some of the textual representations of the ISLISP objects. This representation is readable by the `read` function. Lexemes are described in §6

Null The object `nil` is the only object whose class is `<null>`. Upon input, it may be written as `nil` or `()`. It is implementation defined whether `nil` prints as `nil` or `()`.

List Proper lists are those lists terminated by `nil`. Usually they are denoted as (obj_1 obj_2 $\ldots obj_n$). A dotted list (*i.e.,* a list whose last tail is not `nil`) appears as (obj_1 obj_2 $\ldots obj_n$. obj_{n+1}).

Character An instance of the `<character>` class is represented by `#\?`, where “?” is the character in question. There are two special standard characters that are not represented in this way, namely *newline* and *space*, whose representations are `#\newline` and `#\space`, respectively.

Cons A cons is expressed as (*car* . *cdr*), where the *car* and *cdr* are objects.

Integer An integer (radix 10) is represented as a sequence of digits optionally preceded by a `+` or `-` sign. If the number is represented in binary radix (or in octal or hexadecimal) then the textual representation is preceded by `#b` (or `#o` or `#x`, respectively).

Float A floating point number is written in one of the following formats:

$[s]dd\ldots d\texttt{.}dd\ldots d$
$[s]dd\ldots d\texttt{.}dd\ldots d\texttt{E}[s]dd\ldots d$
$[s]dd\ldots d\texttt{.}dd\ldots d\texttt{e}[s]dd\ldots d$
$[s]dd\ldots d\texttt{E}[s]dd\ldots d$
$[s]dd\ldots d\texttt{e}[s]dd\ldots d$

where s is either "+" or "-", and d is one of "0"–"9". For example: `987.12`, `+12.5E-13`, `-1.5E12`, `1E32`[1].

Vector A vector of class `<general-vector>` is written as `#(`$obj_1 \ldots obj_n$`)`.

Array An array of class `<general-array*>` or `<general-vector>` can be written on input as `#`n`a` (where n is an integer indicating the number of dimensions of the array) followed by a nested structure of sequences denoting the contents of the array. This structure is defined as follows. If $n = 1$ the structure is simply `(`obj_1 `...` obj_n`)`. If $n > 1$ and the dimensions are n_1 n_2 `...`, the structure is `(`str_1 `...` str_{n_1}`)`, where the str_i are the structures of the n_1 subarrays, each of which has dimensions `(`n_2 `...)`. As an example, the representation of `(create-array '(2 3 4) 5)` is as follows:
`#3a(((5 5 5 5) (5 5 5 5) (5 5 5 5)) ((5 5 5 5) (5 5 5 5) (5 5 5 5))).`

On output (see `format`), arrays of class `<general-vector>` will be printed using `#(...)` notation.

String A string is represented by the sequence of its characters enclosed in a pair of `"`'s. For example: `"abc"`. Special characters are preceded with a backslash as an escape character.

Symbol A named symbol is represented by its print name. Vertical bars (`|`) might need to enclose the symbol if it contains certain special characters; see §18. The notation, if any, used for unnamed symbols is implementation defined.

There are objects which do not have a textual representation, such as a class or an instance of the `<function>` class.

8 Reserved identifiers

Symbols whose names contain a colon (`:`) or an ampersand (`&`) are reserved and may not be used as identifiers. Symbols whose names start with colon (`:`) are called ***keywords***.

9 Errors

An ***error*** is a situation arising during execution in which the processor is unable to continue correct execution according to the semantics defined in this document. The act of detecting and reporting such an error is called ***signaling*** the error.

A ***violation*** is a situation arising during preparation for execution in which the textual requirements of this document are not met. A violation shall be detected during preparation for execution.

[1] This number, although belonging to the set of natural numbers, usually is considered as only a floating point number because of its representation.

9.1 Classes of error specification

The wording of error specification in this document is as follows:

(a) "an error shall be signaled"

An implementation shall detect an error of this kind no later than the completion of execution of the form having the error, but might detect them sooner (*e.g.,* when the code is being prepared for execution).

Evaluation of the current form shall stop. If no active handler is established by `with-handler`, it is implementation defined whether the entire running process exits, a debugger is entered, or control is transferred elsewhere within the process.

(b) "the consequences are undefined"

This means that the consequences are unpredictable. The consequences may range from harmless to fatal. No conforming ISLisp text may depend on the results or effects. A conforming ISLisp text must treat the consequences as unpredictable. In places where "must," "must not," or "may not" are used, then this is equivalent to stating that "the consequences are undefined" if the stated requirement is not met and no specific consequence is explicitly stated. An implementation is permitted to signal an error in this case.

For indexing and cross-referencing convenience, errors in this document have an associated *error identification* label, notated by text such as "(error-id. *sample*)." The text of these labels has no formal significance to ISLisp texts or processors; the actual class of any object which might be used by the implementation to represent the error and the text of any error message that might be displayed is implementation dependent.

9.2 Pervasive error types

Most errors are described in detail in the contect in which they occur. Some error types are so pervasive that their detailed descriptions are consolidated here rather than repeated in full detail upon each occurrence.

1. Domain error: an error shall be signaled if the object given as argument of a standard function for which a class restriction is in effect is not an instance of the class which is required in the definition of the function (error-id. *domain-error*).
2. Arity error: an error shall be signaled if a function is activated with a number of arguments which is different than the number of parameters as required in the function definition (error-id. *arity-error*).
3. Undefined entity error: an error shall be signaled if the entity denoted by an identifier does not exist when a reference to that entity is made (error-id. *undefined-entity*). Two commonly occuring examples of this type of error are *undefined-function* and *unbound-variable*.

This list does not exhaust the space of error types. For a more complete list, see §29.4.

10 Classes

In ISLISP, data types are covered by the class system. A ***class*** is an object that determines the structure and behavior of a set of other objects, which are called its ***instances***. Every ISLISP object is an instance of a class. The behavior is the set of operations that can be performed on an instance.

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 be named by an *identifier*. For example, this *identifier* can be used as a parameter specializer in method definitions. The `class` special form can be used to refer to access the class object corresponding to its name.

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, or if C_1 is defined by this document to be a direct superclass of C_2 (for example, by use of a directed arrow from C_1 to C_2 in Figure 1). 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."

If a user-defined class C inherits from two classes, C_1 and C_2, the only superclasses that C_1 and C_2 may have in common are `<standard-object>` or `<object>`. This allows a restricted form of multiple inheritance.

Every ISLISP object is a *direct instance* of exactly one class which is called "its" class.

An *instance* of a class is either a direct instance of that class or an instance of one of its subclasses.

Classes are organized into a ***directed acyclic graph*** defined by the subclass relation. The nodes are classes and there is an edge from C_1 to C_2 iff C_2 is direct subclass of C_1. This graph is called the *inheritance graph*. It has as root the class `<object>`, the only class with no superclass. Therefore it is the 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 `<standard-class>` except itself.

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***, or override, 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.

10.1 Metaclasses

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. The term *metaclass* is used to refer to a

```
<object>
├─> <basic-array>
│   ├─> <basic-array*>
│   │   └─> <general-array*>
│   └─> <basic-vector>
│       ├─> <general-vector>
│       └─> <string>
├─> <built-in-class>
├─> <character>
├─> <function>
│   └─> <generic-function>
│       └─> <standard-generic-function>
├─> <list>
│   ├─> <cons>
│   └─> <null>
├─> <symbol>
│   └─> <null>
├─> <number>
│   ├─> <float>
│   └─> <integer>
├─> <serious-condition>
│   ├─> <error>
│   │   ├─> <arithmetic-error>
│   │   │   ├─> <division-by-zero>
│   │   │   ├─> <floating-point-overflow>
│   │   │   └─> <floating-point-underflow>
│   │   ├─> <control-error>
│   │   ├─> <parse-error>
│   │   ├─> <program-error>
│   │   │   ├─> <domain-error>
│   │   │   └─> <undefined-entity>
│   │   │       ├─> <unbound-variable>
│   │   │       └─> <undefined-function>
│   │   ├─> <simple-error>
│   │   └─> <stream-error>
│   │       └─> <end-of-stream>
│   └─> <storage-exhausted>
├─> <standard-class>
├─> <standard-object>
└─> <stream>
```

Figure 1. Class Inheritance

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 ISLisp Object System provides the following predefined metaclasses:

- 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 or restricted capabilities. For example, it is not possible to define subclasses of a built-in class.

10.2 Predefined classes

The following classes are primitive classes in the class system (*i.e.,* predefined classes that are not metaclasses):

<arithmetic-error>	<floating-point-underflow>	<simple-error>
<basic-array>	<function>	<standard-generic-function>
<basic-array*>	<general-array*>	<standard-object>
<basic-vector>	<general-vector>	<storage-exhausted>
<character>	<generic-function>	<stream>
<cons>	<integer>	<stream-error>
<control-error>	<list>	<string>
<division-by-zero>	<null>	<symbol>
<domain-error>	<number>	<unbound-variable>
<end-of-stream>	<object>	<undefined-entity>
<error>	<parse-error>	<undefined-function>
<float>	<program-error>	
<floating-point-overflow>	<serious-condition>	

The classes <standard-class> and <built-in-class> are predefined metaclasses.

A user-defined class, defined by defclass, must be implemented as an instance of <standard-class>. A predefined class can be implemented either as an instance of <standard-class> (as if defined by defclass) or as an instance of <built-in-class>.

Figure 1 shows the required inheritance relationships among the classes defined by ISLisp. For each pair of classes C_1 and C_2 in this figure, if C_1 is linked directly by an arrow to C_2, C_1 is a direct superclass of C_2 (and C_2 is a direct subclass of C_1). Additional relationships might exist, subject to the following constraints:

1. It is implementation defined whether <standard-generic-function> is a subclass of the class <standard-object>.
2. Except as described in Figure 1 and the above constraint on <standard-generic-function>, no other subclass relationships exist among the classes defined in this document. However, additional implementation-specific subclass

relationships may exist between implementation-specific classes and classes defined in this document.

3. The class precedence list for `<null>` observes the partial order `<null>`, `<symbol>`, `<list>`, `<object>`.

4. Users may define additional classes using `defclass`.

A built-in class is one whose instances have restricted capabilities or special representations. The `defclass` defining form must not be used to define subclasses of a built-in class. An error shall be signaled if `create` is called to create an instance of a built-in class.

A standard class is an instance of `<standard-class>`, and a built-in class is an instance of `<built-in-class>`.

A standard class defined with no direct superclasses is guaranteed to be disjoint from all of the classes in the figure, except for the classes named `<standard-object>` and `<object>`.

The class `<function>` is the class of all functions. The class `<standard-generic-function>` is the default class of all generic functions.

10.3 Standard classes

10.3.1 Slots

An object that has `<standard-class>` as its metaclass has zero or more named slots. The slots of an object are determined by the class of the object. Each slot can hold one object as its value. The name of a slot is an identifier.

When a slot does not have a value, the slot is said to be ***unbound***. The consequences are undefined if an attempt is made to retrieve the value of an unbound slot.

Storing and retrieving the value of a slot is done by generic functions defined by the `defclass` defining form.

All slots are local; *i.e.,* there are no shared slots accessible by several instances.

A class is said to ***define*** a slot with a given name when the `defclass` defining form for that class contains a slot specifier with that name. Defining a slot does not immediately create a slot; it causes a slot to be created each time an instance of the class is created.

A slot is said to be ***accessible*** in an instance of a class if the slot is defined by the class of the instance or is inherited from a superclass of that class. At most one slot of a given name can be accessible in an instance. A detailed explanation of the inheritance of slots is given in the section §15.1.3.

10.3.2 Creating instances of classes

The generic function `create` creates and returns a new instance of a class. ISLISP 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 by providing default initial values. Further initialization activities can be performed by methods written for generic functions that are part of the initialization protocol.

11 Scope and extent

In describing ISLisp, the notions of *scope* and *extent* are useful. The first is a syntactic concept, the latter is a semantic concept. Although syntactic constructs, especially identifiers, are used to refer to runtime entities (*i.e.,* objects arising during execution), a single entity cannot have both scope and extent. *Scope* is a feature of an identifier, referring to that textual part of an ISLisp text (see §4.38 and §5) within which this identifier occurs with unique meaning. *Extent* refers to the interval of execution time during which a certain object exists.

A ***namespace*** is a mapping from identifiers to meanings. In ISLisp there are six namespaces: variable, dynamic variable, function, class, block, and tagbody tag. It is therefore possible for a single identifier to have any or all of these six meanings, depending on the context. For example, an identifier's meaning is determined by the function namespace when the identifier appears in the operator position of a function application form, whereas the same identifier's meaning is determined by the variable namespace if it appears in an argument position in the same form.

11.1 The lexical principle

ISLisp is designed following the principle of ***lexical visibility***. This principle states that an ISLisp text must be structured in properly nested lexical blocks of visibility. Within a block, all defined identifiers of that block and of all enclosing outer blocks are visible. Each identifier in a namespace has the meaning determined by the innermost block that defines it.

ISLisp also supports a form of ***dynamic binding***. Dynamic bindings are established and accessed by a separate mechanism (*i.e.,* `defdynamic`, `dynamic-let`, and `dynamic`). The dynamic value associated with such an identifier is the one that was established by the most recently executed ***active block*** that established it, where an *active* block is one that has been established and not yet disestablished. Because a separate mechanism is used, the lexical meaning of and the dynamic value associated with an identifier are simultaneously accessible wherever both are defined.

11.2 Scope of identifiers

The ***scope*** of an identifier is that part of an ISLisp text where the meaning of the identifier is defined. It starts textually with the definition point—a point that is specified individually for each form that establishes an identifier. Only identifiers can have a scope.

For each namespace, if an identifier has scope s_a and an identical identifier (in the same namespace) has nested scope s_b, then the scope s_b of the inner identifier and every scope contained in it are not part of the scope s_a. It is said that the inner scope ***shadows*** the outer scope.

Each complete ISLisp text unit is processed in a scope called the ***toplevel scope***.

In each namespace, nested binding forms shadow outer binding forms and defining forms.

```
(let ((a1 f-a1)
      ...
      (x f-x)         ...
      (z1 f-z1))
      ...             ; now a1...x...z1 are applicable, their scope begins here
   (let ((a2 f-a2)    ; a1...x...z1 might be defined newly, but:
          ...         ; the outer a1...x...z1 are still usable
          (x f-x2)    ; the inner a2...x...z2 are not yet usable
          ...
          (z2 f-z2))  ; the scope of the outer x becomes shadowed
                      ; the scope for the inner a2...x...z2 starts
      ...             ; now outer a1, z1 and inner a2...x...z2 are applicable
   )                  ; scopes of a2...x...z2 end here
   ...                ; scope of outer x becomes unshadowed
)                     ; scopes of a1...x...z1 end here
```

Figure 2. Scope Example

11.3 Some specific scope rules

The toplevel scope is the scope of identifiers of required built-in functions, special operators, defining operators, and constants.

Reserved identifiers are not subject to the lexical principle, because they are not identifiers. They cannot be defined or bound. See §8.

11.4 Extent

Complementary to scope which is a syntactic concepts, ***extent*** is a semantic concept: It describes the lifetime of entities.

Objects are created at some time during execution. In most cases, it is undetermined when an object ends its existence: its lifetime begins when the object is created and ends when reference to it is no longer possible (and the object is subject to garbage collection). In this case the object is said to have ***indefinite extent***.

In other cases the processor creates entities that are associated with prepared text. The lifetime of such objects begins at the activation point of a defining construct and ends at the end of activation; in this case the object is said to have ***dynamic extent***.

During execution, defining forms and the following binding forms create bindings at their activation points:

`block`	`let`	`with-open-io-file`
`dynamic-let`	`let*`	`with-open-output-file`
`flet`	`tagbody`	`with-standard-input`
`for`	`with-error-output`	`with-standard-output`
`labels`	`with-open-input-file`	

The bindings established by defining forms may have indefinite extent. Even in local binding constructs, bindings might not vanish upon activation end of the prepared block—if one or more function objects are created during execution of the prepared block that contain references to those bindings, the bindings will have a lifetime equal to the longest lifetime of those function objects.

Example:

```
(defun copy-cell (x) (cons (car x) (cdr x)))
```

The scope of the identifier `x` is the body alone—*i.e.,* `(cons (car x) (cdr x))`. The meaning of `x` is defined for the entire body. `x`, as identifier, cannot have an extent. The `defun` form for `copy-cell` is prepared for execution and thereby `copy-cell` becomes a prepared function. During execution the prepared function `copy-cell` might be activated. Activation in this case results in the creation of a binding between the variable denoted by `x` and the object which is used as argument. The binding of `x` is an entity whose extent lasts from the activation point to the activation end of the function. (In general the extent of a binding can last beyond the activation end, but this does not occur in this simple case.) We say that the binding of `x` is ***established*** upon activation of the function and is ***disestablished*** at activation end.

12 Forms and evaluation

12.1 Forms

Execution presupposes successful preparation for execution of an ISLisp text subject to the evaluation model. Execution is an activation of a prepared text form that results in a value and perhaps in some side-effects.

An ISLisp text is a sequence of forms.

Throughout this document the value a form returns is described, but in general a form might not return if one of its subforms executes a *non-local exit* (see §14.7.1). Therefore, it should be understood that all such descriptions implicitly include the provision that *if the form returns*, a particular value is returned.

The following are valid forms in ISLisp:

- Compound forms
  - Special forms
  - Defining forms
  - Function application forms
  - Macro forms
- Identifiers
- Literals

A form, when evaluated, returns an object as its value, though some forms may not return (*e.g.,* `return-from`).

A compound form is written as `(`*operator argument**`)`. The *operator* must be a special operator, a defining operator, an identifier, or a lambda expression. The identifier names a function, a macro, or a generic function. It is a violation if *operator* is a literal.

A ***toplevel form*** is a form that is either not lexically nested within another form or is lexically nested only within one or more `progn` forms. Special forms and function application forms at toplevel are called ***set-up forms***. It is a violation if a defining form is not a toplevel form.

12.2 Function application forms

A ***function application form*** is a compound form whose operator is an identifier (naming a function) or whose operator is a lambda expression. All of the arguments are evaluated, from left to right, and the function is called with (or "applied to") arguments that are, in the same order, the objects resulting from these evaluations. This document describes a function application form in the following format:

`(`*function-name argument**`)` → *result-class* **function**

This describes an ordinary function.

`(`*generic-function-name argument**`)` → *result-class* **generic function**

This describes a generic function.

`(`*local-function-name argument**`)` → *result-class* **local function**

This describes an ordinary function that is available only in a specified lexical scope.

12.3 Special forms

A ***special form*** is a form whose arguments are treated in a special way; for example, arguments are not evaluated or are evaluated in a special order. It is implementation defined whether any special form is implemented as a macro (see §12.5 and §16). Special forms are recognized because they have a ***special operator*** in their operator position. The following are special operators:

`and`	`dynamic-let`	`let*`	`throw`
`assure`	`flet`	`or`	`unwind-protect`
`block`	`for`	`progn`	`while`
`case`	`function`	`quote`	`with-error-output`
`case-using`	`go`	`return-from`	`with-handler`
`catch`	`if`	`set-dynamic`	`with-open-input-file`
`class`	`ignore-errors`	`setf`	`with-open-io-file`
`cond`	`labels`	`setq`	`with-open-output-file`
`convert`	`lambda`	`tagbody`	`with-standard-input`
`dynamic`	`let`	`the`	`with-standard-output`

There might be additional, implementation-defined special operators.

This document describes the evaluation of special forms in the following format:

(*special-operator argument**) → *result-class* **special operator**

12.4 Defining forms

A ***defining form*** is a toplevel special form (see §12.3) that establishes a binding between *name* and an object which is the result of handling the *arguments* according to the semantics implied by *defining-form-name*; it is a violation if a defining form is not a toplevel form. For each namespace, defining forms can occur at most once for the same *name* and, in case of method definitions for the same parameter profile. A defining form is a compound form whose operator is a ***defining operator***. These are the defining operators:

`defclass`	`defdynamic`	`defglobal`	`defmethod`
`defconstant`	`defgeneric`	`defmacro`	`defun`

This document describes defining forms in the following format:

(*defining-form-name* <u>*name*</u> *argument**) → *<symbol>* **defining operator**

12.5 Macro forms

Macro forms are expanded during preparation for execution.

For information on how macros are processed, see §16.

12.6 The evaluation model

This section provides an operational model of the process of evaluation.

The process of evaluation has two steps: A valid ISLISP text is first prepared for execution, and then the prepared text is executed. Both the process of preparing the text for execution and the properties of a prepared text are implementation dependent, except that all macros have been expanded in the prepared text (see §16). The process of execution which follows is described in terms of fully macroexpanded forms.

A prepared form is executed as follows:

1. If the form is a *literal*, the result is the form itself.
2. If the form is an *identifier*, the result is the object denoted by the identifier in the variable namespace of the current lexical environment. An error shall be signaled if no binding has been established for the identifier in the variable namespace of current lexical environment (see §9.2) (error-id. *unbound-variable*).
3. If the form is a compound form, then one of the following cases must apply:
   (a) If the *operator* is a *special operator*, then the form is a special form and its *arguments* are evaluated according to the definition of the special operator. For example, `if` first evaluates its condition expression and, depending on the result obtained, it then evaluates the "then" form or the "else" form.
   (b) If the *operator* names a *defining form*, then the first argument is an identifier. The remaining *arguments* are handled according to the specification of the defining form and the resulting object is used to establish a binding between the identifier and that object in the appropriate namespace.
   (c) If the *operator* is a `lambda`-expression, then the *arguments* are evaluated. The order of evaluation of the *arguments* is sequentially from left to right. Then the function denoted by the `lambda`-expression is invoked with the evaluated arguments as actual parameters. The result is the value returned by the function, if it returns.
   Example:
   ```
   ((lambda (x) (+ x x)) 4)          ⇒  8
   ```
   (d) Otherwise, the compound form is a function application form. The operator position of the form is an identifier; it will be evaluated in the function namespace to produce a function to be called. An error shall be signaled if no binding has been established for the identifier in the function namespace of the current lexical environment (see §9.2) (error-id. *undefined-function*). The *arguments* are evaluated in order from left to right. Then the function is invoked with the evaluated arguments as actual parameters. The result is the value returned by the function, if it returns.
4. Otherwise, an error shall be signaled (error-id. *undefined-function*).

See §9.2 for descriptions of error situations that might occur during execution of the above cases.

12.7 Functions

A function can ***receive*** some objects as *arguments* upon activation. If a function returns, it ***returns*** an object as its ***value***. A function binding can be established in one of the following ways:

- by using function defining forms; *i.e.,* the `defun`, `defgeneric`, and `defclass` defining forms
- by using `labels` and `flet` special forms

(`functionp` *obj*) → *boolean* **function**

Returns `t` if *obj* is a (normal or generic) function; otherwise, returns `nil`. *obj* may be any ISLISP object.

Example:

```
(functionp (function car))          ⇒ t
```

Function bindings are entities established during execution of a prepared `labels` or `flet` forms or by a function-defining form. A function binding is an association between an identifier, *function-name*, and a function object that is denoted by *function-name*—if in operator position—or by (`function` *function-name*) elsewhere.

(`function` *function-name*) → <*function*> **special operator**
`#'`*function-name* → <*function*> **syntax**

This special form denotes a reference to the function named by *function-name*. This special form is used to refer to identifiers defined by function-defining forms, `labels`, or `flet` which are not in operator position.

(`function` *function-name*) can be written as `#'`*function-name*.

It returns the function object named by *function-name*.

An error shall be signaled if no binding has been established for the identifier in the function namespace of current lexical environment (see §9.2) (error-id. *undefined-function*). The consequences are undefined if the *function-name* names a macro or special form.

Example:

```
(funcall (function -) 3)            ⇒ -3
(apply #'- '(4 3))                  ⇒ 1
```

(`lambda` *lambda-list* *form**) → <*function*> **special operator**

Where:

lambda-list ::= (*identifier** [`&rest` *identifier*]) |
(*identifier** [`:rest` *identifier*])

and where no *identifier* may appear more than once in *lambda-list*.

Execution of the `lambda` special form creates a function object.

The scope of the identifiers of the *lambda-list* is the sequence of forms *form**, collectively referred to as the *body*.

When the prepared function is activated later (even if transported as object to some other activation) with some arguments, the body of the function is evaluated as if it was at the same textual position where the `lambda` special form is located, but in a context where the lambda variables are bound in the variable namespace with the values of the corresponding arguments. A `&rest` or `:rest` variable, if any, is bound to the list of the values of the remaining arguments. An error shall be signaled if the number of arguments received is incompatible with the specified *lambda-list* (error-id. *arity-error*).

Once the lambda variables have been bound, the body is executed. If the body is empty, `nil` is returned otherwise the result of the evaluation of the last form of body is returned if the body was not left by a *non-local exit* (see §14.7.1).

If the function receives a `&rest` or `:rest` parameter R, the list L_1 to which that parameter is bound has indefinite extent. L_1 is newly allocated unless the function was called with `apply` and R corresponds to the final argument, L_2, to that call to `apply` (or some subtail of L_2), in which case it is implementation defined whether L_1 shares structure with L_2.

Example:

```
((lambda (x y) (+ (* x x) (* y y))) 3 4)
  ⇒ 25
((lambda (x y &rest z) z) 3 4 5 6)
  ⇒  (5 6)
((lambda (x y :rest z) z) 3 4 5 6)
  ⇒  (5 6)
(funcall (lambda (x y) (- y (* x y))) 7 3)
  ⇒ -18
```

`(labels ((`*function-name* *lambda-list* *form**`)*)` *body-form**`)` → <*object*> **special operator**
`(flet ((`*function-name* *lambda-list* *form**`)*)` *body-form**`)` → <*object*> **special operator**

The `flet` and `labels` special forms allow the definition of new identifiers in the function namespace for function objects.

In a `labels` special form the scope of an identifier *function-name* is the whole `labels` special form (excluding nested scopes, if any); for the `flet` special form, the scope of an identifier is only the *body-form**. Within these scopes, each *function-name* is bound to a function object whose behavior is equivalent to `(lambda` *lambda-list form**`)`, where free identifier references are resolved as follows:

- For a `labels` form, such free references are resolved in the lexical environment that was active immediately outside the `labels` form augmented by the function bindings for the given *function-names* (*i.e.,* any reference to a function *function-name* refers to a binding created by the `labels`).

- For a `flet` form, free identifier references in the `lambda`-expression are resolved in the lexical environment that was active immediately outside the `flet` form (*i.e.,* any reference to a function *function-name* are not visible).

During activation, the prepared `labels` or `flet` establishes function bindings and then evaluates each *body-form* in the body sequentially; the value of the last one (or `nil` if there is none) is the value returned by the special form activation.

No *function-name* may appear more than once in the function bindings.

Example:

```
(labels  ((evenp (n)
            (if (= n 0)
                t
                (oddp (- n 1))))
          (oddp (n)
            (if (= n 0)
                nil
                (evenp (- n 1)))))
  (evenp 88))                        ⇒ t

(flet ((f (x) (+ x 3)))
  (flet ((f (x) (+ x (f x))))
    (f 7)))                          ⇒ 17
```

(apply *function obj* list*) → <*object*> **function**

Applies *function* to the arguments, *obj**, followed by the elements of *list*, if any. It returns the value returned by *function*.

An error shall be signaled if *function* is not a function (error-id. *domain-error*). Each *obj* may be any ISLISP object. An error shall be signaled if *list* is not a proper list (see §7) (error-id. *improper-argument-list*).

Example:

```
(apply (if (< 1 2) (function max) (function min))
       1 2 (list 3 4))               ⇒ 4

(defun compose (f g)
  (lambda (:rest args)
    (funcall f (apply g args))))     ⇒ compose

(funcall (compose (function sqrt) (function *)) 12 75)
  ⇒ 30
```

(`funcall` *function obj**) → <*object*> **function**

Activates the specified function *function* and returns the value that the function returns. The ith argument ($2 \leq i$) of `funcall` becomes the $(i-1)$th argument of the function. `funcall` could have been defined using `apply` as follows:

```
(defun funcall (function :rest arguments)
  (apply function arguments))
```

An error shall be signaled if *function* is not a function (error-id. *domain-error*). Each *obj* may be any ISLISP object.

Example:

```
(let ((x '(1 2 3)))
  (funcall (cond ((listp x) (function car))
                 (t (lambda (x) (cons x 1)))) x))
  ⇒ 1
```

12.8 Defining operators

Although the names defined by defining forms can be used throughout the current toplevel scope, the prepared toplevel forms in an ISLISP text unit are executed sequentially from left to right.

Two defining forms with the same identifier in the same namespace are not allowed in one toplevel scope.

(`defconstant` *name* *form*) → <*symbol*> **defining operator**

This form is used to define a named constant in the variable namespace of the current toplevel scope. The scope of *name* is the entire current toplevel scope except the body *form*.

Although *name* is globally constant, a variable binding for *name* can be locally established by a binding form.

The result of the evaluation of *form* is bound to the variable named by *name*. The binding and the object created as the result of evaluating the second argument are immutable. The symbol named *name* is returned.

Example:

```
(defconstant e 2.7182818284590451)  ⇒ e
e                                   ⇒ 2.7182818284590451
(defun f () e)                      ⇒ f
(f)                                 ⇒ 2.7182818284590451
```

(defglobal <u>*name*</u> *form*) → <*symbol*> **defining operator**

This form is used to define an identifier in the variable namespace of the current toplevel scope. The scope of *name* is the entire current toplevel scope except the body *form*.

form is evaluated to compute an initializing value for the variable named *name*. Therefore, `defglobal` is used only for defining variables and not for modifying them. The symbol named *name* is returned.

A lexical variable binding for *name* can still be locally established by a binding form; in that case, the local binding lexically shadows the outer binding of *name* defined by `defglobal`.

Example:

```
(defglobal today 'wednesday)        ⇒ today
today                               ⇒ wednesday
(defun what-is-today () today)      ⇒ what-is-today
(what-is-today)                     ⇒ wednesday
(let ((what-is-today 'thursday)) (what-is-today))
  ⇒ wednesday
(let ((today 'thursday)) (what-is-today))
  ⇒ wednesday
```

(defdynamic <u>*name*</u> *form*) → <*symbol*> **defining operator**

This form is used to define a dynamic variable identifier in the dynamic variable namespace. The scope of *name* is the entire current toplevel scope except the body *form*.

The symbol named *name* is returned.

Example:

```
(defdynamic *color* 'red)           ⇒ *color*
(dynamic *color*)                   ⇒ red
(defun what-color () (dynamic *color*))
  ⇒ what-color
(what-color)                        ⇒ red
(dynamic-let ((*color* 'green)) (what-color))
  ⇒ green
```

(defun <u>*function-name*</u> <u>*lambda-list*</u> <u>*form*</u>*) → <*symbol*> **defining operator**

The `defun`-form defines *function-name* as an identifier in the function namespace; *function-name* is bound to a function object equivalent to (lambda *lambda-list* *form**).

The scope of *function-name* is the whole current toplevel scope. Therefore, the definition of a function admits recursion, occurrences of *function-name* within the *form** refer to the function being defined. The binding between *function-name* and the function object is immutable.

`defun` returns the function name which is the symbol named *function-name*. The free identifiers in the body *form** (*i.e.,* those which are not contained in the lambda list) follow the rules of lexical scoping.

<u>`Example:`</u>

```
(defun caar (x) (car (car x)))      ⇒ caar
```

13 Predicates

13.1 Boolean values

The values `t` and `nil` are called ***booleans***. `t` denotes true, and `nil` is the only value denoting false. ***Predicates***, also called ***boolean functions***, are functions that return `t` when satisfied and `nil` otherwise.

Any object other than `nil` is treated as true (not just `t`). When objects are treated as true or `nil` this way they are called ***quasi-booleans***.

`t` is an identifier naming the symbol `t`, and `nil` is an identifier naming the symbol `nil` (which is also the empty list). `nil` is the unique instance of the `<null>` class.

Like boolean functions, the `and` and `or` special forms return truth values; however, these truth values are `nil` when the test is not satisfied and a non-`nil` value otherwise. The result of `and` and `or` are quasi-booleans.

`t` → <*symbol*>	**named constant**
`nil` → <*null*>	**named constant**

`t` is a named constant whose value is the symbol `t` itself. `nil` is a named constant whose value is the symbol `nil` itself.

13.2 Class predicates

The following functions are one-argument class membership predicates:

`basic-array*-p`	`floatp`	`integerp`	`stringp`
`basic-array-p`	`functionp`	`listp`	`symbolp`
`basic-vector-p`	`general-array*-p`	`null`	
`characterp`	`general-vector-p`	`numberp`	
`consp`	`generic-function-p`	`streamp`	

In addition, the function `instancep` is a two-argument predicate that tests membership in an arbitrary class.

13.3 Equality

(`eq` *obj*$_1$ *obj*$_2$) → *boolean* **function**
(`eql` *obj*$_1$ *obj*$_2$) → *boolean* **function**

`eq` and `eql` test whether *obj*$_1$ and *obj*$_2$ are same identical object. They return `t` if the objects are the same; otherwise, they return `nil`. Two objects are the same if there is no operation that could distinguish them (without modifying them), and if modifying one would modify the other the same way.

For `eq`, the consequences are implementation defined if both *obj*$_1$ and *obj*$_2$ are numbers or both are characters. For `eql` the meaning for numbers and characters is defined as follows:

- If *obj*$_1$ and *obj*$_2$ are numbers, `eql` tests whether they are direct instances of the same class and have the same value.

  If an implementation supports positive and negative zeros as *distinct* values, then `(eql 0.0 -0.0)` returns `nil`. When the syntax `-0.0` is read and it is interpreted as the value `0.0` then `(eql 0.0 -0.0)` returns `t`.

- If *obj*$_1$ and *obj*$_2$ are characters, `eql` tests whether they are the same character (see `char=`).

Example:

```
(eql () ())                    ⇒ t
(eq  () ())                    ⇒ t
(eql '() '())                  ⇒ t
(eq  '() '())                  ⇒ t
(eql 'a 'a)                    ⇒ t
(eq  'a 'a)                    ⇒ t
(eql 'a 'A)                    ⇒ t
(eq  'a 'A)                    ⇒ t
(eql 'a 'b)                    ⇒ nil
(eq  'a 'b)                    ⇒ nil
(eql 'f 'nil)                  ⇒ nil
(eq  'f 'nil)                  ⇒ nil
(eql 2 2)                      ⇒ t
(eq  2 2)                      ⇒ nil or t  (implementation-defined)
(eql 2 2.0)                    ⇒ nil
(eq  2 2.0)                    ⇒ nil
(eql 100000000 100000000)      ⇒ t
(eq  100000000 100000000)      ⇒ nil or t  (implementation-defined)
(eql 10.00000 10.0)            ⇒ t
(eq  10.00000 10.0)            ⇒ nil or t  (implementation-defined)
(eql (cons 1 2) (cons 1 2))    ⇒ nil
(eq  (cons 1 2) (cons 1 2))    ⇒ nil
(let ((x '(a))) (eql x x))     ⇒ t
```

```
(let ((x '(a))) (eq x x))          ⇒  t
(eql '(a) '(a))                    ⇒  nil or t   (implementation-defined)
(eq  '(a) '(a))                    ⇒  nil or t   (implementation-defined)
(let ((x '(b))
      (y '(a b)))
  (eql x (cdr y)))                 ⇒  nil or t   (implementation-defined)
(let ((x '(b))
      (y '(a b)))
  (eq x (cdr y)))                  ⇒  nil or t   (implementation-defined)
(eql '(b) (cdr '(a b)))            ⇒  nil or t   (implementation-defined)
(eq  '(b) (cdr '(a b)))            ⇒  nil or t   (implementation-defined)
(let ((p (lambda (x) x)))
  (eql p p))                       ⇒  t
(let ((p (lambda (x) x)))
  (eq p p))                        ⇒  t
(let ((x "a")) (eql x x))          ⇒  t
(let ((x "a")) (eq x x))           ⇒  t
(eql "a" "a")                      ⇒  nil or t   (implementation-defined)
(eq  "a" "a")                      ⇒  nil or t   (implementation-defined)
(let ((x "")) (eql x x))           ⇒  t
(let ((x "")) (eq x x))            ⇒  t
(eql "" "")                        ⇒  nil or t   (implementation-defined)
(eq  "" "")                        ⇒  nil or t   (implementation-defined)
(eql #\a #\A)                      ⇒  nil
(eq  #\a #\A)                      ⇒  nil
(eql #\a #\a)                      ⇒  t
(eq  #\a #\a)                      ⇒  nil or t   (implementation-defined)
(eql #\space #\Space)              ⇒ t
(eq  #\space #\Space)              ⇒ nil or t   (implementation-defined)
(eql #\space #\space)              ⇒ t
(eq  #\space #\space)              ⇒ nil or t   (implementation-defined)
```

`(equal` *obj*$_1$ *obj*$_2$`)` → *boolean* **function**

This function tests whether *obj*$_1$ and *obj*$_2$ are isomorphic—*i.e.,* whether *obj*$_1$ and *obj*$_2$ denote the same structure with equivalent values. `equal` returns `t` if the test was satisfied, and `nil` if not. Specifically:

If *obj*$_1$ and *obj*$_2$ are direct instances of the same class, `equal` returns `t` if they are `eql`. Otherwise (if they are direct instances of the same class but not `eql`), the result is `t` if one of the following cases applies:

(a) lists: either *obj*$_1$ and *obj*$_2$ are both the empty list (*i.e.,* `nil`), or

```
(and (equal (car obj1) (car obj2))
     (equal (cdr obj1) (cdr obj2)))
```
holds;

(b) basic arrays:

```
(equal (array-dimensions obj1)
       (array-dimensions obj2))
```

holds and for every valid reference (`aref` obj_1 ind_1 $\ldots ind_n$)

> (`equal` (`aref` obj_1 ind_1 $\ldots ind_n$)
>   (`aref` obj_2 ind_1 $\ldots ind_n$)) is satisfied.

Otherwise the value is `nil`.

obj_1 and obj_2 may be any ISLISP objects.

Example:

```
(equal 'a 'a)                        ⇒  t
(equal 2 2)                          ⇒  t
(equal 2 2.0)                        ⇒  nil
(equal '(a) '(a))                    ⇒  t
(equal '(a (b) c)
       '(a (b) c))                   ⇒  t
(equal (cons 1 2) (cons 1 2))        ⇒  t
(equal '(a) (list 'a))               ⇒  t
(equal "abc" "abc")                  ⇒  t
(equal (vector 'a) (vector 'a))      ⇒  t
(equal #(a b) #(a b))                ⇒  t
(equal #(a b) #(a c))                ⇒  nil
(equal "a" "A")                      ⇒  nil
```

13.4 Logical connectives

(`not` *obj*) → *boolean* **function**

This predicate is the logical "not" (or "¬"). It returns `t` if *obj* is `nil` and `nil` otherwise. *obj* may be any ISLISP object.

Example:

```
(not t)                              ⇒ nil
(not '())                            ⇒ t
(not 'nil)                           ⇒ t
(not nil)                            ⇒ t
(not 3)                              ⇒ nil
(not (list))                         ⇒ t
(not (list 3))                       ⇒ nil
```

(`and` *form**) → <*object*> **special operator** `and` is the sequential logical "and" (or "∧"). *forms* are evaluated from left to right until either one of them evaluates to `nil` or else none are left. If one of them evaluates to `nil`, then `nil` is returned from the `and`; otherwise, the value of the last evaluated form is returned. The form `and` is equivalent to the following:

`(and)`	≡	`'t`
`(and` *form*`)`	≡	*form*
`(and` $form_1$ $form_2$ … $form_n$`)`	≡	`(if` $form_1$ `(and` $form_2$ … $form_n$`) 'nil)`[2]

Example:

```
(and (= 2 2) (> 2 1))                ⇒  t
(and (= 2 2) (< 2 1))                ⇒ nil
(and (eql 'a 'a) (not (> 1 2)))      ⇒  t
(let ((x 'a)) (and x (setq x 'b)))   ⇒ b
(let ((x nil)) (and x (setq x 'b)))  ⇒ nil
(let ((time 10))
  (if (and (< time 24) (> time 12))
      (- time 12) time))             ⇒ 10
(let ((time 18))
  (if (and (< time 24) (> time 12))
      (- time 12) time))             ⇒ 6
```

`(or` *form**`)` → <*object*> **special operator**

`or` is the sequential logical "or" (or "∨"). *forms* are evaluated from left to right until either one of them evaluates to a non-`nil` value or else none are left. If one of them evaluates to a non-`nil` value, then this non-`nil` value is returned, otherwise `nil` is returned. The form `or` is equivalent to the following:

`(or)`	≡	`'nil`
`(or` *form*`)`	≡	*form*

`(or` $form_1$ $form_2$ … $form_n$`)`	≡	`((lambda (`*var*`)`
		 `(if` *var* *var* `(or` $form_2$ … $form_n$`)))` $form_1$`)`
		 where *var* does not occur in $form_2$ … $form_n$

Example:

```
(or (= 2 2) (> 2 1))                ⇒ t
(or (= 2 2) (< 2 1))                ⇒ t
(let ((x 'a)) (or x (setq x 'b)))   ⇒ a
(let ((x nil)) (or x (setq x 'b)))  ⇒ b
```

[2]For the definition of `if`, see §14.4 below.

14 Control structure

14.1 Constants

constant → <*object*> **syntax**

There are three kinds of constants: literals, quoted expressions, and named constants. Quoted expressions are described below.

The consequences are undefined if an attempt is made to alter the value of a constant.

The result of evaluating the literal constant *constant* is *constant* itself. Instances of the following classes are literal constants: `<basic-array>`, `<character>`, and `<number>`

Example:

```
#2A((a b c) (d e f))          ⇒ #2A((a b c) (d e f))
#\a                           ⇒ #\a
145932                        ⇒ 145932
"abc"                         ⇒ "abc"
#(a b c)                      ⇒ #(a b c)
```

(quote *obj*) → <*object*> **special operator**
'*obj* → <*object*> **syntax**

A quoted expression denotes a reference to an object. This notation is used to include any object in an ISLISP text.

The character ' (apostrophe or single quote) is syntax for quotation. That is, `(quote a)` ≡ `'a`.

The result of the evaluation of the `quote` special form is *obj*.

Example:

```
(quote a)                     ⇒ a
(quote #(a b c))              ⇒ #(a b c)
(quote (+ 1 2))               ⇒ (+ 1 2)
'()                           ⇒ nil
'a                            ⇒ a
'#(a b c)                     ⇒ #(a b c)
'(car l)                      ⇒ (car l)
'(+ 1 2)                      ⇒ (+ 1 2)
'(quote a)                    ⇒ (quote a)
''a                           ⇒ (quote a)
(car ''a)                     ⇒ quote
```

The consequences are undefined if an attempt is made to alter the value of a quoted expression.

14.2 Variables

Variable bindings are entities established during execution of the prepared variable-binding forms or by the activation of functions.

A ***variable*** is used to refer to an association between an *identifier* and an ISLISP object, and is denoted by that identifier. The association can be altered (by assignment) using the `setf` special form or `setq` special form.

The following are variable binding forms:

`defglobal` `let` `for` `let*`

---

var → <*object*> **syntax**

The value of *var* is the object associated with *var* in its variable binding.

Example:

```
(defglobal x 0)             ⇒ x
x                           ⇒ 0
(let ((x 1)) x)             ⇒ 1
x                           ⇒ 0
```

(`setq` *var* *form*) → <*object*> **special operator**

This form represents an assignment to the variable denoted by the identifier. In consequence, the identifier may designate a different object than before, the value of *form*.

The result of the evaluation of *form* is returned. This result is used to modify the variable binding denoted by the identifier *var* (if it is mutable). `setq` can be used only for modifying bindings, and not for establishing a variable. The `setq` special form must be contained in the scope of *var*, established by `defglobal`, `let`, `let*`, `for`, or a lambda expression.

Example:

```
(defglobal x 2)             ⇒ x
(+ x 1)                     ⇒ 3
(setq x 4)                  ⇒ 4
(+ x 1)                     ⇒ 5
(let ((x 1)) (setq x 2) x)  ⇒ 2
(+ x 1)                     ⇒ 5
```

(`setf` place form) → <object> **special operator**

This macro is used for generalized assignment.

`setf` takes a *place* and stores in this place the result of the evaluation of the form *form*. The *place* form is not evaluated as a whole entity, but subforms of *place* are evaluated sequentially from left to right to determine a place to be assigned a value. When *place* is denoted by an identifier, `setf` behaves exactly as `setq`. The returned value is the result of the evaluation of *form*. The valid places for the `setf` special form are as follows:

variables	*var*
dynamic bindings	(`dynamic` *var*)
the components of a basic-array	(`aref` *basic-array* z_1 ... z_n)
the components of a general array	(`garef` *general-array* z_1 ... z_n)
the components of a list	(`elt` *list* *z*)
the components of a vector	(`elt` *basic-vector* *z*)
the left component of a cons	(`car` *cons*)
the right component of a cons	(`cdr` *cons*)
a property of a symbol	(`property` *symbol* *property*)
a slot of an instance of a class	(*reader-function-name* *instance*)

A place can also be a macro form that expands (during preparation for execution) into a place or a function application form with operator *op* for which `setf` is defined or for which a generic function named (`setf` *op*) has been defined. In these last two cases, that function will receive as arguments the new value to be assigned followed by the objects that resulted from evaluating the arguments of the function application form.

Example:

```
(setf (car x) 2)                     ⇒  2
 In the cons x, the car now is 2.

(defmacro first (spot)
  `(car ,spot))                      ⇒ first
(setf (first x) 2)                   ⇒  2
 In the cons x, the car now is 2.
```

(`let` ((var form)*) body-form*) → <object> **special operator**

The `let` special form is used to define a scope for a group of identifiers for a sequence of forms *body-form** (collectively referred to as the *body*). The list of pairs (*var* *form*)* is called the `let` variable list. The scope of the identifier *var* is the *body*.

The *forms* are evaluated sequentially from left to right; then each variable denoted by the identifier *var* is initialized to the corresponding value. Using these bindings along with the already existing bindings of visible identifiers the *body-forms* are evaluated. The returned value of `let` is the result of the evaluation of the last *body-form* of its body (or `nil` if there is none).

No *var* may appear more than once in `let` variable list.

Note: Although this form is a special form, one can think of it as a macro whose rewriting rules are as follows:

```
(let () body-form*)         ≡    (progn body-form*)[3]
(let ((var1 form1)          ≡    ((lambda (var1 var2 ... varn)
      (var2 form2)                    body-form*
         ...                      ) form1 form2 ... formn)[4]
      (varn formn))
   body-form*)
```

Example:

```
(let ((x 2) (y 3))
  (* x y))                    ⇒  6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))                 ⇒  35

(let ((x 1) (y 2))
  (let ((x y) (y x))
    (list x y)))              ⇒ (2 1)
```

`(let* ((`<u>*var*</u> *form*`)*)` *body-form*`*)` → <*object*> **special operator**

The `let*` form is used to define a scope for a group of identifiers for a sequence of forms *body-form** (collectively referred to as the *body*). The first subform (the `let*` variable list) is a list of pairs (*var form*). The scope of an identifier *var* is the *body* along with all *form* forms following the pair (*var form*) in the `let*` variable list.

For each pair (*var form*) the following is done: *form* is evaluated in the context of the bindings in effect at that point in the evaluation. The result of the evaluation is bound to its associated variable named by the identifier *var*. These variable bindings enlarge the set of current valid identifiers perhaps shadowing previous variable bindings (in case some *var* was defined outside), and in this enlarged or modified environment the *body-forms* are executed. The returned value of `let*` is the result of the evaluation of the last form of its body (or `nil` if there is none).

Note: Although this form is a special form, one can think of it as a macro whose rewriting rules are as follows:

```
(let* () body-form*)  ≡  (progn body-form*)
(let* ((var1 form1)   ≡  (let ((var1 form1))
```

[3]For the definition of `progn` see §14.5 below.
[4]For the definition of `lambda` see §12.7.

```
      (var2 form2)          (let ((var2 form2))
          ...                 ...
      (varn formn))             (let ((varn formn))
  body-form*)                       body-form*)...))
```

Example:

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))                      ⇒  70

(let ((x 1) (y 2))
  (let* ((x y) (y x))
    (list x y)))                   ⇒ (2 2)
```

14.3 Dynamic variables

A dynamic variable is an association between an identifier *var* and an ISLisp object in the dynamic variable namespace. Dynamic variables implement a form of dynamic binding.

Dynamic variables are defined globally by `defdynamic` and are established during the execution of a prepared `dynamic-let`.

Dynamic variable bindings defined by `defdynamic` persist indefinitely whereas those established by `dynamic-let` are disestablished upon end of execution of this special form.

The value of a dynamic variable can be accessed by `(dynamic` *var*`)`.

`(dynamic` <u>*var*</u>`)` → <*object*> **special operator**

This special form denotes a reference to the identifier denoting a dynamic variable. This special form is not allowed in the scope of a definition of *var* which is not done by `defdynamic` or `dynamic-let`.

During activation, the current dynamic binding of the variable *var* is returned that was established most recently and is still in effect. An error shall be signaled if such a binding does not exist (error-id. *unbound-variable*).

`(setf (dynamic` <u>*var*</u>`)` *form*`)` → <*object*> **special form**
`(set-dynamic` *form* <u>*var*</u>`)` → <*object*> **special operator**

This special form denotes an assignment to a dynamic variable. This form can appear anywhere that `(dynamic` *var*`)` can appear.

form is evaluated and the result of the evaluation is used to change the dynamic binding of *var*.

An error shall be signaled if *var* has no dynamic value (error-id. *unbound-variable*). `setf` of `dynamic` can be used only for modifying bindings, and not for establishing them.

`(dynamic-let ((`*var* *form*`)*)` *body-form*`*)` → <*object*> **special operator**

The `dynamic-let` special form is used to establish dynamic variable bindings. The first subform (the `dynamic-let` variable list) is a list of pairs `(`*var* *form*`)`. The scope of an identifier *var* defined by `dynamic-let` is the current toplevel scope. The extent of the bindings of each *var* is the extent of the body of the `dynamic-let`. The `dynamic-let` special form establishes dynamic variables for all *vars*.

References to a dynamic variable named by *var* must be made through the `dynamic` special form.

All the initializing forms are evaluated sequentially from left to right, and then the values are associated with the corresponding *vars*. Using these additional dynamic bindings and the already existing bindings of visible identifiers, the forms *body-form** are evaluated in sequential order. The returned value of `dynamic-let` is that of the last *body-form* of the body (or `nil` if there is none). The bindings are undone when control leaves the prepared `dynamic-let` special form.

Example:

```
(defun foo (x)
  (dynamic-let ((y x))
    (bar 1)))                       ⇒ foo

(defun bar (x)
  (+ x (dynamic y)))                ⇒ bar

(foo 2)                             ⇒ 3
```

14.4 Conditional expressions

`(if` *test-form* *then-form* [*else-form*]`)` → <*object*> **special operator**

The *test-form* is evaluated. If its result is anything non-`nil`, the *then-form* is evaluated and its value is returned; otherwise (if the *test-form* returned `nil`), the *else-form* is evaluated and its value is returned.

If no *else-form* is provided, it defaults to `nil`.

Example:

```
(if (> 3 2) 'yes 'no)               ⇒ yes
(if (> 2 3) 'yes 'no)               ⇒ no
(if (> 2 3) 'yes)                   ⇒ nil
```

```
(if (> 3 2) (- 3 2) (+ 3 2))        ⇒ 1

(let ((x 7))
  (if (< x 0) x (- x)))             ⇒ -7
```

(cond (*test form**)*) → <*object*> **special operator**

Executing the prepared `cond`, the clauses (*test form**) are scanned sequentially and in each case the *test* is evaluated; when a *test* delivers a non-`nil` value the scanning process stops and all *forms* associated with the corresponding clause are sequentially evaluated and the value of the last one is returned. If no *test* is true, then `nil` is returned. If no *form* exists for the successful *test* then the value of this *test* is returned.

`cond` obeys the following equivalences:

(cond)	≡	nil
(cond ($test_1$) ($test_2$ $form_2^*$) ...)	≡	(or $test_1$ (cond ($test_2$ $form_2^*$) ...))
(cond ($test_1$ $form_1^+$) ($test_2$ $form_2^*$) ...)	≡	(if $test_1$ (progn $form_1^+$) (cond ($test_2$ $form_2^*$) ...))

Example:

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))              ⇒  greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less))              ⇒  nil

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (t       'equal))             ⇒  equal
```

(case *keyform* ((*key**) *form**)* [(t *form**)]) → <*object*> **special operator**
(case-using *predform* *keyform* ((*key**) *form**)* [(t *form**)])
→ <*object*> **special operator**

The `case` and `case-using` special forms, called ***case forms***, provide a mechanism to execute a matching clause from a series of clauses based on the value of a dispatching form *keyform*.

The clause to be executed is identified by a set of keys. A *key* can be any object. If the keylist of the last clause is `t` the associated clause is executed if no key matches the *keyform*.

keyform is a form to be computed at the beginning of execution of the case form. If the result of evaluating *keyform* is equivalent to a *key*, then the forms, if any, in the corresponding clause are evaluated sequentially and the value of the last one is returned as value of the whole case form. `case` determines match equivalence by using `eql`; `case-using` match determines equivalence by using the result of evaluating *predform*. *predform* must be a boolean or quasi-boolean function that accepts two arguments, the value returned by *keyform* and *key*. If no *form* exists for a matching *key*, the case form evaluates to `nil`. If the value of *keyform* is different from every *key*, and there is a default clause, its forms, if any, are evaluated sequentially, and the value of the last one is the result of the case form.

The same *key* (as determined by the match predicate) may occur only once in a case form.

Example:

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((4 6 8 9) 'composite))          ⇒  composite

(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                        ⇒  nil

(case (car '(c d))
  ((a e i o u) 'vowel)
  ((y) 'semivowel)
  (t 'consonant))                  ⇒  consonant

 (let ((char #\u))
   (case char
     ((#\a #\e #\o #\u #\i) 'vowels)
     (t 'consonants)))             ⇒ vowels

(case-using #'= (+ 1.0 1.0)
  ((1) 'one)
  ((2) 'two)
  (t 'more))                       ⇒ two

(case-using #'string= "bar"
  (("foo") 1)
  (("bar") 2))                     ⇒ 2
```

14.5 Sequencing forms

`(progn` *form**`)` → <*object*> **special operator**

This special form allows a series of forms to be evaluated, where normally only one could be used.

The result of evaluation of the last form of *form** is returned. All the *forms* are evaluated from left to right. The values of all the *forms* but the last are discarded, so they are executed only for their side-effects. `progn` without *forms* returns `nil`.

Example:

```
(defglobal x 0)                     ⇒ x

(progn
  (setq x 5)
  (+ x 1))                          ⇒  6

(progn
  (format (standard-output) "4 plus 1 equals ")
  (format (standard-output) "~D" (+ 4 1)))
  ⇒  nil

  prints  4 plus 1 equals 5
```

14.6 Iteration

(`while` *test-form body-form**) → <*null*> **special operator**

Iterates while the *test-form* returns a true value. Specifically:

1. *test-form* is evaluated, producing a value V_t.
2. If V_t is `nil`, then the `while` form immediately returns `nil`.
3. Otherwise, if V_t is non-`nil`, the forms *body-form** are evaluated sequentially (from left to right).
4. Upon successful completion of the *body-forms**, the while form begins again with step 1.

Example:

```
(let ((x '()) (i 5))
  (while (> i 0) (setq x (cons i x)) (setq i (- i 1)))
  x)                                ⇒ (1 2 3 4 5)
```

(`for` (*iteration-spec**) (*end-test result**) *form**) → <*object*> **special operator**

Where:

iteration-spec ::= (<u>*var*</u> *init* [*step*])

`for` repeatedly executes a sequence of forms *form**, called its *body*. It specifies a set of identifiers naming variables that will be local to the `for` form, their initialization, and their update for each iteration. When a termination condition is met, the iteration exits with a specified result value.

The scope of an identifier *var* is the *body*, the *steps*, the *end-test*, and the *result**. A *step* might be omitted, in which case the effect is the same as if `(`<u>*var*</u> *init* *var*`)` had been written instead of `(`<u>*var*</u> *init*`)`. It is a violation if more than one *iteration-spec* names the same *var* in the same `for` form.

The `for` special form is executed as follows: The *init* forms are evaluated sequentially from left to right. Then each value is used as the initial value of the variable denoted by the corresponding identifier *var*, and the iteration phase begins.

Each iteration begins by evaluating *end-test*. If the result is `nil`, the forms in the *body* are evaluated sequentially (for side-effects). Afterwards, the *step*-forms are evaluated sequentially order from left to right. Then their values are assigned to the corresponding variables and the next iteration begins. If *end-test* returns a non-`nil` value, then the *result** are evaluated sequentially and the value of the last one is returned as value of the whole `for` macro. If no *result* is present, then the value of the `for` macro is `nil`.

<u>`Example:`</u>

```
(for ((vec (vector 0 0 0 0 0))
      (i 0 (+ i 1)))
     ((= i 5) vec)
  (setf (elt vec i) i))            ⇒  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (for ((x x (cdr x))
        (sum 0 (+ sum (car x))))
       ((null x) sum)))            ⇒  25
```

14.7 Non-local exits

14.7.1 Establishing and invoking non-local exits

ISLisp defines three ways in which to perform non-local exits:

Destination Kind	Established by	Invoked by	Operation Performed
block name	`block`	`return-from`	lexical exit
tagbody tag	`tagbody`	`go`	lexical transfer of control
catch tag	`catch`	`throw`	dynamic exit

A ***non-local exit***, is an operation that forces transfer of control and possibly data from an invoking special form to a previously established point in a program, called the ***destination*** of the exit.

A ***lexical exit*** is a non-local exit from a `return-from` form to a `block` form which contains it both lexically and dynamically, forcing the `block` to return an object specified in the `return-from` form.

A ***dynamic exit*** is a non-local exit from a `throw` form to a `catch` form which contains it dynamically (but not necessarily lexically), forcing the `catch` to return an object specified in the `throw` form.

A ***lexical transfer of control*** is a non-local exit from a `go` form to a tagged point in a `tagbody` form which contains it both lexically and dynamically.

When a *non-local exit* is initiated, any potential destination that was established more recently than the destination to which control is being transferred is immediately considered invalid.

(`block` *name* *form**) → <*object*> **special operator**
(`return-from` *name* *result-form*) *transfers control and data* **special operator**

The `block` special form executes each *form* sequentially from left to right. If the last *form* exits normally, whatever it returns is returned by the `block` form.

The *name* in a `block` form is not evaluated; it must be an identifier. The scope of *name* is the body *form**—only a `return-from` textually contained in some *form* can exit the block. The extent of *name* is dynamic.

If a `return-from` is executed, the *result-form* is evaluated. If this evaluation returns normally, the value it returns is immediately returned from the innermost lexically enclosing `block` form with the same *name*.

`return-from` is used to return from a `block`. *name* is not evaluated and must be an identifier. A `block` special form must lexically enclose the occurrence of `return-from`; the value produced by *result-form* is immediately returned from the block. The `return-from` form never returns and does not have a value.

An error shall be signaled if an attempt is made to exit a `block` after it has been exited (error-id. *control-error*); It is a violation if *name* is not an identifier. It is a violation if a block with a corresponding name does not exist. See §14.7.2 for other errors.

Example:

```
(block x
  (+ 10 (return-from x 6) 22)) ;;; Bad programming style
  ⇒ 6

(defun f1 ()
  (block b
    (let ((f (lambda () (return-from b 'exit))))
      ... ; big computation
      (f2 f))))                     ⇒ f1

(defun f2 (g)
  ... ; big computation
  (funcall g))                      ⇒ f2
```

```
(f1)                                ⇒ exit

(block sum-block
  (for ((x '(1 a 2 3) (cdr x))
        (sum 0 (+ sum (car x))))
       ((null x) sum)
    (cond ((not (numberp (car x))) (return-from sum-block 0)))))
  ⇒ 0

(defun bar (x y)
  (let ((foo #'car))
    (let ((result
           (block bl
             (setq foo (lambda () (return-from bl 'first-exit)))
             (if x (return-from bl 'second-exit) 'third-exit))))
      (if y (funcall foo) nil)
      result)))                     ⇒ bar

(bar t nil)                         ⇒ second-exit
(bar nil nil)                       ⇒ third-exit
(bar nil t)      an error shall be signaled
(bar t t)        an error shall be signaled
```

(catch *tag-form* *form**) → <*object*> **special operator**
(throw *tag-form* *result-form*) *transfers control and data* **special operator**

The special forms catch and throw provide a facility for programming of structured non-local dynamic exits. A catch form and a throw form are said to correspond if the *tag-form* of the catch and the *tag-form* of the throw evaluate to the same object, a ***catch tag***. A catch tag may be any object other than a number or a character; the comparison of *catch tags* uses eq.

The catch special form first evaluates the *tag-form* to produce a *catch tag*, and then executes each *form* sequentially from left to right. If execution of the *forms* finishes normally, whatever is returned by the last *form* is returned by the catch form.

Prior to execution of the *forms* of a catch form C_0, an association between the *catch tag* T_0 and the executing form C_0 is dynamically established, upon exit from C_0, the association is disestablished. If there was an outer association for the same *catch tag* T_0, it is hidden during the execution of C_0's *forms*; only the most recently established (*i.e.,* innermost) association for T_0 is ever visible.

If a throw special form is executed, it evaluates the *tag-form* producing a *catch tag* T_1, and then evaluates the *result-form* producing a result R_1. If there is a corresponding association between T_1 and some catch form C_1 that is executing, R_1 is immediately returned as the value of C_1. The throw form can be anywhere in the entire current toplevel scope; it need not be lexically contained within C_1.

An error shall be signaled if there is no outstanding catcher for a T_1 (error-id. *control-error*). See §14.7.2 for other errors.

Example:

```
(defun foo (x)
       (catch 'block-sum (bar x)))  ⇒ foo

(defun bar (x)
  (for ((l x (cdr l))
         (sum 0 (+ sum (car l))))
        ((null l) sum)
    (cond ((not (numberp (car l))) (throw 'block-sum 0)))))
           ⇒ bar

(foo '(1 2 3 4))                    ⇒ 10
(foo '(1 2 a 4))                    ⇒ 0
```

(tagbody {*tagbody-tag* | *form*}*) → <*object*> **special operator**
(go *tagbody-tag*) *transfers control* **special operator**

`tagbody` executes the *forms* sequentially from left to right, discarding their values. If the execution of the last *form* completes normally, `nil` is returned by the `tagbody` special form.

The series of *tagbody-tags* and *forms* is collectively referred to as the body of a `tagbody` form. An identifier *tagbody-tag* that appears at toplevel of the body denotes a ***tagbody tag*** that can be used with `go` to transfer control to that point in the body. Any compound form that appears is taken as a *form*. Literals are not permitted at the toplevel of a tagbody. No *tagbody-tag* may appear more than once in the tags in the body.

The namespace used for tagbody tags is distinct from that used for block tags.

At any point lexically contained in the tagbody a form (`go` *tag*) can be used to transfer control to a tag *tag* that appears among the *tagbody-tags*, except where a *tag* is shadowed according to the *lexical principle* (see §11.1).

A *tagbody-tag* established by `tagbody` has lexical scope, but the point in the program to which it refers has dynamic extent. Once `tagbody` has been exited, it is no longer valid to use `go` to transfer to any tag in its body.

The determination of which elements of the body are *tagbody-tags* and which are *forms* is made prior to any macro expansion of that element. If *form* is a macro form and its macro expansion is a symbol or literal, that atom is treated as a *form*, not as a *tagbody-tag*.

It is a violation if a tagbody tag is other than an identifier. See §14.7.2 for other errors.

Note: As a stylistic matter, programmers are not encouraged to use `tagbody` and `go` in everyday programming. The primary uses for which these forms are intended are for implementing other control abstractions (using macros), and for the occassional real-world situation that uses unstructured imperative transfer of control (such as a finite state machine).

Example:

```
(defmacro with-retry (:rest forms)
  (let ((tag (gensym)))
    `(block ,tag
       (tagbody
         ,tag
         (return-from ,tag
           (flet ((retry () (go ,tag)))
             ,@forms))))))
 ⇒ with-retry

(let ((i -5))
  (with-retry
    ;; if-error is a hypothetical error correction function
    ;; not supplied by ISLISP.
    (if-error (sqrt (setq i (+ i 4)))
              (retry))))
 ⇒ 1.7320508075688772
```

14.7.2 Assuring data consistency during non-local exits

`(unwind-protect` *form* *cleanup-form**`)` → <*object*> **special operator**

`unwind-protect` first evaluates *form*. Evaluation of the *cleanup-forms* always occurs, regardless of whether the exit is normal or non-local.

If the *form* exits normally yielding a value R, then if all of the *cleanup-forms* exit normally the value R is returned by the `unwind-protect` form.

If a *non-local exit* from *form* occurs, then the *cleanup-forms* are executed as part of that exit, and then if all of the *cleanup-forms* exit normally the original non-local exit continues.

The *cleanup-forms* are evaluated from left to right, discarding the resulting values. If execution of the *cleanup-forms* finishes normally, exit from the `unwind-protect` form proceeds as described above. It is permissible for a *cleanup-form* to contain a non-local exit from the `unwind-protect` form, subject to the following constraint:

An error shall be signaled if during execution of the *cleanup-forms* of an `unwind-protect` form, a non-local exit is executed to a destination which has been marked as invalid due to some other non-local exit that is already in progress (see §14.7.1) (error-id. *control-error*).

Note: Because ISLISP does not specify an interactive debugger, it is unspecified whether or how error recovery can occur interactively if programmatic handling fails. The intent is that if the ISLISP processor does not terminate abnormally, normal mechanisms for non-local exit (`return-from`, `throw`, or `go`) would be used as necessary and would respect these *cleanup-forms*.

Example:

```
(defun foo (x)
```

```
  (catch 'duplicates
      (unwind-protect (bar x)
        (for ((l x (cdr l)))
             ((null l) 'unused)
          (remove-property (car l) 'label)))))
 ⇒ foo

(defun bar (l)
       (cond ((and (symbolp l) (property l 'label))
              (throw 'duplicates 'found))
             ((symbolp l) (setf (property l 'label) t))
             ((bar (car l)) (bar (cdr l)))
             (t nil)))
 ⇒ bar

(foo '(a b c))                    ⇒ t
(property 'a 'label)              ⇒ nil
(foo '(a b a c))                  ⇒ found
(property 'a 'label)              ⇒ nil

(defun test ()
  (catch 'outer (test2)))         ⇒ test

(defun test2 ()
  (block inner
    (test3 (lambda ()
             (return-from inner 7)))))
 ⇒ test2

(defun test3 (fun)
  (unwind-protect (test4) (funcall fun)))
 ⇒ test3

(defun test4 ()
  (throw 'outer 6))               ⇒ test4

(test)                            ⇒ an error shall be signaled
```

In the `test` example, the `throw` executed in `test4` has as destination the catcher established in `test`. The `unwind-protect` in `test3` intercepts the transfer of control and attempts to execute a `return-from` from the `block` in `test2`. Because this `block` is established within the dynamic extent of the destination catcher, an error is signaled.

15 Objects

15.1 Defining classes

The `defclass` defining form is used to define a new named class.

The definition of a class includes the following:

- The name of the new class.
- 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. A class definition must not contain two slot specifiers with the same name.
- A set of ***class options***. Each class option pertains to the class as a whole.

The slot options and class options of the `defclass` defining 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 retrieving or storing slot values and inquiring whether a value is bound to the slot.
- Indicating that the metaclass of that class is to be other than the default.

`(defclass` *class-name* `(`*sc-name**`)` `(`*slot-spec**`)` *class-opt**`)` → <*symbol*> **defining operator**

Where:

class-name	::=	*identifier*
sc-name	::=	*identifier*
slot-spec	::=	*slot-name* | `(`*slot-name* *slot-opt**`)`
slot-name	::=	*identifier*
slot-opt	::=	`:reader` *reader-function-name* |
		`:writer` *writer-function-name* |
		`:accessor` *reader-function-name* |
		`:boundp` *boundp-function-name* |
		`:initform` *form* |
		`:initarg` *initarg-name*
initarg-name	::=	*identifier*
reader-function-name	::=	*identifier*
writer-function-name	::=	*identifier*
boundp-function-name	::=	*identifier*
class-opt	::=	`(:metaclass` *class-name*`)` |
		`(:abstractp` *abstract-flag*`)`
abstract-flag	::=	`t` | `nil`

The `defclass` defining form returns the symbol named *class-name* as its result.

The *class-name* argument is an identifier which becomes the name of the new class. The defining point of the *class-name* is the end of the `defclass` defining form.

Each superclass name argument *sc-name* is an identifier that specifies a direct superclass of the new class. The new class will inherit slots and their `:reader` or `:writer` or `:accessor` methods from each of its superclasses. See §15.1.3 for a definition of how slots are inherited, and §15.2.3 for a definition of how methods are inherited. No *sc-name* may appear more than once in super class names. It is a violation if the superclasses of any two direct superclasses *sc-name* have superclasses other than `<standard-object>` and `<object>` in common unless a metaclass other than `<standard-class>` is specified.

Each *slot-spec* argument is the name of the slot or a list consisting of the slot name followed by zero or more slot options. The *slot-name* argument is an identifier that is syntactically valid for use as an ISLISP variable name. No slot names may appear more than once in *slot-spec*

The following slot options are available:

- The `:reader` slot option specifies that an unqualified method with the parameter profile `((`*x class-name*`))` is to be defined on the generic function named *reader-function-name* to retrieve the value of the given slot. The `:reader` slot option may be specified more than once for a given slot.
- The `:writer` slot option specifies that an unqualified method with the parameter profile `((`*y* `<object>) (`*x class-name*`))` is to be defined on the generic function named *writer-function-name* to store the value into the slot. The *writer-function-name* argument is an identifier. The `:writer` slot option may be specified more than once for a given slot.
- The `:accessor` slot option specifies that an unqualified method is to be defined on the generic function named *reader-function-name* to retrieve the value of the given slot. Furthermore, there is a generic function such that `(setf (`*reader-function-name x*`)` *y*`)` is equivalent to calling this generic function with first argument *y* and second argument *x*. This generic function is extended by a method with the parameter profile `((`*y* `<object>) (`*x class-name*`))`. The *reader-function-name* argument is an identifier. The `:accessor` slot option may be specified more than once for a given slot.
- The `:boundp` slot option specifies that an unqualified method with the parameter profile `((`*x class-name*`))` is to be defined on the generic function named *boundp-function-name* to test whether the given slot has been given a value. The `:boundp` slot option may be specified more than once for a given slot.
- The `:initform` slot option is used to provide a default initial value form to be used in the initialization of the slot. The `:initform` slot option may be specified once at most for a given slot. This form is evaluated every time it is used to initialize the slot. The lexical scope of the identifiers used in the initialization of the slot is the lexical scope of those identifiers in the `defclass` form. Note that the lexical scope refers both to variable and to function identifiers. In contrast, the current dynamic bindings used are those existing during activation of `create`. For more information, see §15.4.1.
- The `:initarg` slot option declares an initialization argument named *initarg-name* and specifies that this initialization argument initializes the given slot. If the initialization argument and associated value are supplied in the call to `initialize-object`, the value will be stored into the given slot and the slot's `:initform` slot option, if any, is not evaluated. If none of the initialization arguments specified for a given slot has a value, the slot is initialized according to the `:initform` option, if specified. The consequences are undefined if more than one initialization argument for the same slot is supplied. For more information, see §15.4.1.

The generic functions, to which the methods created by the `:reader`, `:writer`, and `:accessor` slot options belong are called ***slot accessors***.

No implementation is permitted to extend the syntax of `defclass` to allow `(`*slot-name form*`)` as an abbreviation for `(`*slot-name* `:initform` *form*`)`.

Each class option is an option that refers to the class as a whole. The following class options are available:

- The `:metaclass` class option is used to specify that instances of the class being defined are to have a different metaclass than the default provided by the system, that is, different from the class `<standard-class>`. The *class-name* argument is the name of the desired metaclass. The `:metaclass` class option may be specified once at most. It is a violation if `<built-in-class>` is specified as the metaclass.
- The `:abstractp` class option is used to specify that the class is an abstract class. If this option is supplied and *abstract-flag* is `t`, `create` will signal an error if an attempt is made to create an instance of this class. If the option is unsupplied, or if *abstract-flag* is `nil`, the class is not an abstract class. It is a violation if the *abstract-flag* is supplied but is neither `t` nor `nil`.

The following rules of `defclass` hold for standard classes:

- The `defclass` defining form must be in the scope of any superclass identifier it refers to.
- All the superclasses of a class must be defined before an instance of the class can be made.
- Any reference to *class-name* as a parameter specializer in a `defmethod` form must be in the scope of *class-name*. That is, a `defmethod` form that names a class must textually follow the `defclass` form that defines that class.

An ISLISP processor may be extended to cover situations where these rules are not obeyed. These extensions shall be implementation defined.

Some slot options are inherited by a class from its superclasses, and some can be shadowed or altered by providing a local slot description. No class options are inherited. For a detailed description of how slots and slot options are inherited, see the section §15.1.3.

If no slot accessors are specified for a slot, the slot cannot be accessed.

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

15.1.1 Determining the class precedence list

The `defclass` defining 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.

The 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.

Let $C_1, \ldots, C_n$ be the direct superclasses of C in the order defined in the `defclass` defining form for C. Let $P_1, \ldots, P_n$ be the class precedence lists for $C_1, \ldots, C_n$, respectively. Define $P \cdot Q$ on class precedence lists P and Q to be the two lists appended. Then the class precedence list for C is $C \cdot P_1 \cdot \ldots \cdot P_n$ with duplicate classes removed by repeated application of the following rule: If a class appears twice in the resulting class precedence list, the leftmost occurrence is removed.

It is a violation if an attempt is made to define an instance of `<standard-class>` whose direct superclasses have class precedence lists with classes other than `<standard-object>` and `<object>` in common.

15.1.2 Accessing slots

Slots can be accessed by use of the slot accessors created or modified by the `defclass` defining form.

The `defclass` defining form provides syntax for generating methods to retrieve and store slot values. If a ***reader*** is requested, a method is automatically generated for retrieving the value of the slot, but no method for storing a value into it is generated. If a ***writer*** is requested, a method is automatically generated for storing a value into the slot, but no method for retrieving its value is generated. If an ***accessor*** is requested, a method for retrieving the value of the slot and a method for storing a value into the slot are automatically generated.

When a reader or writer is specified for a slot, the name of the generic function to which the generated method belongs is directly specified. If the name specified for the writer option is the identifier *name*, the name of the generic function for storing a value into the slot is the identifier *name*, and the generic function takes two arguments: the new value and the instance, in that order. If the name specified for the accessor option is the identifier *name*, the name of the generic function for retrieving the slot value is the identifier *name*, and storing a value into the slot can be done by using the syntax `(setf (`*name* *instance*`)` *new-value*`)`.

A generic function created or modified by supplying reader, writer, or accessor slot options is a direct instance of `<standard-generic-function>`.

15.1.3 Inheritance of slots and slot options

The set of the names of all slots accessible in an instance of a class C is the union of the sets of names of slots defined by C and its superclasses. The ***structure*** of an instance is the set of names of slots in that instance.

In the simplest case, only one class among C and its superclasses defines a slot with a given slot name. If a slot is defined by a superclass of C, the slot is said to be ***inherited***. The characteristics of the slot are determined by the slot specifier of the defining class.

In general, more than one class among C and its superclasses can define a slot with a given name. In such cases, only one slot with the given name is accessible in an instance of C, and the characteristics of that slot are a combination of the several slot specifiers, computed as follows:

- All the slot specifiers for a given slot name are ordered from most specific to least specific, according to the order in C's class precedence list. All references to the specificity of slot specifiers immediately below refer to this ordering.
- The default initial value form for a slot is the value of the `:initform` slot option in the most specific slot specifier that contains one. If no slot specifier contains an `:initform` slot option, the slot has no default initial value form.

The `:reader`, `:writer`, and `:accessor` slot options create methods rather than define the characteristics of a slot. Reader and writer methods are inherited in the sense described in the section §15.2.3.

15.2 Generic functions

A ***generic function*** is a function whose application behavior depends on the classes 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 of its arguments.

A generic function can be used in the same ways that an ordinary function can be used.

A ***method*** consists of a method function, a lambda list, 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 formal parameter of each method has an associated parameter specializer, and the method is invoked only on arguments that satisfy its parameter specializers.

The method combination facility controls the selection of methods, the order in which they are activated, and the value that is returned by the generic function. ISLISP provides a default method combination type but does not provide a facility for declaring new types of method combination.

Like an ordinary ISLISP function, a generic function takes arguments, performs a series of operations, and returns a value. An ordinary function has a single body of code that is always executed when the function is called. A generic function has a set of bodies of code of which a non-empty subset is selected for execution. The selected bodies of code and the manner of their combination are determined by the classes of the arguments to the generic function and by its method combination type.

`(generic-function-p` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is a generic function; otherwise, returns `nil`. *obj* may be any ISLISP object.

15.2.1 Defining generic functions

Some forms specify the options of a generic function, such as the type of method combination it uses. These forms will be referred to as "forms that specify generic function options." These forms are the `defgeneric` defining forms.

Some forms define methods for a generic function. These forms will be referred to as "method-defining forms." These forms are the `defmethod` and `defclass` defining forms.

During preparation for execution, a `defmethod` form must be preceded by the `defgeneric` form for the generic function to be specialized. (Methods implicitly defined by `defclass` due to `:reader`, `:writer`, or `:accessor` options do not need a preceding `defgeneric`.)

(`defgeneric` *func-spec* *lambda-list* {*option* | *method-desc*}*) → <*symbol*> **defining operator**

Where:

func-spec	::=	*identifier* | `(setf` *identifier*`)`
lambda-list	::=	`(`*var** [`&rest` *var*]`)` |
		`(`*var** [`:rest` *var*]`)`
option	::=	`(:method-combination` {*identifier* | *keyword*}`)` |
		`(:generic-function-class` *class-name*`)`
method-desc	::=	`(:method` *method-qualifier** *parameter-profile* *form**`)`
method-qualifier	::=	*identifier* | *keyword*
parameter-profile	::=	`(`{*var* | `(`*var* *parameter-specializer-name*`)`}* [{`&rest` | `:rest`}*var*]`)`
parameter-specializer-name	::=	*class-name*
class-name	::=	*identifier*
var	::=	*identifier*

The `defgeneric` defining form is used to define a generic function and to specify options and declarations that pertain to a generic function as a whole.

It returns the generic function name *func-spec*.

The scope of the generic function identifier *func-spec* is the entire current toplevel scope.

The *lambda-list* argument is an ordinary function lambda-list.

The following options are provided. A given option may occur only once.

- The `:generic-function-class` option specifies that the generic function is to have a different class from the default provided by the system, that is, different from the class `<standard-generic-function>`. The *class-name* argument is the name of a class that can be the class of a generic function.
- The `:method-combination` option is followed by a symbol or keyword that names a type of method combination. The names of the built-in method combination types are `nil` and `standard`.

The *method-desc* arguments define methods that will belong to the generic function, as if defined by `defmethod`. The *method-qualifier* and *parameter-profile* arguments in a method description are the same as for `defmethod`. The *form* arguments specify the method body.

If no method descriptions are specified, a generic function with no methods is created. An error shall be signaled if a generic function is called and no methods apply.

The *lambda-list* argument of `defgeneric` specifies the shape of lambda-lists for the methods on this generic function. All methods on the resulting generic function must have parameter-profiles that are congruent with this shape. For further details on method congruence, see §15.2.2.2.

Implementations can extend `defgeneric` to include other implementation-defined options.

15.2.2 Defining methods for generic functions

(`defmethod` *func-spec* *method-qualifier** *parameter-profile* *form**)
→ <*symbol*> **defining operator**

Where:

func-spec	::=	*identifier* | (`setf` *identifier*)
method-qualifier	::=	*identifier* | *keyword*
parameter-profile	::=	({*var* | (*var* *parameter-specializer-name*)}* [{`&rest` | `:rest`} *var*])
parameter-specializer-name	::=	*class-name*
class-name	::=	*identifier*
var	::=	*identifier*

The `defmethod` defining form defines a method on a generic function. It returns the generic function name *func-spec*.

A method-defining form contains the code that is to be executed when the arguments to the generic function cause the method that it defines to be invoked.

Preparing a method-defining form for execution causes one of the following cases:

- It is a violation if the given name *func-spec* already designates a generic function and this generic function contains a method that agrees with the new one on parameter specializers and qualifiers. For a definition of one method agreeing with another on parameter specializers and qualifiers, see the section §15.2.2.1.
- If the given name *func-spec* designates a generic function and this generic function does not contain a method that agrees with the new one on parameter specializers and qualifiers, the new method is added to the generic function.
- It is a violation if the `defmethod` defining form is in the scope of a *func-spec* identifier that does not designate a generic function.
- It is a violation if the given name *func-spec* does not exist in the current toplevel scope immediately containing the `defmethod` defining form. Furthermore, it is a violation if a `defgeneric` form for *func-spec* does not precede the method-defining form in the text unit being prepared for execution unless the method-defining form is a `defclass`.

The parameter-profile of the method being defined must be congruent with the lambda-list of the generic function. See §15.2.2.2 for a definition of congruence in this context.

Each *method-qualifier* argument is an object that is used as an attribute to the given method by method combination. A method qualifier is a non-`nil` symbol or keyword. The method combination type further restricts what a method qualifier may be. The standard method combination type allows for unqualified methods or methods whose sole qualifier is one of the keywords `:before`, `:after`, `:around`.

The *parameter-profile* argument is like an ordinary function lambda-list except that the names of required parameters can be replaced by specialized parameters. A specialized parameter is a list of the form (*variable-name parameter-specializer-name*). Only required parameters may be specialized. A parameter specializer name is an identifier that names a class. If no parameter specializer name is specified for a given required parameter, the parameter specializer defaults to the class named `<object>`.

The *form* arguments specify the method body.

No two methods with agreeing parameter specializers and qualifiers may be defined for the same generic function. See the section §15.2.2.1 for a definition of agreement in this context.

A method is not a function and cannot be invoked as a function.

15.2.2.1 Agreement on parameter specializers and qualifiers Two methods are said to agree with each other on parameter specializers and qualifiers if the following conditions hold:

1. Both methods have the same number of required parameters. Suppose the parameter specializers of the two methods are $P_{1,1} \ldots P_{1,n}$ and $P_{2,1} \ldots P_{2,n}$.
2. For each $1 \leq i \leq n$, $P_{1,i}$ agrees with $P_{2,i}$. The parameter specializer $P_{1,i}$ agrees with $P_{2,i}$ if $P_{1,i}$ and $P_{2,i}$ denote the same class. Otherwise $P_{1,i}$ and $P_{2,i}$ do not agree.
3. The qualifiers of both methods, if any, are the same.

The parameter specializers are derived from the parameter profiles as described above.

15.2.2.2 Congruent lambda-lists for all methods of a generic function These rules define the congruence of a set of lambda-lists, including the parameter profile of each method for a given generic function and the lambda-list specified for the generic function itself, if given.

1. Each lambda-list must have the same number of required parameters.
2. If any lambda-list mentions `&rest` or `:rest`, each lambda-list must mention `&rest` or `:rest`.

15.2.3 Inheritance of methods

A subclass inherits methods in the following sense: Any method applicable to all instances of a class is also applicable to all instances of any subclass of that class, since they are also instances of that class.

The inheritance of methods acts the same way regardless of whether the method was created by using the `defmethod` form or by using one of the `defclass` options that causes methods to be generated automatically.

15.3 Calling generic functions

When a generic function is called with particular arguments, it must determine the code to execute. This code is called the ***effective method*** for those arguments. The effective method is a ***combination*** of the applicable methods in the generic function, which might be some or all of the defined methods. An error shall be signaled if a generic function is called and no methods apply.

When the effective method has been determined, it is invoked with the same arguments that were passed to the generic function. Whatever value it returns is returned as the value of the generic function.

The effective method is determined by the following three-step procedure:

1. Select the applicable methods.
2. Sort the applicable methods by precedence order, putting the most specific method first.
3. Apply applicable methods according to the method combination.

15.3.1 Selecting the applicable methods

Given a generic function and a set of arguments, an ***applicable method*** is a method for that generic function whose parameter specializers are satisfied by their corresponding arguments. The following definition specifies what it means for a method to be applicable and for an argument to satisfy a parameter specializer.

Let $\langle A_1, \ldots, A_n \rangle$ be the required arguments to a generic function in order. Let $\langle P_1, \ldots, P_n \rangle$ be the parameter specializers corresponding to the required parameters of the method M in order. The method M is ***applicable*** when each A_i ***satisfies*** P_i. If A_i is an instance of a class C, then it is said that A_i *satisfies* P_i when $C = P_i$ or when C is a subclass of P_i.

A method all of whose parameter specializers are the class named `<object>` is called a ***default method***; it is always applicable but might be shadowed by a more specific method.

Methods can have ***qualifiers***, which give the method combination procedure a way to distinguish among methods. A method that has one or more qualifiers is called a ***qualified method***. A method with no qualifiers is called an ***unqualified method***. Any object after `:method` and before the first list in *method-desc* is regarded as a qualifier, but only non-`nil` symbols and keywords are accepted as qualifiers. The qualifiers defined by standard method combination are keywords.

15.3.2 Sorting the applicable methods

To compare the precedence of two methods, their parameter specializers are examined in order. The examination order is from left to right.

The corresponding parameter specializers from each method are compared. When a pair of parameter specializers are equal, the next pair are compared for equality. If all corresponding parameter specializers are equal, the two methods must have different qualifiers; in this case, either method can be selected to precede the other.

If some corresponding parameter specializers are not equal, the first pair of parameter specializers that are not equal determines the precedence. The more specific of the two methods is the method whose parameter specializer appears earlier in the class precedence list of the corresponding argument. Because of the way in which the set of applicable methods is chosen, the parameter specializers are guaranteed to be present in the class precedence list of the class of the argument.

The resulting list of applicable methods has the most specific method first and the least specific method last.

15.3.3 Applying methods

In general, the effective method is some combination of the applicable methods. It is defined by a form that contains calls to some or all of the applicable methods, returns the value that will be returned as the value of the generic function, and optionally makes some of the methods accessible by means of `call-next-method`. This form is the body of the effective method; it is augmented with an appropriate lambda-list to make it a function.

The role of each method in the effective method is determined by its method qualifiers and the specificity of the method. A qualifier serves to mark a method, and the meaning of a qualifier is determined by the way that these marks are used by this step of the procedure. An error shall be signaled if an applicable method has an unrecognized qualifier.

ISLisp provides two method combination types. To specify that a generic function is to use one of these method combination types, the name of the method combination type is given as the argument to the `:method-combination` option to `defgeneric`.

The names of the method combination types are `nil` and `standard`.

15.3.3.1 Simple method combination In the simple case—the `nil` method combination type where all applicable methods are primary methods—the effective method calls the most specific method. That method can call the next most specific method by using `call-next-method`. The method that `call-next-method` calls is referred to as the ***next method***. The predicate `next-method-p` tests whether a next method exists. An error shall be signaled if `call-next-method` is called and there is no next most specific method.

15.3.3.2 Standard method combination Standard method combination is used if no other type of method combination is specified or if the method combination `standard` is specified.

Primary methods define the main action of the effective method, while ***auxiliary methods*** modify that action in one of three ways. A primary method has no method qualifiers. An auxiliary method is a method whose method qualifier is `:before`, `:after`, or `:around`.

- A `:before` method has the keyword `:before` as its qualifier. A `:before` method specifies code that is to be run before any primary methods.

- An `:after` method has the keyword `:after` as its qualifier. An `:after` method specifies code that is to be run after primary methods.
- An `:around` method has the keyword `:around` as its qualifier. An `:around` method specifies code that is to be run instead of other applicable methods but which is able to cause some of them to be run.

The semantics of standard method combination is as follows:

- If there are any `:around` methods, the most specific `:around` method is called. It supplies the value of the generic function.
- Inside the body of an `:around` method, `call-next-method` can be used to call the next method. When the next method returns, the `:around` method can execute more code, perhaps based on the returned value. An error shall be signaled if `call-next-method` is used and there is no applicable method to call. The function `next-method-p` can be used to determine whether a next method exists.
- If an `:around` method invokes `call-next-method`, the next most specific `:around` method is called, if one is applicable. If there are no `:around` methods or if `call-next-method` is called by the least specific `:around` method, the other methods are called as follows:
- All the `:before` methods are called, in most-specific-first order. Returned values are ignored. An error shall be signaled if `call-next-method` is used in a `:before` method.
- The most specific primary method is called. Inside the body of a primary method, the form `call-next-method` can be used to call the next most specific primary method. When that method returns, the previous primary method can execute more code, perhaps based on the returned value. An error shall be signaled if `call-next-method` is used and there are no more applicable primary methods. The `next-method-p` function can be used to determine whether a next method exists. If `call-next-method` is not used, only the most specific primary method is called.
- All the `:after` methods are called in most-specific-last order. Returned values are ignored. An error shall be signaled if `call-next-method` is used in an `:after` method.
- If no `:around` methods were invoked, the most specific primary method supplies the value returned by the generic function. The value returned by the invocation of `call-next-method` in the least specific `:around` method is that returned by the most specific primary method.

An error shall be signaled if there is an applicable method but no applicable primary method while using standard method combination.

The `:before` methods are run in most-specific-first order while the `:after` methods are run in least-specific-first order. The design rationale for this difference can be illustrated with an example. Suppose class C_1 modifies the behavior of its superclass, C_2, by adding `:before` and `:after` methods. Whether the behavior of the class C_2 is defined directly by methods on C_2 or is inherited from its superclasses does not affect the relative order of invocation of methods on instances of the class C_1. Class C_1's `:before` method runs before all of class C_2's methods. Class C_1's `:after` method runs after all of class C_2's methods.

By contrast, all `:around` methods run before any other methods run. Thus a less specific `:around` method runs before a more specific primary method.

If only primary methods are used and if `call-next-method` is not used, only the most specific method is invoked; that is, more specific methods shadow more general ones.

15.3.4 Calling more general methods

`(call-next-method)` → <*object*> **local function**

The `call-next-method` function can be used within the body of a method to call the next method.

It returns the value returned by the method it calls.

The type of method combination used determines which methods can invoke `call-next-method` and what is the next method to be called.

In the case of simple method combination where the method combination type is `nil` the next method is the next most specific method.

The standard method combination type allows `call-next-method` to be used within primary methods and `:around` methods. The standard method combination type defines the next method as specified in §15.3.3.2.

`call-next-method` passes the current method's original arguments to the next method. Neither using `setq` nor rebinding variables with the same names as parameters of the method affects the values `call-next-method` passes to the method it calls. The `call-next-method` function returns the value returned by the method it calls. After `call-next-method` returns, further computation is possible. The `next-method-p` function can be used to test whether there is a next method.

The functional binding of `call-next-method` is lexical within the body of the method-defining form; *i.e.,* it is as if it were established by `labels`. The function object to which the binding refers has indefinite extent.

An error shall be signaled if `call-next-method` is used in methods that do not support it. An error shall be signaled if `call-next-method` is executed and there is no next method.

`(next-method-p)` → *boolean* **local function**

The `next-method-p` function can be used within the body of a method defined by a method-defining form to determine whether a next method exists. The `next-method-p` function takes no arguments and returns `t` or `nil`.

The functional binding of `next-method-p` is lexical within the body of the method-defining form; *i.e.,* it is as if it were established by `labels`. The function object to which the binding refers has indefinite extent.

15.4 Object creation and initialization

(`create` *class* {*initarg* *initval*}*) → <*object*> **generic function**

The function `create` creates and returns a new instance of a class. The argument is a class object.

The initialization of a new instance consists of several distinct steps, including the following: allocating storage for the instance, filling slots with values, and executing user-supplied methods that perform additional initialization. The last two steps of `create` are implemented by the generic function `initialize-object` to provide a mechanism for customizing those steps. The initialization arguments (the *initargs* and *initvals*) are given as a single list argument to `initialize-object`. The instance returned by `create` is the new instance, which has been modified and returned by `initialize-object`.

15.4.1 Initialize-object

The generic function `initialize-object` is called by `create` to initialize a newly created instance. It uses standard method combination. Methods for `initialize-object` can be defined on user-defined classes in order to augment or override the system-supplied slot-filling mechanisms (described below).

During initialization, `initialize-object` is invoked after a new instance whose slots are unbound has been created.

The generic function `initialize-object` is called with the new instance. There is a system-supplied primary method for `initialize-object` whose parameter specializer is the class `<standard-object>`. This method fills in the slots according to the initialization arguments provided and according to the `:initform` forms for the slots as follows:

- If the slot already has a value, no attempt is made to change that value.
- If an initialization argument and value pair for the slot was provided among the initialization arguments, the slot is initialized with the value from that pair. The name of the initialization argument for a slot is declared by the `:initarg` option to slots in `defclass`. The consequences are undefined if more than one initialization argument for the same slot is supplied.
- If the slot has a default initial value form (see `defclass`), that form is evaluated in the lexical environment in which that form was established and in the current dynamic environment. The result of the evaluation is an object which becomes the value of the slot.
- Otherwise, the slot is left uninitialized.

(`initialize-object` *instance* *initialization-arguments*) → <*object*> **generic function**

The generic function `initialize-object` is called by `create` to initialize a newly created instance. The generic function `initialize-object` is called with the new instance and a list of initialization arguments.

The system-supplied primary method on `initialize-object` initializes the slots of the instance with values according to the *initialization-arguments* (an alternating list of initialization argument keywords and values) and the `:initform` forms of the slots.

The *instance* argument is the object to be initialized. The modified instance is returned as the result. Programmers can define methods for `initialize-object` to specify actions to be taken when an instance is initialized. If only `:after` methods are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of `initialize-object`. The consequences are undefined if a programmer-defined primary method for this generic function does not return *instance*.

15.5 Class enquiry

`(class-of` *obj*`)` → <*class*> **function**

Returns the class of which the given *obj* is a direct instance. *obj* may be any ISLisp object.

`(instancep` *obj class*`)` → *boolean* **function**

Returns `t` if *obj* is an instance (directly or otherwise) of the class *class*; otherwise, returns `nil` *obj* may be any ISLisp object. An error shall be signaled if *class* is not a class object (error-id. *domain-error*).

`(subclassp` $class_1$ $class_2$`)` → *boolean* **function**

Returns `t` if the class $class_1$ is a subclass of the class $class_2$; otherwise, returns `nil`. An error shall be signaled if either $class_1$ or $class_2$ is not a class object (error-id. *domain-error*).

`(class` <u>*class-name*</u>`)` → <*class*> **special operator**

Returns the class object that corresponds to the class named *class-name*.

16 Macros

Macros are a feature to extend the language syntactically. A macro is an abstraction for surface transformations. Because ISLisp texts (*e.g.,* function definitions) can be represented internally by objects in ISLisp, the surface transformations can be described by means of list processing. Forms are represented by conses or other objects and a macro describes a transformation function from one group of objects onto another.

Macros can be internally defined by ***expander*** functions which implement the transformation from one group of objects to another. The operation of an expander functions is specified by a `defmacro` defining form.

An expander receives a form as argument and returns a different form as value. The primary activity of an expander is to create sets of nested lists; for this purpose, the backquote facility is provided.

Macros are expanded at preparation time. No runtime information is available.

The set of usable operations is restricted to simple data structure creation and manipulation; those operations are forbidden that cause side-effects to the environment (such as I/O to the terminal), to externally accessible data structure (such as a modification to the property list of a symbol), or to the macro form itself.

Macro definitions are allowed only at toplevel. Redefinition (*i.e.,* multiple definition) of macros is forbidden. A macro's definition must textually precede any use of that macro during preparation for execution.

The result of expanding a macro form is another form. If that form is a macro form, it is expanded by the same mechanism until the result is not a macro form.

When a toplevel form is a macro form, its resulting macro expansion is also considered to be a toplevel form.

A macro form can appear as the place specified in a `setf` special form. See `setf` on page 35.

`(defmacro` *macro-name* *lambda-list* *form**`)` → <*symbol*> **defining operator**

Defines a named (toplevel) macro. No implicit block with the macro name is established when the macro-expansion function is invoked. *macro-name* must be an identifier whose scope is the current toplevel scope in which the `defmacro` form appears. *lambda-list* is as defined in page 23. The definition point of *macro-name* is the closing parenthesis of the *lambda-list*.

Example:

```
(defmacro caar(x) (list 'car (list 'car x)))
  ⇒ caar
```

'*form* → <*object*> **syntax**
,*form* → <*object*> **syntax**
,@*form* → <*object*> **syntax**

' or ***quasiquote*** constructs a list structure. quasiquote, like `quote`, returns its argument unevaluated if no commas or the syntax `,` (unquote) or `,@` (unquote-splicing) appear within the *form*.

`,` (unquote) syntax is valid only within ' (quasiquote) expressions. When appearing within a quasiquote the *form* is evaluated and its result is inserted into the quasiquote structure instead of the unquote *form*.

`,@` (unquote-splicing) is also syntax valid only within ` expressions. When appearing within a quasiquote the expression *form* must evaluate to a list. The elements of the list are spliced into the enclosing list in place of the unquote-splicing *form* sequence.

Quasiquote forms may be nested. Substitutions are made only for unquoted expressions appearing at the same nesting level, which increases by one inside each successive quasiquotation and decreases by one inside each unquotation.

Example:

```
`(list ,(+ 1 2) 4)
  ⇒  (list 3 4)
(let ((name 'a)) `(list name ,name ',name))
  ⇒  (list name a (quote a))
`(a ,(+ 1 2) ,@(create-list 3 'x) b)
  ⇒  (a 3 x x x b)
`((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
   ⇒  ((foo 7) . cons)
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
  ⇒  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
    `(a `(b ,,name1 ,',name2 d) e))
  ⇒  (a `(b ,x ,'y d) e)
```

17 Declarations and coercions

(the <u>*class-name*</u> *form*) → <*object*> **special operator**
(assure <u>*class-name*</u> *form*) → <*object*> **special operator**

These forms evaluate *form*. If *form* returns, the returned value is returned by the `the` or `assure` form. In addition, these forms specify that the value of *form* is of the class specified by *class-name* (which must be the name of an existing class).

In a `the` special form, the consequences are undefined if the value of *form* is not of the class or a subclass of the class designated by *class-name*. In an `assure` special form, an error shall be signaled if the value of *form* is not of the class or a subclass of the class designated by *class-name* (error-id. *domain-error*).

Example:

```
(the <integer> 10)                 ⇒ 10
(the <number> 10)                  ⇒ 10
(the <float> 10)      the consequences are undefined
(assure <integer> 10)              ⇒ 10
(assure <number> 10)               ⇒ 10
```

```
(assure <float> 10)
```
an error shall be signaled

(`convert` *obj* <u>*class-name*</u>) → <*object*> **special operator**

Returns an object of class *class-name* which corresponds to the object *obj* according to one of the following projections, called a ***coercion function***. The table shows *obj* along the left-hand column and *class-name* along the top row (with `<>`'s in class names omitted here only for textual brevity):

	character	integer	float	symbol	string	general-vector	list
character	=	I	–	I(3)	–(4)	–	–
integer	I	=	X	–	X(5)	–	–
float	–	–(1)	=	–	X(6)	–	–
symbol	–	–	–	=	I(3)	–	–
string	–	X(2)	X(2)	I(3)	=	X(7)	X
general-vector	–	–	–	–	–	=	X
list	–	–	–	–	–	X	=

Legend:

= This is the identity function

X This coercion shall be provided

- **X(2)** An error shall be signaled if this coercion is attempted and the string does not contain the textual representation of a number of the target class. In all other respects, this is the same as `parse-number`.
- **X(5)** This may be the same as the `~D` format directive.
- **X(6)** This may be the same as the `~G` format directive.
- **X(7)** This is the identity if strings are vectors in the implementation.

I This coercion shall be provided, but its definition is implementation defined.

- **I(3)** This coercion shall be provided, but its definition is implementation defined. The coercion depends on the implementation's *neutral* alphabetic characters (see §18.1.2).

– An error shall be signaled if this coercion is attempted.

- **–(1)** Programmers requiring this kind of functionality may wish to consider instead using one of the functions, `floor`, `ceiling`, `round`, or `truncate`.
- **–(4)** programmers requiring this kind of functionality may wish to consider instead using the following: `(create-string 1` *obj*`)`

If an implementation provides implementation-defined classes, it may also provide implementation-defined coercions for the names of those classes using `convert`.

Example:

```
(convert 3 <float>)                  ⇒ 3.0
(convert "abc" <general-vector>)     ⇒ #(#\a #\b #\c)
(convert #(a b) <list>)              ⇒ (a b)
```

18 Symbol class

A symbol (an instance of class `<symbol>`) is an object. Symbols can be ***named*** or ***unnamed***. A symbol's name is sometimes called a ***print name*** because it is used to identify the symbol during reading and printing. Symbols can have associated ***properties***.

(symbolp *obj*) → *boolean* **function**

Returns `t` if *obj* is a symbol (instance of class `<symbol>`); otherwise, returns `nil`. The *obj* may be any ISLISP object.

Example:

```
(symbolp 'a)          ⇒ t
(symbolp "a")         ⇒ nil
(symbolp #\a)         ⇒ nil
(symbolp 't)          ⇒ t
(symbolp t)           ⇒ t
(symbolp 'nil)        ⇒ t
(symbolp nil)         ⇒ t
(symbolp '())         ⇒ t
(symbolp '*pi*)       ⇒ t
(symbolp *pi*)        ⇒ nil
```

18.1 Symbol names

Symbols can be either ***named*** or ***unnamed***.

There is a mapping from names to symbols. Distinct symbols (symbols that are not `eq`) always have distinct names. No such mapping is defined for *unnamed* symbols.

The name of a symbol is represented as a string.

18.1.1 Notation for symbols

The constituent characters of a symbol's name are described in §6.

A *named* symbol is denoted by its *print name* enclosed within the vertical bars (“`|`”). However, the enclosing vertical bars are not necessary if the symbol satisfies the following conditions:

1. The symbol’s print name consists only of *neutral* alphabetic characters (see §18.1.2) or the following additional characters:

   ```
   0 1 2 3 4 5 6 7 8 9 + - < > / * = ? _ ! $ % [ ] ^ { } ~
   ```

   (This set may have additional implementation-defined characters.)

2. The first character of the symbol’s print name is a neutral alphabetic character or one of the following characters:

   ```
   < > / * = ? _ ! $ % [ ] ^ { } ~
   ```

   (This set may have additional implementation-defined characters.)

In addition, the following are the names of symbols that can be written without enclosing vertical bars:

```
+   -   1+   1-
```

If the symbol name contains a vertical bar, the vertical bar must be preceded by a backslash “\”. If the symbol name contains a backslash, the backslash must be preceded by another backslash. For example, “`|\\\\\|\\\||`” denotes a symbol whose name is a 5 character string containing backslash, backslash, vertical bar, backslash, vertical bar.

Note: All required symbols can be written without vertical bars.

It is implementation defined whether the names of symbols can contain colon (`:`) or ampersand (`&`). Consequently, whether `&rest` and keywords (*e.g.,* `:rest`, `:before`, and `:after`) are represented as symbols or something else is implementation defined.

18.1.2 Alphabetic case in symbol names

If the enclosing vertical bars are omitted, the case of alphabetic characters in a symbol is translated by the reader to a canonical case that is used internally. Therefore, for example, each of the following denotes the same symbol in all implementations:

```
foo   foO   fOo   fOO   Foo   FoO   FOo   FOO
```

Internally, alphabetic case in a symbol’s print name is maintained, and is significant. For example, `|FOO|` and `|foo|` are not the same symbol in any implementation. However, the reader canonicalizes the case of symbols whose names are not written enclosed by vertical bars. So `foo` and `FOO` both represent the same symbol, but it is implementation defined whether that symbol is `|foo|` or `|FOO|`.

Specifically, each implementation has an implementation-defined attribute called its ***neutral alphabetic case***, which is either “lowercase” or “uppercase.” If the neutral alphabetic case of an implementation is lowercase, the ***neutral alphabetic characters*** for that implementation are defined to be as follows:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Otherwise (if the neutral alphabetic case of an implementation is uppercase), the neutral alphabetic characters for that implementation are defined to be as follows:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Continuing again with the above example, if the neutral alphabetic case of an implementation is lowercase, `foo`, `FOO` and `|foo|` denote the same symbol; otherwise, `foo`, `FOO` and `|FOO|` denote the same symbol.

An implementation may extend the set of neutral alphabetic characters to include additional implementation-defined characters.

18.1.3 `nil` and `()`

The symbol `nil`, which represents both the false value and the empty list, can also be denoted by `()`.

18.2 Symbol properties

A ***property*** of a symbol is a named association between a ***property indicator*** and a ***property value***. A symbol s is said to have a property p if a property indicator p is associated with s.

(`property` *symbol* *property-name* [*obj*]) → <*object*> **function**

Returns the value of the property named *property-name* associated with the symbol *symbol*. If *symbol* has no property named *property-name*, *obj* (which defaults to `nil`) is returned.

An error shall be signaled if either *symbol* or *property-name* is not a symbol (error-id. *domain-error*). *obj* may be any ISLisp object

Example:

```
(property 'zeus 'daughter)          ⇒  athena
```

(`setf` (`property` *symbol* *property-name*) *obj*) → <*object*> **special form**
(`set-property` *obj* *symbol* *property-name*) → <*object*> **function**

Causes *obj* to be the new value of the property named *property-name* asssociated with the symbol *symbol*. If the property named *property-name* already exists, its corresponding property value is replaced; otherwise, a new property is created. *obj* is returned.

An error shall be signaled if either *symbol* or *property-name* is not a symbol (error-id. *domain-error*). *obj* may be any ISLisp object

Example:

```
(setf (property 'zeus 'daughter) 'athena)
  ⇒ athena
(set-property 'athena 'zeus 'daughter)
  ⇒ athena
```

(remove-property *symbol property-name*) → <*object*> **function**

Removes the property *property-name* associated with *symbol* and returns the property value of the removed property if there is such a property. If there is no such property, `nil` is returned.

An error shall be signaled if either *symbol* or *property-name* is not a symbol (error-id. *domain-error*).

Example:

```
(remove-property 'zeus 'daughter)   ⇒ athena
```

18.3 Unnamed symbols

(gensym) → <*symbol*> **function**

Returns an unnamed symbol. `gensym` is useful for writing macros. It is impossible for an identifier to name an unnamed symbol.

Example:

```
(defmacro twice (x)
  (let ((v (gensym)))
    `(let ((,v ,x)) (+ ,v ,v))))     ⇒ twice
(twice 5)                           ⇒ 10
```

19 Number class

The class `<number>` has the disjoint subclasses `<float>` and `<integer>`.

19.1 Number class

(numberp *obj*) → *boolean* **function**

Returns t if *obj* is a number (instance of class <number>); otherwise, returns nil. The *obj* may be any ISLISP object.

Example:

```
(numberp 3)                    ⇒ t
(numberp -0.3)                 ⇒ t
(numberp '(a b c))             ⇒ nil
(numberp "17")                 ⇒ nil
```

(parse-number *string*) → *<number>* **function**

The characters belonging to *string* are scanned (as if by read) and if the resulting lexeme is the textual representation of a number, the number it represents is returned.

An error shall be signaled if *string* is not a string (error-id. *domain-error*). An error shall be signaled if *string* is not the textual representation of a number (error-id. *cannot-parse-number*).

Example:

```
(parse-number "123.34")           ⇒ 123.34
(parse-number "#XFACE")           ⇒ 64206
(parse-number "-37.")      an error shall be signaled
(parse-number "-.5")       an error shall be signaled
                 since floating-point number lexemes have
                 at least one mantissa digit before and at least
                 one mantissa digit after the decimal point.
```

(= x_1 x_2) → *boolean* **function**

Returns t if x_1 has the same mathematical value as x_2; otherwise, returns nil. An error shall be signaled if either x_1 or x_2 is not a number (error-id. *domain-error*).

Note: = differs from eql because = compares only the mathematical values of its arguments, whereas eql also compares the representations.

Example:

```
(= 3 4)                           ⇒ nil
(= 3 3.0)                         ⇒ t
(= (parse-number "134.54") 134.54) ⇒ t
```

```
(= 0.0 -0.0)                        ⇒ t
```

(/= x_1 x_2) → *boolean* **function**

Returns `t` if x_1 and x_2 have mathematically distinct values; otherwise, returns `nil`. An error shall be signaled if either x_1 or x_2 is not a number (error-id. *domain-error*).

Example:

```
(/= 3 4)                            ⇒ t
(/= 3 3.0)                          ⇒ nil
(/= (parse-number "134.54") 134.54)⇒ nil
```

(>= x_1 x_2) → *boolean* **function**
(<= x_1 x_2) → *boolean* **function**
(> x_1 x_2) → *boolean* **function**
(< x_1 x_2) → *boolean* **function**

`>=` returns `t` if x_1 is greater than or `=` x_2. `<=` returns `t` if x_1 is less than or `=` x_2. `>` returns `t` if x_1 is greater than x_2. `<` returns `t` if x_1 is less than x_2.

The mathematical values of the arguments are compared. An error shall be signaled if either x_1 or x_2 is not a number (error-id. *domain-error*).

Example:

```
(> 2 2)                             ⇒ nil
(> 2.0 2)                           ⇒ nil
(> 2 -10)                           ⇒ t
(> 100 3)                           ⇒ t
(< 2 2)                             ⇒ nil
(< 1 2)                             ⇒ t
(>= 2 2)                            ⇒ t
(>= 2.0 2)                          ⇒ t
(>= -1 2)                           ⇒ nil
(<= -1 2)                           ⇒ t
(<= 2 -1)                           ⇒ nil
```

In the remaining definitions in this section, any noted coercion to `<float>` is done as if by `float` or by `(convert` z `<float>)`.

(+ x*) → <*number*> **function**
(* x*) → <*number*> **function**

The functions `+` and `*` return the sum and the product, respectively, of their arguments. If all arguments are integers, the result is an integer. If any argument is a float, the result is a float. When given no arguments, `+` returns 0 and `*` returns 1. An error shall be signaled if any x is not a number (error-id. *domain-error*).

Example:

```
(+ 12 3)                             ⇒ 15
(+ 1 2 3)                            ⇒ 6
(+ 12 3.0)                           ⇒ 15.0
(+ 4 0.0)                            ⇒ 4.0
(+)                                  ⇒ 0
(* 12 3)                             ⇒ 36
(* 12 3.0)                           ⇒ 36.0
(* 4.0 0)                            ⇒ 0.0
(* 2 3 4)                            ⇒ 24
(*)                                  ⇒ 1
```

`(-` x^+`)` → <*number*> **function**

Given one argument, x, this function returns its additive inverse. An error shall be signaled if x is not a number (error-id. *domain-error*).

If an implementation supports a `-0.0` that is distinct from `0.0`, then `(- 0.0)` returns `-0.0`; in implementations where `-0.0` and `0.0` are not distinct, `(- 0.0)` returns `0.0`.

Example:

```
(- 1)                                ⇒ -1
(- -4.0)                             ⇒ 4.0
(- 4.0)                              ⇒ -4.0
(eql (- 0.0) -0.0)                   ⇒ t
(eql (- -0.0) 0.0)                   ⇒ t
```

Given more than one argument, $x_1 \ldots x_n$, `-` returns their successive differences, $x_1 - x_2 - \ldots - x_n$. An error shall be signaled if any x is not a number (error-id. *domain-error*).

Example:

```
(- 1 2)                              ⇒ -1
(- 92 43)                            ⇒ 49
(- 2.3 -3.0)                         ⇒ 5.3
(- 0.0 0.0)                          ⇒ 0.0
(- 3 4 5)                            ⇒ -6
```

(quotient *dividend* *divisor*$^+$) → <*number*> **function**
(reciprocal *x*) → <*number*> **function**

The function `reciprocal` returns the reciprocal of its argument x; that is, $1/x$. An error shall be signaled if x is zero (error-id. *division-by-zero*).

The function `quotient`, given two arguments *dividend* and *divisor*, returns the quotient of those numbers. The result is an integer if *dividend* and *divisor* are integers and *divisor* evenly divides *dividend*, otherwise it will be a float.

Given more than two arguments, `quotient` operates iteratively on each of the $divisor_1$ ... $divisor_n$ as in $dividend/divisor_1/\ldots/divisor_n$. The type of the result follows from the two-argument case because the three-or-more-argument `quotient` can be defined as follows:

(quotient *dividend* $divisor_1$ $divisor_2$ ...)
≡ (quotient (quotient *dividend* $divisor_1$) $divisor_2$ ...)

An error shall be signaled if *dividend* is not a number (error-id. *domain-error*). An error shall be signaled if any *divisor* is not a number (error-id. *domain-error*). An error shall be signaled if any *divisor* is zero (error-id. *division-by-zero*).

Example:

```
(reciprocal 2)                ⇒ 0.5
(quotient 10 5)               ⇒ 2
(quotient 1 2)                ⇒ 0.5
(quotient 2 -0.5)             ⇒ -4.0
(quotient 0 0.0)     an error shall be signaled
(quotient 2 3 4)              ⇒ 0.16666666666666666
```

(max x^+) → <*number*> **function**
(min x^+) → <*number*> **function**

The function `min` returns the least (closest to negative infinity) of its arguments. The comparison is done by `<`.

The function `max` returns the greatest (closest to positive infinity) of its arguments. The comparison is done by `>`.

An error shall be signaled if any x is not a number (error-id. *domain-error*).

Example:

```
(max -5 3)                    ⇒ 3
(max 2.0 3)                   ⇒ 3
(max 2 2.0)                   ⇒ 2 or 2.0   (implementation-defined)
```

`(max 1 5 2 4 3)`	⇒ `5`
`(min 3 1)`	⇒ `1`
`(min 1 2.0)`	⇒ `1`
`(min 2 2.0)`	⇒ `2` *or* `2.0` *(implementation-defined)*
`(min 1 5 2 4 3)`	⇒ `1`

`(abs` *x*`)` → <*number*> **function**

The function `abs` returns the absolute value of its argument. An error shall be signaled if x is not a number (error-id. *domain-error*).

Example:

`(abs -3)`	⇒ `3`
`(abs 2.0)`	⇒ `2.0`
`(abs -0.0)`	⇒ `0.0`

`(exp` *x*`)` → <*number*> **function**

Returns e raised to the power x, where e is the base of the natural logarithm. An error shall be signaled if x is not a number (error-id. *domain-error*).

Example:

`(exp 1)`	⇒ `2.718281828459045`
`(exp 2)`	⇒ `7.38905609893065`
`(exp 1.23)`	⇒ `3.4212295362896734`
`(exp 0)`	⇒ `1` *or* `1.0` *(implementation-defined)*

`(log` *x*`)` → <*number*> **function**

Returns the natural logarithm of x. An error shall be signaled if x is not a positive number (error-id. *domain-error*).

Example:

`(log 2.718281828459045)`	⇒ `1.0`
`(log 10)`	⇒ `2.302585092994046`
`(log 1)`	⇒ `0` *or* `0.0` *(implementation-defined)*

(expt x_1 x_2) → <*number*> **function**

Returns x_1 raised to the power x_2. The result will be an integer if x_1 is an integer and x_2 is a non-negative integer. An error shall be signaled if x_1 is zero and x_2 is negative, or if x_1 is zero and x_2 is a zero float, or if x_1 is negative and x_2 is not an integer.

Example:

```
(expt 2 3)                    ⇒ 8
(expt -100 2)                 ⇒ 10000
(expt 4 -2)                   ⇒ 0.0625
(expt 0.5 2)                  ⇒ 0.25
(expt x 0)                    ⇒ 1     if x is an integer
(expt x 0)                    ⇒ 1.0   if x is a float
(expt -0.25 -1)               ⇒ -4.0
(expt 100 0.5)                ⇒ 10.0
(expt 100 -1.5)               ⇒ 0.001
(expt x 0.0)                  ⇒ 1.0   if x is a positive float
(expt 0.0 0.0)     an error shall be signaled
```

(sqrt x) → <*number*> **function**

Returns the non-negative square root of x. An error shall be signaled if x is not a non-negative number (error-id. *domain-error*).

Example:

```
(sqrt 4)                      ⇒ 2
(sqrt 2)                      ⇒ 1.4142135623730951
(sqrt -1)     an error shall be signaled
```

---

pi → <*float*> **named constant**

The value of this constant is an approximation of π.

Example:

```
*pi*                          ⇒ 3.141592653589793
```

(sin x) → <*number*> **function**
(cos x) → <*number*> **function**
(tan x) → <*number*> **function**

The function `sin` returns the sine of x. The function `cos` returns the cosine of x. The function `tan` returns the tangent of x. In each case, x must be given in radians.

An error shall be signaled if x is not a number (error-id. *domain-error*).

Example:

```
;; Note that conforming processors are permitted to vary
;; in the floating precision of these results.
(sin 1)                    ⇒ 0.8414709848078965
(sin 0)                    ⇒ 0 or 0.0  (implementation-defined)
(sin 0.001)                ⇒ 9.999998333333417E-4
(cos 1)                    ⇒ 0.5403023058681398
(cos 0)                    ⇒ 1 or 1.0  (implementation-defined)
(cos 0.001)                ⇒ 0.9999995000000417
(tan 1)                    ⇒ 1.557407724654902
(tan 0)                    ⇒ 0 or 0.0  (implementation-defined)
(tan 0.001)                ⇒ 0.0010000003333334668
```

(atan x) → <*number*> **function**

Returns the arc tangent of x.

The result is a (real) number that lies between $-\pi/2$ and $\pi/2$ (both exclusive).

The following definition for (one-argument) arc tangent determines the range and branch cuts:

$$\texttt{arctan}\, x = \frac{\texttt{log}\,(1+ix) - \texttt{log}\,(1-ix)}{2i}$$

An error shall be signaled if x is not a number (error-id. *domain-error*).

(atan2 x_1 x_2) → <*number*> **function**

Given a point (x_2, x_1) in rectangular coordinates, this function returns the phase of its representation in polar coordinates. If x_1 is zero and x_2 is negative, the result is positive. If x_1 and x_2 are both zero, the result is implementation defined.

An error shall be signaled if x is not a number (error-id. *domain-error*).

The value of `atan2` is always between $-\pi$ (exclusive) and π (inclusive) when minus zero is not supported; when minus zero is supported, the range includes $-\pi$.

	y Condition	x Condition	Cartesian locus	Range of result
	$y = 0$	$x > 0$	Positive x-axis	0
*	$y = +0$	$x > 0$	Positive x-axis	$+0$
*	$y = -0$	$x > 0$	Positive x-axis	-0
	$y > 0$	$x > 0$	Quadrant I	$0 <$ result $< \pi/2$
	$y > 0$	$x = 0$	Positive y-axis	$\pi/2$
	$y > 0$	$x < 0$	Quadrant II	$\pi/2 <$ result $< \pi$
	$y = 0$	$x < 0$	Negative x-axis	π
*	$y = +0$	$x < 0$	Negative x-axis	π
*	$y = -0$	$x < 0$	Negative x-axis	$-\pi$
	$y < 0$	$x < 0$	Quadrant III	$-\pi <$ result $< -\pi/2$
	$y < 0$	$x = 0$	Negative y-axis	$-\pi/2$
	$y < 0$	$x > 0$	Quadrant IV	$-\pi/2 <$ result < 0
	$y = 0$	$x = 0$	Origin	implementation defined
*	$y = +0$	$x = +0$	Origin	$+0$
*	$y = -0$	$x = +0$	Origin	-0
*	$y = +0$	$x = -0$	Origin	π
*	$y = -0$	$x = -0$	Origin	$-\pi$

Figure 3. Quadrant information for `atan2`

The signs of x_1 (indicated as y) and x_2 (indicated as x) are used to derive quadrant information. Figure 19.1 details various special cases. The asterisk (*) indicates that the entry in the figure applies to implementations that support minus zero.

Example:

```
(atan2 0 3.0)                       ⇒ 0 or 0.0  (implementation-defined)
(atan2 1 1)                         ⇒ 0.7853981633974483
(atan2 1.0 -0.3)                    ⇒ 1.8622531212727635
(atan2 0.0 -0.5)                    ⇒ 3.141592653589793
(atan2 -1 -1)                       ⇒ -2.356194490192345
(atan2 -1.0 0.3)                    ⇒ -1.2793396
(atan2 0.0 0.5)                     ⇒ 0.0

(defun asin (x) (atan2 x (sqrt (- 1 (expt x 2)))))  ⇒ asin

(defun acos (x) (atan2 (sqrt (- 1 (expt x 2))) x))  ⇒ acos

(defun atan (x) (atan2 x 1))  ⇒ atan
```

`(sinh` *x*`)` → <*number*> **function**
`(cosh` *x*`)` → <*number*> **function**
`(tanh` *x*`)` → <*number*> **function**

The function `sinh` returns the hyperbolic sine of x. The function `cosh` returns the hyperbolic cosine of x. The function `tanh` returns the hyperbolic tangent of x.

An error shall be signaled if x is not a number (error-id. *domain-error*).

Example:

```
(sinh 1)        ⇒ 1.1752011936438014
(sinh 0)        ⇒ 0 or 0.0   (implementation-defined)
(sinh 0.001)    ⇒ 0.00100000016666675
(cosh 1)        ⇒ 1.5430806348152437
(cosh 0)        ⇒ 1 or 1.0   (implementation-defined)
(cosh 0.001)    ⇒ 1.0000005000000416
(tanh 1)        ⇒ 0.7615941559557649
(tanh 0)        ⇒ 0 or 0.0   (implementation-defined)
(tanh 0.001)    ⇒ 9.999996666668002E-4
```

`(atanh x)` → *<number>* **function**

Returns the hyperbolic arc tangent of x. An error shall be signaled if x is not a number with absolute value less than 1 (error-id. *domain-error*).

Example:

```
(atanh 0.5)     ⇒ 0.5493061443340549
(atanh 0)       ⇒ 0 or 0.0 (implementation-defined)
(atanh 0.001)   ⇒ 0.0010000003333335335

(defun asinh (x) (atanh (quotient x (sqrt (+ 1 (expt x 2))))))  ⇒ asinh

(defun acosh (x) (atanh (quotient (sqrt (* (- x 1) (+ x 1))) x)))  ⇒ acosh
```

19.2 Float class

This class represents the set of floating-point numbers. Each floating-point number is represented by a rational number with some given precision; see IEEE standard 754-1985 for details.

Floating-point numbers are written in one of the following formats:

$[s]dd\ldots d.dd\ldots d$
$[s]dd\ldots d.dd\ldots d\mathtt{E}[s]dd\ldots d$
$[s]dd\ldots d.dd\ldots d\mathtt{e}[s]dd\ldots d$
$[s]dd\ldots d\mathtt{E}[s]dd\ldots d$
$[s]dd\ldots d\mathtt{e}[s]dd\ldots d$

where s is either "+" or "-", and $dd\ldots d$ is at least one digit from "0"–"9".

There must be at least one digit before the decimal point and at least one mantissa digit after the decimal point.

`*most-positive-float*` → <*float*> **named constant**
`*most-negative-float*` → <*float*> **named constant**

The value of `*most-positive-float*` is the implementation-dependent floating-point number closest to positive infinity.

The value of `*most-negative-float*` is the implementation-dependent floating-point number closest to negative infinity.

`(floatp` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is a float (instance of class `<float>`); otherwise, returns `nil`. The *obj* may be any ISLISP object.

Example:

```
(floatp "2.4")                    ⇒ nil
(floatp 2)                        ⇒ nil
(floatp 2.0)                      ⇒ t
```

`(float` *x*`)` → <*float*> **function**

Returns *x* itself if it is an instance of the class `<float>` and returns a floating-point approximation of *x* otherwise. An error shall be signaled if *x* is not a number (error-id. *domain-error*).

Example:

```
(float 0)                         ⇒ 0.0
(float 2)                         ⇒ 2.0
(float -2.0)                      ⇒ -2.0
(float 123456789123456789123456789) ⇒ 1.2345678912345679E26
```

`(floor` *x*`)` → <*integer*> **function**

Returns the greatest integer less than or equal to *x*. That is, *x* is truncated towards negative infinity. An error shall be signaled if *x* is not a number (error-id. *domain-error*).

Example:

```
(floor 3.0)                       ⇒ 3
```

```
(floor 3.4)            ⇒ 3
(floor 3.9)            ⇒ 3
(floor -3.9)           ⇒ -4
(floor -3.4)           ⇒ -4
(floor -3.0)           ⇒ -3
```

`(ceiling` *x*`)` → <*integer*> **function**

Returns the smallest integer that is not smaller than *x*. That is, *x* is truncated towards positive infinity. An error shall be signaled if *x* is not a number (error-id. *domain-error*).

Example:

```
(ceiling 3.0)          ⇒  3
(ceiling 3.4)          ⇒  4
(ceiling 3.9)          ⇒  4
(ceiling -3.9)         ⇒ -3
(ceiling -3.4)         ⇒ -3
(ceiling -3.0)         ⇒ -3
```

`(truncate` *x*`)` → <*integer*> **function**

Returns the integer between 0 and *x* (inclusive) that is nearest to *x*. That is, *x* is truncated towards zero. An error shall be signaled if *x* is not a number (error-id. *domain-error*).

Example:

```
(truncate 3.0)         ⇒ 3
(truncate 3.4)         ⇒ 3
(truncate 3.9)         ⇒ 3
(truncate -3.4)        ⇒ -3
(truncate -3.9)        ⇒ -3
(truncate -3.0)        ⇒ -3
```

`(round` *x*`)` → <*integer*> **function**

Returns the integer nearest to *x*. If *x* is exactly halfway between two integers, the even one is chosen. An error shall be signaled if *x* is not a number (error-id. *domain-error*).

Example:

```
(round 3.0)            ⇒ 3
```

```
(round 3.4)                    ⇒ 3
(round -3.4)                   ⇒ -3
(round 3.6)                    ⇒ 4
(round -3.6)                   ⇒ -4
(round 3.5)                    ⇒ 4
(round -3.5)                   ⇒ -4
(round 2.5)                    ⇒ 2
(round -0.5)                   ⇒ 0
```

19.3 Integer class

Integer objects correspond to mathematical integers.

Arithmetic operations that only involve integers behave in a mathematically correct way, regardless of the size of the integer. If there are cases where arithmetic on integers would produce results or intermediate expressions that exceed the precision of the underlying hardware, an ISLisp processor shall simulate any necessary operations in software in order to assure mathematical correctness. The circumstances, if any, for which such simulation is necessary is implementation defined; the point at which such simulation will exceed the capacity of the processor is also implementation defined.

Integers are written in one of the following formats.

`#B` $[s]bb\ldots b$, each b being either "0" or "1".
`#b` $[s]bb\ldots b$, each b being either "0" or "1".
`#O` $[s]oo\ldots o$, each o being one of "0"–"7".
`#o` $[s]oo\ldots o$, each o being one of "0"–"7".
$[s]dd\ldots d$, each d being one of "0"–"9".
`#X` $[s]xx\ldots x$, each x being one of "0"–"9", "A"–"F", "a"–"f".
`#x` $[s]xx\ldots x$, each x being one of "0"–"9", "A"–"F", "a"–"f".

where s is either "+" or "-".

Note: In ISLisp, there is no variable that controls the reader. Thus the above notations are exactly the notations for integers.

`(integerp` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is an integer (instance of class `<integer>`); otherwise, returns `nil`. *obj* may be any ISLisp object.

Example:

```
(integerp 3)                   ⇒ t
(integerp 3.4)                 ⇒ nil
(integerp "4")                 ⇒ nil
(integerp '(a b c))            ⇒ nil
```


(div z_1 z_2) → <*integer*> **function**
(mod z_1 z_2) → <*integer*> **function**

`div` returns the greatest integer less than or equal to the quotient of z_1 and z_2. An error shall be signaled if z_2 is zero (error-id. *division-by-zero*).

`mod` returns the remainder of the integer division of z_1 by z_2. The sign of the result is the sign of z_2. The result lies between 0 (inclusive) and z_2 (exclusive), and the difference of z_1 and this result is divisible by z_2 without remainder.

`div` and `mod` satisfy:

```
(= z1 (+ (* (div z1 z2) z2) (mod z1 z2)))
```

That is, the evaluation of the above form always return `t`.

An error shall be signaled if either z_1 or z_2 is not an integer (error-id. *domain-error*).

Example:

```
(div 12 3)          ⇒ 4
(div 14 3)          ⇒ 4
(div -12 3)         ⇒ -4
(div -14 3)         ⇒ -5
(div 12 -3)         ⇒ -4
(div 14 -3)         ⇒ -5
(div -12 -3)        ⇒ 4
(div -14 -3)        ⇒ 4
(mod 12 3)          ⇒ 0
(mod 7 247)         ⇒ 7
(mod 247 7)         ⇒ 2
(mod 14 3)          ⇒ 2
(mod -12 3)         ⇒ 0
(mod -14 3)         ⇒ 1
(mod 12 -3)         ⇒ 0
(mod 14 -3)         ⇒ -1
(mod -12 -3)        ⇒ 0
(mod -14 -3)        ⇒ -2
```

(gcd z_1 z_2) → <*integer*> **function**

`gcd` returns the greatest common divisor of its integer arguments. The result is a non-negative integer. For nonzero arguments the greatest common divisor is the largest integer z such that z_1 and z_2 are integral multiples of z.

An error shall be signaled if either z_1 or z_2 is not an integer (error-id. *domain-error*).

Example:

```
(gcd 12 5)                       ⇒ 1
(gcd 15 24)                      ⇒ 3
(gcd -15 24)                     ⇒ 3
(gcd 15 -24)                     ⇒ 3
(gcd -15 -24)                    ⇒ 3
(gcd 0 -4)                       ⇒ 4
(gcd 0 0)                        ⇒ 0
```

`(lcm` z_1 z_2`)` → <*integer*> **function**

`lcm` returns the least common multiple of its integer arguments. `gcd` and `lcm` satisfies:

```
(= (* (gcd m n) (lcm m n)) (abs (* m n)))
```

That is, the evaluation of the above form always return `t`.

An error shall be signaled if either z_1 or z_2 is not an integer (error-id. *domain-error*).

Example:

```
(lcm 2 3)                        ⇒ 6
(lcm 15 24)                      ⇒ 120
(lcm 15 -24)                     ⇒ 120
(lcm -15 24)                     ⇒ 120
(lcm -15 -24)                    ⇒ 120
(lcm 0 -4)                       ⇒ 0
(lcm 0 0)                        ⇒ 0
```

`(isqrt` z`)` → <*integer*> **function**

Returns the greatest integer less than or equal to the exact positive square root of z. An error shall be signaled if z is not a non-negative integer (error-id. *domain-error*).

Example:

```
(isqrt 49)                       ⇒ 7
(isqrt 63)                       ⇒ 7
(isqrt 1000000000000002000000000000000)
  ⇒ 1000000000000000
```

20 Character class

Characters are represented as instances of the <character> class. This insulates the programmer from particular character codes.

The ISLISP character set has at least ninety-five printing characters plus a newline character. The ISLISP printing characters are the space character, and the following ninety-four non-blank characters:

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

A character literal is denoted by #\ followed by a token which is either the character itself, or, if the character has a name, the character's name. For example, the letter A is denoted by "#\A". The newline and space characters have the names "newline" and "space," respectively, so they can be denoted by "#\newline" and "#\space". (Case is not significant when naming a character.)

Characters are ordered by char<, and this order satisfies:

```
0<1<2<3<4<5<6<7<8<9
A<B<C<D<E<F<G<H<I<J<K<L<M<N<O<P<Q<R<S<T<U<V<W<X<Y<Z
a<b<c<d<e<f<g<h<i<j<k<l<m<n<o<p<q<r<s<t<u<v<w<x<y<z
```

where *char1* < *char2* means that (char< *char1* *char2*) is true.

(characterp *obj*) → *boolean* **function**

Returns t if *obj* is a character (instance of class <character>); otherwise, returns nil. *obj* may be any ISLISP object.

Example:

```
(characterp #\a)      ⇒ t
(characterp "a")      ⇒ nil
(characterp 'a)       ⇒ nil
```

(char= $char_1$ $char_2$) → *boolean* **function**
(char/= $char_1$ $char_2$) → *boolean* **function**
(char< $char_1$ $char_2$) → *boolean* **function**
(char> $char_1$ $char_2$) → *boolean* **function**
(char<= $char_1$ $char_2$) → *boolean* **function**
(char>= $char_1$ $char_2$) → *boolean* **function**

The function char= tests whether $char_1$ is the same character as $char_2$. The function char< tests whether $char_1$ is less than $char_2$. The function char<= tests whether $char_1$ is less than or equal to $char_2$. The ordering used is the partial order defined above, extended to a total order on all characters in an implementation-defined manner. If the test is satisfied, `t` is returned; otherwise, `nil` is returned.

Two characters are `char/=` if and only if they are not `char=`. Two characters are `char>` if and only if they are not `char<=`. Two characters are `char>=` if and only if they are not `char<`.

An error shall be signaled if either $char_1$ or $char_2$ is not a character (error-id. *domain-error*).

Example:

```
(char= #\a #\a)          ⇒ t
(char= #\a #\b)          ⇒ nil
(char= #\a #\A)          ⇒ nil
(char/= #\a #\a)         ⇒ nil
(char< #\a #\a)          ⇒ nil
(char< #\a #\b)          ⇒ t
(char< #\b #\a)          ⇒ nil
(char< #\a #\A)          ⇒ nil or t   (implementation-defined)
(char< #\* #\a)          ⇒ nil or t   (implementation-defined)
(char> #\b #\a)          ⇒ t
(char<= #\a #\a)         ⇒ t
(char<= #\a #\A)         ⇒ nil or t   (implementation-defined)
(char>= #\b #\a)         ⇒ t
(char>= #\a #\a)         ⇒ t
```

21 List class

The `<list>` class is partitioned into two subclasses `<cons>` and `<null>`.

21.1 Cons

A cons (sometimes also called “dotted pair”) consists of two components; the left component is called car and the right component is called cdr. The constructor of this class is `cons`. Conses are written as

(*car* . *cdr*)

where *car* and *cdr* denote the values in the car and cdr components, respectively, of the cons object. As a special case, if the *cdr* value is `nil`, then the cons object is written as

(*car*)

Thus, in general, a data structure that consists of cons objects will be written in either of the following formats:

$(x_1 \;.\; (x_2 \;.\; \ldots\; (x_{n-1} \;.\; x_n)\; \ldots\;))$
$(x_1 \;.\; (x_2 \;.\; \ldots\; (x_{n-1})\; \ldots\;))$

These may be written, respectively, as

$(x_1\; x_2\; \ldots\; x_{n-1} \;.\; x_n)$
$(x_1\; x_2\; \ldots\; x_{n-1})$

`(consp` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is a cons (instance of class `<cons>`); otherwise, returns `nil`. *obj* may be any ISLisp object.

Example:

```
(consp '(a . b))          ⇒ t
(consp '(a b c))          ⇒ t
(consp '())               ⇒ nil
(consp #(a b))            ⇒ nil
```

`(cons` *obj*$_1$ *obj*$_2$`)` → *<cons>* **function**

This function builds a cons from two objects, with *obj*$_1$ as its car (or 'left') part and with *obj*$_2$ as its cdr (or 'right') part. An error shall be signaled if the requested cons cannot be allocated (error-id. *cannot-create-cons*). Both *obj*$_1$ and *obj*$_2$ may be any ISLisp object.

Example:

```
(cons 'a '())             ⇒ (a)
(cons '(a) '(b c d))      ⇒ ((a) b c d)
(cons "a" '(b c))         ⇒ ("a" b c)
(cons 'a 3)               ⇒ (a . 3)
(cons '(a b) 'c)          ⇒ ((a b) . c)
```

`(car` *cons*`)` → *<object>* **function**

The function `car` returns the left component of the *cons*. An error shall be signaled if *cons* is not a cons (error-id. *domain-error*).

Example:

```
(car '())                 an error shall be signaled
(car '(a b c))            ⇒ a
```

```
(car '((a) b c d))                  ⇒  (a)
(car '(1 . 2))                      ⇒  1
```

(cdr *cons*) → <*object*> **function**

The function **cdr** returns the right component of the *cons*. An error shall be signaled if *cons* is not a cons (error-id. *domain-error*).

Example:

```
(cdr '())                     an error shall be signaled
(cdr '((a) b c d))                  ⇒  (b c d)
(cdr '(1 . 2))                      ⇒  2
```

(setf (car *cons*) *obj*) → <*object*> **special form**
(set-car *obj* *cons*) → <*object*> **function**

Updates the left component of *cons* with *obj*. The returned value is *obj*. An error shall be signaled if *cons* is not a cons (error-id. *domain-error*). *obj* may be any ISLISP object.

Example:

```
(let ((x (list 'apple 'orange)))
  (list x (car x)
        (setf (car x) 'banana)
        x (car x)))
  ⇒ ((banana orange) apple banana (banana orange) banana)
```

(setf (cdr *cons*) *obj*) → <*object*> **special form**
(set-cdr *obj* *cons*) → <*object*> **function**

Updates the right component of *cons* with *obj*. The returned value is *obj*. An error shall be signaled if *cons* is not a cons (error-id. *domain-error*). *obj* may be any ISLISP object.

Example:

```
(let ((x (list 'apple 'orange)))
  (list x (cdr x)
        (setf (cdr x) 'banana)
        x (cdr x)))
  ⇒ ((apple . banana) (orange) banana (apple . banana) banana)
```

21.2 Null class

This class consists of only one element, the object called `nil`. This object is the false value in boolean expressions. The length of the sequence `nil` is 0.

`(null` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is `nil`; otherwise, returns `nil`.[5] *obj* may be any ISLISP object.

Example:

```
(null '(a b c))                  ⇒  nil
(null '())                       ⇒  t
(null (list))                    ⇒  t
```

21.3 List operations

`(listp` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is a list (instance of class `<list>`); otherwise, returns `nil`. *obj* may be any ISLISP object.

Example:

```
(listp '(a b c))                 ⇒  t
(listp '())                      ⇒  t
(listp '(a . b))                 ⇒  t
(let ((x (list 'a)))
  (setf (cdr x) x)
  (listp x))                     ⇒  t
(listp "abc")                    ⇒  nil
(listp #(1 2))                   ⇒  nil
(listp 'jerome)                  ⇒  nil
```

`(create-list` *i* [*initial-element*]`)` → *<list>* **function**

Returns a list of length *i*. If *initial-element* is given, the elements of the new list are initialized with this object; otherwise, the initialization is implementation defined. An error shall be signaled if the requested list cannot be allocated (error-id. *cannot-create-list*). An error shall be

[5] If the naming conventions were strictly observed, `null` would be named `nullp`; it is named `null` for historical and compatibility reasons.

signaled if *i* is not a non-negative integer (error-id. *domain-error*). *initial-element* may be any ISLISP object.

Example:

```
(create-list 3 17)                  ⇒ (17 17 17)
(create-list 2 #\a)                 ⇒ (#\a #\a)
```

(list *obj**) → <*list*> **function**

Returns a new list whose length is the number of arguments and whose elements are the arguments in the same order as in the `list`-form. An error shall be signaled if the requested list cannot be allocated (error-id. *cannot-create-list*). Each *obj* may be any ISLISP object.

Example:

```
(list 'a (+ 3 4) 'c)                ⇒  (a 7 c)
(list)                              ⇒  nil
```

(reverse *list*) → <*list*> **function**
(nreverse *list*) → <*list*> **function**

These functions each return a list whose elements are those of the given *list*, but in reverse order. An error shall be signaled if *list* is not a list (error-id. *domain-error*).

For `reverse`, no side-effect to the given *list* occurs. The resulting list is permitted but not required to share structure with the input *list*.

For `nreverse`, the conses which make up the top level of the given *list* are permitted, but not required, to be side-effected in order to produce this new list. `nreverse` should never be called on a literal object.

Example:

```
(reverse '(a b c d e))              ⇒ (e d c b a)
(reverse '(a))                      ⇒ (a)
(reverse '())                       ⇒ ()
(let* ((x (list 'a 'b)) (y (nreverse x))) (equal x y))
                                    ⇒  implementation-defined
```

(append *list**) → <*list*> **function**

Returns the result of appending all of the *lists*, or `()` if given no *lists*. An error shall be signaled if any *list* is not a list (error-id. *domain-error*).

This function does not modify its arguments. It is implementation defined whether and when the result shares structure with its *list* arguments.

An error shall be signaled if the list cannot be allocated (error-id. *cannot-create-list*).

Example:

```
(append '(a b c) '(d e f))          ⇒ (a b c d e f)
```

(member *obj* *list*) → <*list*> **function**

If *list* contains at least one occurrence of *obj* (as determined by `eql`), the first sublist of *list* whose car is *obj* is returned. Otherwise, `nil` is returned. An error shall be signaled if *list* is not a list (error-id. *domain-error*).

Example:

```
(member 'c '(a b c d e f))          ⇒ (c d e f)
(member 'g '(a b c d e f))          ⇒ nil
(member 'c '(a b c a b c))          ⇒ (c a b c)
```

(mapcar *function* $list^+$) → <*list*> **function**
(mapc *function* $list^+$) → <*list*> **function**
(mapcan *function* $list^+$) → <*list*> **function**
(maplist *function* $list^+$) → <*list*> **function**
(mapl *function* $list^+$) → <*list*> **function**
(mapcon *function* $list^+$) → <*list*> **function**

Successively applies the given *function* to sets of arguments determined by the given *lists*. The way in which the arguments are determined, and the way in which the result is accumulated are how these functions differ.

Function	Argument	Result
`mapcar`	successive elements	successive `cons`
`mapc`	successive elements	none (*i.e.,* $list_1$ returned)
`mapcan`	successive elements	successive “destructive append”
`maplist`	successive sublists	successive `cons`
`mapl`	successive sublists	none (*i.e.,* $list_1$ returned)
`mapcon`	successive sublists	successive “destructive append”

`mapcar` operates on successive elements of the *lists*. *function* is applied to the first element of each *list*, then to the second element of each *list*, and so on. The iteration terminates when the shortest *list* runs out, and excess elements in other lists are ignored. The value returned by `mapcar` is a list of the results of successive calls to *function*.

`mapc` is like `mapcar` except that the results of applying *function* are not accumulated; $list_1$ is returned.

`maplist` is like `mapcar` except that *function* is applied to successive sublists of the *lists*. *function* is first applied to the *lists* themselves, and then to the cdr of each *list*, and then to the cdr of the cdr of each *list*, and so on.

`mapl` is like `maplist` except that the results of applying *function* are not accumulated; $list_1$ is returned.

`mapcan` and `mapcon` are like `mapcar` and `maplist` respectively, except that the results of applying *function* are combined into a list by the use of an operation that performs a destructive form of `append` rather than `list`.

An error shall be signaled if *function* is not a function (error-id. *domain-error*). An error shall be signaled if any *list* is not a list (error-id. *domain-error*).

In all cases, the calls to *function* proceed from left to right, so that if *function* has side-effects, it can rely upon being called first on all of the elements with index 0, then on all of those numbered 1, and so on.

Example:

```
(mapcar #'car '((1 a) (2 b) (3 c))) ⇒ (1 2 3)
(mapcar #'abs '(3 -4 2 -5 -6))      ⇒ (3 4 2 5 6)
(mapcar #'cons '(a b c) '(1 2 3))   ⇒ ((a . 1) (b . 2) (c . 3))

(let ((x 0)) (mapc (lambda (v) (setq x (+ x v))) '(3 5)) x)

          ⇒ 8

(maplist #'append '(1 2 3 4) '(1 2) '(1 2 3))

          ⇒ ((1 2 3 4 1 2 1 2 3) (2 3 4 2 2 3))
(maplist (lambda (x) (cons 'foo x)) '(a b c d))

          ⇒ ((foo a b c d) (foo b c d) (foo c d) (foo d))
(maplist (lambda (x) (if (member (car x) (cdr x)) 0 1))
         '(a b a c d b c))

          ⇒ (0 0 1 0 1 1 1)

(let ((k 0))
  (mapl (lambda (x)
          (setq k (+ k (if (member (car x) (cdr x)) 0 1))))
        '(a b a c d b c))
  k)

          ⇒ 4

(mapcan (lambda (x) (if (> x 0) (list x))) '(-3 4 0 5 -2 7))
```

```
          ⇒ (4 5 7)

(mapcon (lambda (x) (if (member (car x) (cdr x)) (list (car x))))
        '(a b a c d b c b c))

          ⇒ (a b c b c)

(mapcon #'list '(1 2 3 4))          ⇒ ((1 2 3 4) (2 3 4) (3 4) (4))
```

(assoc *obj* *association-list*) → <*list*> **function**

If *assocation-list* contains at least one *cons* whose car is *obj* (as determined by `eql`), the first such *cons* is returned. Otherwise, `nil` is returned. An error shall be signaled if *association-list* is not a list of conses (error-id. *domain-error*).

Example:

```
(assoc 'a '((a . 1) (b . 2)))       ⇒ (a . 1)
(assoc 'a '((a . 1) (a . 2)))       ⇒ (a . 1)
(assoc 'c '((a . 1) (b . 2)))       ⇒ nil
```

22 Arrays

22.1 Array classes

Arrays store data in array components, which are indexed by a tuple of non-negative integers called *indices*.

The total number of elements in the array is the product of the dimensions. Zero-dimensional arrays are permissible and, as a consequence of this rule, can store exactly one element, indexed by an empty tuple of indices.

There are several array classes. For a pictorial representation of their inheritance relationship, see Figure 1. The following is an explanation of the purpose of each of these classes:

- `<basic-array>`

  All arrays are of the abstract class `<basic-array>`, but (as with all abstract classes) there are no direct instances of this class. It is provided for type discrimination purposes only.

  ISLisp defines two direct subclasses of `<basic-array>`: `<basic-vector>` and `<basic-array*>`. These classes are mutually exclusive and form an exhaustive partition of the set of basic-arrays. There shall be no other direct subclasses of `<basic-array>`.

- <basic-vector>

  All one-dimensional arrays are of the abstract class <basic-vector>, but (as with all abstract classes) there are no direct instances of this class. It is provided for type discrimination purposes only.

  ISLISP defines only two direct subclasses of <basic-vector>: <general-vector> and <string>. There may be additional, implementation-defined subclasses of <basic-vector>.

  Note: An implementation might provide specialized array representations for one-dimensional arrays of bits. If provided, such an array representation would be subclass of <basic-vector>.

- <general-vector>

  An object of class <general-vector> is a one-dimensional array that is capable of holding elements of type <object>. When the function create-array is asked to create a one-dimensional array, the resulting array is of this class.

- <string>

  An object of class <string> is a one-dimensional array that is capable only of holding elements of type <character>. When the function create-string is used, the result is of this class.

- <basic-array*>

  All non-one-dimensional arrays are of the abstract class <basic-array*>, but (as with all abstract classes) there are no direct instances of this class. It is provided for type discrimination purposes only.

  ISLISP defines only one direct subclass of <basic-array*>: <general-array*>. There may be additional, implementation-defined subclasses of <basic-array*>.

  Note: An implementation might provide specialized array representations for two-dimensional arrays of 1 or more bits to hold display information for a monochrome or color screen. If provided, such array representations would be subclasses of <basic-array*>.

- <general-array*>

  An object of class <general-array*> is a non-one-dimensional array that is capable of holding elements of type <object>. When the function create-array is asked to create an array of dimensionality other than 1, the resulting array is of this class.

22.2 General arrays

An object that is either of class <general-vector> or of class <general-array*> is sometimes called a "general array."

General arrays are capable of storing any object of class <object>. Those arrays that are not general arrays are the ones restricted to storage objects of more specialized classes.

A general array can be expressed as a textual literal using #*n*a notation (where n is an integer indicating the number of dimensions of the array) followed by a nested list of sequences denoting the contents of the array. This structure can be defined as follows. If $n = 1$ the structure is simply (obj_1 ... obj_n). If $n > 1$ and the dimensions are (n_1 n_2 ...), the structure is (str_1 ... str_{n_1}), where the str_i are the structures of the n_1 subarrays, each of which has dimensions (n_2 ...). For example, the textual representation of (create-array '(2 3 4) 5) is as follows: #3a(((5 5 5 5) (5 5 5 5) (5 5 5 5)) ((5 5 5 5) (5 5 5 5) (5 5 5 5))).

22.3 Array operations

To manipulate arrays ISLISP provides the following functions.

(basic-array-p *obj*) → *boolean* **function**
(basic-array*-p *obj*) → *boolean* **function**
(general-array*-p *obj*) → *boolean* **function**

---

basic-array-p returns t if *obj* is a basic-array (instance of class <basic-array>); otherwise, returns nil. *obj* may be any ISLISP object.

basic-array*-p returns t if *obj* is a basic-array* (instance of class <basic-array*>); otherwise, returns nil. *obj* may be any ISLISP object.

general-array*-p returns t if *obj* is a general-array* (instance of class <general-array*>); otherwise, returns nil. *obj* may be any ISLISP object.

Example:

```
(mapcar (lambda (x)
          (list (basic-array-p x)
                (basic-array*-p x)
                (general-array*-p x)))
        '((a b c)
          "abc"
          #(a b c)
          #1a(a b c)
          #2a((a) (b) (c))))
 ⇒ ((nil nil nil) (t nil nil) (t nil nil) (t nil nil) (t t t))
```

(create-array *dimensions* [*initial-element*]) → <*basic-array*> **function**

This function creates an array of the given *dimensions*. The *dimensions* argument is a list of non-negative integers.

The result is of class <general-vector> if there is only one dimension, or of class <general-array*> otherwise.

If *initial-element* is given, the elements of the new array are initialized with this object, otherwise the initialization is implementation defined.

An error shall be signaled if the requested array cannot be allocated (error-id. *cannot-create-array*).

An error shall be signaled if *dimensions* is not a proper list of non-negative integers (error-id. *domain-error*). *initial-element* may be any ISLISP object.

Example:

```
(create-array '(2 3) 0.0)          ⇒ #2a((0.0 0.0 0.0) (0.0 0.0 0.0))
(create-array '(2) 0.0)            ⇒ #(0.0 0.0)
```

(aref *basic-array* *z**) → <*object*> **function**
(garef *general-array* *z**) → <*object*> **function**

---

aref returns the object stored in the component of the *basic-array* specified by the sequence of integers *z*. This sequence must have exactly as many elements as there are dimensions in the *basic-array*, and each one must satisfy $0 \leq z_i < d_i$, d_i the i^{th} dimension and $0 \leq i < d$, d the number of dimensions. Arrays are indexed 0 based, so the ith row is accessed via the index $i - 1$.

An error shall be signaled if *basic-array* is not a basic-array (error-id. *domain-error*). An error shall be signaled if any *z* is not a non-negative integer (error-id. *domain-error*).

garef is like aref but an error shall be signaled if its first argument, *general-array*, is not an object of class <general-vector> or of class <general-array*> (error-id. *domain-error*).

Example:

```
(defglobal array1 (create-array '(3 3 3) 0))
  ⇒ array1

array1
  ⇒ #3a(((0 0 0) (0 0 0) (0 0 0))
        ((0 0 0) (0 0 0) (0 0 0))
        ((0 0 0) (0 0 0) (0 0 0)))

(aref array1 0 1 2)                ⇒ 0
(setf (aref array1 0 1 2) 3.14)    ⇒ 3.14
(aref array1 0 1 2)                ⇒ 3.14

(aref (create-array '(8 8) 6) 1 1) ⇒ 6
(aref (create-array '() 19))       ⇒ 19
```

(setf (aref *basic-array* *z**) *obj*) → <*object*> **special form**
(set-aref *obj* *basic-array* *z**) → <*object*> **function**
(setf (garef *general-array* *z**) *obj*) → <*object*> **special form**
(set-garef *obj* *general-array* *z**) → <*object*> **function**

These replace the object obtainable by aref or garef with *obj*. The returned value is *obj*. The constraints on the *basic-array*, the *general-array*, and the sequence of indices *z* is the same as for aref and garef.

Example:

```
(setf (aref array1 0 1 2) 3.15)    ⇒ 3.15
```

```
(set-aref 51.3 array1 0 1 2)        ⇒ 51.3
```

(array-dimensions *basic-array*) → *<list>* **function**

Returns a list of the dimensions of a given *basic-array*. An error shall be signaled if *basic-array* is not a basic-array (error-id. *domain-error*). The consequences are undefined if the returned list is modified.

Example:

```
(array-dimensions
  (create-array '(2 2) 0))          ⇒ (2 2)
(array-dimensions (vector 'a 'b))   ⇒ (2)
(array-dimensions "foo")            ⇒ (3)
```

23 Vectors

A vector is a one dimensional array. See §22.1 for detailed information about the relationship of arrays and vectors.

General vectors are written as follows:

#(x_1 x_2 ... x_n)

(basic-vector-p *obj*) → *boolean* **function**
(general-vector-p *obj*) → *boolean* **function**

---

basic-vector-p returns t if *obj* is a basic-vector (instance of class <basic-vector>); otherwise, returns nil. *obj* may be any ISLISP object.

general-vector-p returns t if *obj* is a general-vector (instance of class <general-vector>); otherwise, returns nil. *obj* may be any ISLISP object.

Example:

```
(mapcar (lambda (x)
          (list (basic-vector-p x)
                (general-vector-p x)))
        '((a b c)
          "abc"
          #(a b c)
          #1a(a b c)
```

```
           #2a((a) (b) (c))))
  ⇒ ((nil nil) (t nil) (t t) (t t) (nil nil))
```

(create-vector *i* [*initial-element*]) → <*general-vector*> **function**

Returns a general-vector of length *i*. If *initial-element* is given, the elements of the new vector are initialized with this object, otherwise the initialization is implementation defined. An error shall be signaled if the requested vector cannot be allocated (error-id. *cannot-create-vector*). An error shall be signaled if *i* is not a non-negative integer (error-id. *domain-error*). *initial-element* may be any ISLisp object.

Example:

```
(create-vector 3 17)            ⇒ #(17 17 17)
(create-vector 2 #\a)           ⇒ #(#\a #\a)
```

(vector *obj**) → <*general-vector*> **function**

Returns a new general-vector whose elements are its *obj* arguments. The length of the newly created vector is, therefore, the number of *objs* passed as arguments. The vector is indexed by integers ranging from 0 to *dimension*-1. An error shall be signaled if the requested vector cannot be allocated (error-id. *cannot-create-vector*). Each *obj* may be any ISLisp object.

Example:

```
(vector 'a 'b 'c)               ⇒  #(a b c)
(vector)                        ⇒  #()
```

24 String class

A string is a vector that is capable only of holding elements of type `<character>`. See §22.1 for detailed information about the relationship of arrays, vectors, and strings.

Any implementation-defined character can be a string element. In ISLisp, string indices are 0-based. Strings are written by listing all the element characters in order and by enclosing them with double quotes "`"`". If the string has a double quote as its element, the double quote must be preceded by a backslash "\". If the string has a backslash as its element, the backslash must be preceded by another backslash. Strings contained in program text as literals are immutable objects. The representation of non-printing characters is implementation defined.

(stringp *obj*) → *boolean* **function**

Returns `t` if *obj* is a string (instance of class `<string>`); otherwise, returns `nil`. *obj* may be any ISLISP object.

Example:

```
(stringp "abc")                    ⇒ t
(stringp 'abc)                     ⇒ nil
```

(`create-string` *i* [*initial-character*]) → <*string*> **function**

Returns a string of length *i*. If *initial-character* is given, then the characters of the new string are initialized with this character, otherwise the initialization is implementation defined. An error shall be signaled if the requested string cannot be allocated (error-id. *cannot-create-string*). An error shall be signaled if *i* is not a non-negative integer or if *initial-character* is not a character (error-id. *domain-error*).

Example:

```
(create-string 3 #\a)              ⇒ "aaa"
(create-string 0 #\a)              ⇒ ""
```

(`string=` $string_1$ $string_2$) → *quasi-boolean* **function**
(`string/=` $string_1$ $string_2$) → *quasi-boolean* **function**
(`string<` $string_1$ $string_2$) → *quasi-boolean* **function**
(`string>` $string_1$ $string_2$) → *quasi-boolean* **function**
(`string>=` $string_1$ $string_2$) → *quasi-boolean* **function**
(`string<=` $string_1$ $string_2$) → *quasi-boolean* **function**

The function `string=` tests whether $string_1$ is the same string as $string_2$. The function `string<` tests whether $string_1$ is less than $string_2$. The function `string<=` tests whether $string_1$ is less than or equal to $string_2$.

The ordering used is based on character comparisons.

Two strings are `string=` if they are of the same length, l, and if for every i, where $0 \leq i < l$, `(char= (elt` $string_1$ i`) (elt` $string_2$ i`))` holds.

Two strings $string_1$ and $string_2$ are in order (`string<`) if in the first position in which they differ the character of $string_1$ is `char<` the corresponding character of $string_2$, or if the $string_1$ is a proper prefix of $string_2$ (of shorter length and matching in all the characters of $string_1$).

Two strings are `string<=` if they are either `string<` or they are `string=`.

Two strings are `string/=` if and only if they are not `string=`. Two strings are `string>` if and only if they are not `string<=`. Two strings are `string>=` if and only if they are not `string<`.

For these 6 string comparison functions, if the test is satisfied, an implementation-defined non-`nil` value is returned; otherwise, `nil` is returned.

An error shall be signaled if either *string*$_1$ or *string*$_2$ is not a string (error-id. *domain-error*).

Example:

```
(if (string= "abcd" "abcd") t nil)  ⇒ t
(if (string= "abcd" "wxyz") t nil)  ⇒ nil
(if (string= "abcd" "abcde") t nil) ⇒ nil
(if (string= "abcde" "abcd") t nil) ⇒ nil
(if (string/= "abcd" "wxyz") t nil) ⇒ t
(if (string< "abcd" "abcd") t nil)  ⇒ nil
(if (string< "abcd" "wxyz") t nil)  ⇒ t
(if (string< "abcd" "abcde") t nil) ⇒ t
(if (string< "abcde" "abcd") t nil) ⇒ nil
(if (string<= "abcd" "abcd") t nil) ⇒ t
(if (string<= "abcd" "wxyz") t nil) ⇒ t
(if (string<= "abcd" "abcde") t nil)⇒ t
(if (string<= "abcde" "abcd") t nil)⇒ nil
(if (string> "abcd" "wxyz") t nil)  ⇒ nil
(if (string>= "abcd" "abcd") t nil) ⇒ t
```

(`char-index` *char string* [*start-position*]) → <*object*> **function**

Returns the position of *char* in *string*, The search starts from the position indicated by *start-position* (which is 0-based and defaults to `0`). The value returned if the search succeeds is an offset from the beginning of the *string*, not from the starting point. If the *char* does not occur in the *string*, `nil` is returned. The function `char=` is used for the comparisons.

An error shall be signaled if *char* is not a character or if *string* is not a string (error-id. *domain-error*).

Example:

```
(char-index #\b "abcab")          ⇒ 1
(char-index #\B "abcab")          ⇒ nil
(char-index #\b "abcab" 2)        ⇒ 4
(char-index #\d "abcab")          ⇒ nil
(char-index #\a "abcab" 4)        ⇒ nil
```

(`string-index` *substring string* [*start-position*]) → <*object*> **function**

Returns the position of the given *substring* within *string*. The search starts from the position indicated by *start-position* (which is 0-based and defaults to `0`). The value returned if the search succeeds is an offset from the beginning of the *string*, not from the starting point. If that *substring* does not occur in the *string*, `nil` is returned. Presence of the substring is done by sequential use of `char=` on corresponding elements of the two strings.

An error shall be signaled if either *substring* or *string* is not a string (error-id. *domain-error*).

Example:

```
(string-index "foo" "foobar")       ⇒ 0
(string-index "bar" "foobar")       ⇒ 3
(string-index "FOO" "foobar")       ⇒ nil
(string-index "foo" "foobar" 1)     ⇒ nil
(string-index "bar" "foobar" 1)     ⇒ 3
(string-index "foo" "")             ⇒ nil
(string-index "" "foo")             ⇒ 0
```

(`string-append` *string**) → <*string*> **function**

Returns a single string containing a sequence of characters that results from appending the sequences of characters of each of the *strings*, or `""` if given no *strings*. An error shall be signaled if any *string* is not a string (error-id. *domain-error*).

This function does not modify its arguments. It is implementation defined whether and when the result shares structure with its *string* arguments.

An error shall be signaled if the string cannot be allocated (error-id. *cannot-create-string*).

Example:

```
(string-append "abc" "def")         ⇒ "abcdef"
(string-append "abc" "abc")         ⇒ "abcabc"
(string-append "abc" "")            ⇒ "abc"
(string-append "" "abc")            ⇒ "abc"
(string-append "abc" "" "def")      ⇒ "abcdef"
```

25 Sequence functions

Objects that are either of class `<basic-vector>` or of class `<list>` are sometimes called "sequences". The operations upon sequences are called "sequence functions."

(`length` *sequence*) → <*integer*> **function**

Returns the length of *sequence* as an integer greater than or equal to 0.

When *sequence* is a basic-vector, `length` returns its dimension.

When *sequence* is a list, the result is the number of elements in the list; if an element is itself a list, the elements within this sublist are not counted. In the case of dotted lists, `length` returns the number of conses at the uppermost level of the list. For example, `(length '(a b . c))` ⇒ 2, since `'(a b . c)` ≡ `(cons 'a (cons 'b 'c))`.

An error shall be signaled if *sequence* is not a basic-vector or a list (error-id. *domain-error*).

Example:

```
(length '(a b c))                ⇒ 3
(length '(a (b) (c d e)))        ⇒ 3
(length '())                     ⇒ 0
(length (vector 'a 'b 'c))       ⇒ 3
```

`(elt` *sequence z*`)` → <*object*> **function**

Given a *sequence* and an integer z satisfying $0 \leq z <$ `(length` *sequence*`)`, `elt` returns the element of *sequence* that has index z. Indexing is 0-based; *i.e.,* $z = 0$ designates the first element. An error shall be signaled if z is an integer outside of the mentioned range (error-id. *index-out-of-range*).

An error shall be signaled if *sequence* is not a basic-vector or a list or if z is not an integer (error-id. *domain-error*).

Example:

```
(elt '(a b c) 2)                 ⇒ c
(elt (vector 'a 'b 'c) 1)        ⇒ b
(elt "abc" 0)                    ⇒ #\a
```

`(setf (elt` *sequence z*`)` *obj*`)` → <*object*> **special form**
`(set-elt` *obj sequence z*`)` → <*object*> **function**

These replace the object obtainable by `elt` with *obj*. The returned value is *obj*.

An error shall be signaled if z is an integer outside of the valid range of indices (error-id. *index-out-of-range*). An error shall be signaled if *sequence* is not a basic-vector or a list or if z is not an integer (error-id. *domain-error*). *obj* may be any ISLISP object.

Example:

```
(let ((string (create-string 5 #\x)))
```

```
(setf (elt string 2) #\0)
x)                                  ⇒ "xx0xx"
```

(subseq *sequence* z_1 z_2) → *sequence* **function**

Given a sequence *sequence* and two integers z_1 and z_2 satisfying
$0 \leq z_1 \leq z_2 \leq$ (length *sequence*), this function returns the subsequence of length $z_2 - z_1$, containing the elements with indices from z_1 (inclusive) to z_2 (exclusive). The subsequence is newly allocated, and has the same class as *sequence*.

An error shall be signaled if the requested subsequence cannot be allocated (error-id. *cannot-create-sequence*). An error shall be signaled if z_1 or z_2 are outside of the bounds mentioned (error-id. *index-out-of-range*). An error shall be signaled if *sequence* is not a basic-vector or a list, or if z_1 is not an integer, or if z_2 is not an integer (error-id. *domain-error*).

Example:

```
(subseq "abcdef" 1 4)               ⇒ "bcd"
(subseq '(a b c d e f) 1 4)         ⇒ (b c d)
(subseq (vector 'a 'b 'c 'd 'e 'f) 1 4)
  ⇒ #(b c d)
```

(map-into *destination* *function* *sequence**) → *sequence* **function**

Destructively modifies *destination* to contain the results of applying *function* to successive elements in the *sequences*. The *destination* is returned.

If *destination* and each element of *sequences* are not all the same length, the iteration terminates when the shortest sequence (of any of the *sequences* or the *destination*) is exhausted.

The calls to *function* proceed from left to right, so that if *function* has side-effects, it can rely upon being called first on all of the elements with index 0, then on all of those numbered 1, and so on.

An error shall be signaled if *destination* is not a basic-vector or a list (error-id. *domain-error*). An error shall be signaled if any *sequence* is not a basic-vector or a list (error-id. *domain-error*).

Example:

```
(setq a (list 1 2 3 4))             ⇒ (1 2 3 4)
(setq b (list 10 10 10 10))         ⇒ (10 10 10 10)
(map-into a #'+ a b)                ⇒ (11 12 13 14)
a                                   ⇒ (11 12 13 14)
b                                   ⇒ (10 10 10 10)
(setq k '(one two three))           ⇒ (one two three)
```

```
(map-into a #'cons k a)          ⇒ ((one . 11) (two . 12) (three . 13) 14)
(let ((x 0))
  (map-into a
    (lambda () (setq x (+ x 2)))))

         ⇒ (2 4 6 8)
a                                ⇒ (2 4 6 8)
```

26 Stream class

Streams are instances of the <stream> class. They are objects that serve as sources or sinks of data.

`(streamp` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is a stream (instance of class <stream>); otherwise, returns `nil`. *obj* may be any ISLISP object. `streamp` is unaffected by whether its argument, if an instance of the class <stream>, is open or closed.

Example:

```
(streamp (standard-input))       ⇒ t
(streamp '())                    ⇒ nil
```

`(open-stream-p` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is an open stream; otherwise, returns `nil`.

`(input-stream-p` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is a stream that can handle input operations; otherwise, returns `nil`.

Example:

```
(input-stream-p (standard-input))   ⇒ t
(input-stream-p (standard-output))  ⇒ nil
(input-stream-p '(a b c))           ⇒ nil
```

`(output-stream-p` *obj*`)` → *boolean* **function**

Returns `t` if *obj* is a stream that can handle output operations; otherwise, returns `nil`.

Example:

```
(output-stream-p (standard-output)) ⇒ t
(output-stream-p (standard-input))  ⇒ nil
(output-stream-p "hello")           ⇒ nil
```

`(standard-input)` → <*stream*> **function**
`(standard-output)` → <*stream*> **function**
`(error-output)` → <*stream*> **function**

The function `standard-input` returns the stream used as the default stream for input functions.

The function `standard-output` returns the stream used as the default stream for output functions.

The function `error-output` returns the stream used as the default stream for warnings and non-interactive error messages.

The value returned by each of these functions is initially implementation-defined, but can be dynamically bound; see `with-standard-input`, `with-standard-output`, and `with-error-output`.

`(with-standard-input` *stream-form form** `)` → <*object*> **special operator**
`(with-standard-output` *stream-form form** `)` → <*object*> **special operator**
`(with-error-output` *stream-form form** `)` → <*object*> **special operator**

These special forms first evaluate their *stream-form* argument to produce a stream *s* and then evaluate their body *forms* in a dynamic environment where the corresponding function (`standard-input`, `standard-output`, or `error-output`) returns the stream *s*. The returned value of each of these forms is the result of the evaluation of the last *form* of their body (or `nil` if there is none).

Example:

```
(with-standard-input (create-string-input-stream "this is a string")
  (list (read) (read)))
 ⇒ (this is)
```

26.1 Streams to files

Streams might be connected to files or devices. Given a name, a stream can be created that is connected to a file having that name. File systems in which files are not named are not supported.

A ***filename*** is represented by a string. The correct syntax of filenames is implementation defined.

Streams to files are created by `open-input-file`, `open-output-file`, `open-io-file`, `with-open-input-file`, `with-open-output-file`, and `with-open-io-file`.

`(open-input-file` *filename* [*element-class*]`)` → <*stream*> **function**
`(open-output-file` *filename* [*element-class*]`)` → <*stream*> **function**
`(open-io-file` *filename* [*element-class*]`)` → <*stream*> **function**

`open-input-file` opens a file for input only. `open-output-file` opens a file for output only. `open-io-file` opens a file for both input and output.

An error shall be signaled if *filename* is not a string. The corresponding file is opened in an implementation-defined way. These functions return an instance of the `<stream>` class connected to the file specified by *filename*.

The *element-class* can be either the class `<character>` (the default) or a positive integer that is a number of bits in a byte to be used for a binary stream. All implementations must support a value of 8 (denoting integer byte values from 0 to 255), but some implementations might support other byte sizes as well.

Note: When a binary stream is used, the order of bits within bytes and bytes within words (*i.e.,* whether it is right to left or left to right) are each implementation defined.

Example:

```
(open-input-file "example.lsp" 8)
  ⇒ implementation-defined
```

`(with-open-input-file (`<u>*name*</u> *filename* [*element-class*]`)` *form**`)` → <*object*>
special operator
`(with-open-output-file (`<u>*name*</u> *filename* [*element-class*]`)` *form**`)` → <*object*>
special operator
`(with-open-io-file (`<u>*name*</u> *filename* [*element-class*]`)` *form**`)` → <*object*>
special operator

Each of these special forms opens a stream to a file (using `open-input-file`, `open-output-file`, or `open-io-file`, respectively), evaluates the *forms*, closes the file, and returns the value returned by the last *form* (or `nil` if there are no *forms*).

The *filename* and *element-class* are evaluated and passed as arguments to the appropriate file-opening function. The stream created by opening the file is bound to the variable named by *name* (as if a `let` was used), so the identifier *name* can be used to refer to the stream.

The stream is closed on exit, whether or not exit from these special forms is normal. For this reason, these special forms are usually preferred over the corresponding functions for opening and closing files.

Example:

```
(with-open-output-file (outstream "example.dat")
    (format outstream "hello"))
                                    ⇒ nil

(with-open-input-file (instream "example.dat")
    (read instream))
                                    ⇒ hello
```

`(close` *stream*`)` → *implementation defined* **function**

The function `close` closes the stream *stream*. If *stream* is closed it may no longer be used in input or output operations. *Closing* a *file stream* ends the association between the *stream* and its *file*. If the *stream* was already closed this function performs nothing. The result value is implementation defined. An error shall be signaled if *stream* is not a stream (error-id. *domain-error*).

Example:

```
(defglobal input-str (open-input-file "data.lsp"))
  ⇒ input-str

(close input-str)                   ⇒ implementation-defined
(close input-str)                   ⇒ implementation-defined
```

`(finish-output` *stream*`)` → *<null>* **function**

Completes any pending output to the destination designated by *stream*. Waits until the pending output is complete and then returns `nil`. For instance, pending output might be stored in a buffer; in this case `finish-output` forces the buffer to be written to the stream's destination. An error shall be signaled if *stream* is not a stream that can handle output operations (error-id. *domain-error*).

Example:

```
(defglobal output-str (open-output-file "data.lsp"))
  ⇒ output-str

(finish-output output-str)          ⇒ nil
```

26.2 Other streams

Non-file streams can be created by the following functions:

`create-string-input-stream` `create-string-output-stream`

A string stream is a stream that is simply a string. For input, the reading functions construct objects from a character sequence obtained from an input string. For output, the printing functions deliver characters which are collected to a result string.

(`create-string-input-stream` *string*) → <*stream*> **function**

Creates and returns an input stream from the *string*. An error shall be signaled if *string* is not a string (error-id. *domain-error*).

Example:

```
(let ((str (create-string-input-stream "this is a string")))
  (list (read str) (read str) (read str)))
  ⇒ (this is a)
```

(`create-string-output-stream`) → <*stream*> **function**

This function creates and returns a string output stream. The output to a string stream can be retrieved by `get-output-stream-string`.

Example:

```
(let ((str (create-string-output-stream)))
  (format str "hello")
  (format str "world")
  (get-output-stream-string str))
  ⇒ "helloworld"
```

(`get-output-stream-string` *stream*) → <*string*> **function**

Returns a string containing all characters written to *stream* since the last call to this function or since the creation of the stream, if this function has not been called with *stream* before. An error shall be signaled if *stream* is not a stream created with `create-string-output-stream` (error-id. *domain-error*).

Example:

```
(let ((out-str (create-string-output-stream)))
  (format out-str "This is a string")
  (let ((part1 (get-output-stream-string out-str)))
```

```
  (format out-str "right!")
  (list part1 (get-output-stream-string out-str))))
⇒ ("This is a string" "right!")
```

27 Input and output

27.1 Argument conventions for input functions

Most of the reader functions that do input treat their arguments as follows:

When end-of-stream is reached (*i.e.,* an attempt is made to read a stream element immediately after the last one in the stream), the behavior depends on the value of *eos-error-p* (which defaults to `t`): if *eos-error-p* is `nil`, the function returns the *eos-value* (which defaults to `nil`); otherwise, an error shall be signaled (error-id. *end-of-stream*).

If the *input-stream* is not specified, the standard input stream (the value returned by the `standard-input` function) is used. An error shall be signaled if an *input-stream* does not satisfy the `input-stream-p` predicate (error-id. *not-an-input-stream*).

27.2 Character I/O

The following operations are used for character I/O. An error shall be signaled if an attempt is made to perform a character I/O operation on a stream that does not handle such operations.

(`read` [*input-stream* [*eos-error-p* [*eos-value*]]]) → <*object*> **function**

The function `read` returns the ISLISP object that is created as the result of reading its textual representation from the stream *input-stream*.

See §27.1 for information about how *input-stream*, *eos-error-p*, and *eos-value* are treated.

Example:

```
(defglobal str (create-string-input-stream "hello #(1 2 3) 123 #\\A"))
  ⇒ str

(read str)                         ⇒ hello
(read str)                         ⇒ #(1 2 3)
(read str)                         ⇒ 123
(read str)                         ⇒ #\A
(read str nil "the end")           ⇒ "the end"
```

(`read-char` [*input-stream* [*eos-error-p* [*eos-value*]]]) → <*object*> **function**

`read-char` reads a single character from *input-stream* and returns the corresponding character object.

See §27.1 for information about how *input-stream*, *eos-error-p*, and *eos-value* are treated.

Example:

```
(defglobal str (create-string-input-stream "hi"))
  ⇒ str

(read-char str)                      ⇒ #\h
(read-char str)                      ⇒ #\i
(read-char str)  an error shall be signaled
```

(`preview-char` [*input-stream* [*eos-error-p* [*eos-value*]]]) → <*object*> **function**

Returns the next character of *input-stream*, if any. The character is not consumed; the next attempt to peek at or read a character from the stream sees that same character.

See §27.1 for information about how *input-stream*, *eos-error-p*, and *eos-value* are treated.

Example:

```
(let ((s (create-string-input-stream "foo")))
  (list (preview-char s) (read-char s) (read-char s)))
 ⇒ (#\f #\f #\o)
```

(`read-line` [*input-stream* [*eos-error-p* [*eos-value*]]]) → <*object*> **function**

Reads a line of characters from *input-stream* and returns them as a string (without the newline character at the end of the line). If an end-of-stream is reached before the next newline character and a non-empty line has been read prior to the end-of-stream, that line is returned.

See §27.1 for information about how *input-stream*, *eos-error-p*, and *eos-value* are treated.

Example:

```
(with-open-output-file (out "newfile")
  (format out "This is an example")
  (format out "~%")
  (format out "look at the output file"))
```

```
  ⇒ nil

(defglobal str (open-input-file "newfile"))
  ⇒ str

(read-line str)                    ⇒ "This is an example"
(read-line str)                    ⇒ "look at the output file"
```

(stream-ready-p *input-stream*) → *boolean* **function**

Returns t if an attempt to obtain the next element from the *stream* will not cause the processor to have to wait; otherwise, returns nil. An error shall be signaled if *stream* is not a stream that can handle input operations (error-id. *domain-error*).

Example:

```
(with-open-output-file (out "testfile.dat")
  (format out "This is an example"))⇒ nil

(with-open-input-file (in "testfile.dat")
  (stream-ready-p in))             ⇒ t
```

(format *output-stream format-string obj**) → *<null>* **function**
(format-char *output-stream char*) → *<null>* **function**
(format-float *output-stream float*) → *<null>* **function**
(format-fresh-line *output-stream*) → *<null>* **function**
(format-integer *output-stream integer radix*) → *<null>* **function**
(format-object *output-stream obj escape-p*) → *<null>* **function**
(format-tab *output-stream column*) → *<null>* **function**

The function format has the side-effect of printing according to *format-string*. It returns nil. An error shall be signaled if the *output-stream* parameter does not satisfy the output-stream-p predicate (error-id. *not-an-output-stream*). An error shall be signaled if *format-string* is not a string (error-id. *domain-error*). The following is a summary of all the available format directives:

obj refers to the next item of the set of *obj** to be processed.

~A Aesthetic: The *obj* is any object. *obj* is printed as it would with ~S, but without escape characters. Characters are output directly without any conversion. That is, the output generated using this format directive is suitable for being read by a human reader.

This effect is implemented by (format-object *output-stream obj* nil).

~B Binary: An error shall be signaled if *obj* is not an integer. *obj* is printed in binary radix (radix 2).

This effect is implemented by `(format-integer` *output-stream obj* `2)`.

`~C` Character: An error shall be signaled if *obj* is not a character. *obj* is output directly without any conversion.

This effect is implemented by `(format-char` *output-stream obj*`)`.

`~D` Decimal: An error shall be signaled if *obj* is not an integer. *obj* is printed in decimal radix (radix 10).

This effect is implemented by `(format-integer` *output-stream obj* `10)`.

`~G` General floating point: An error shall be signaled if *obj* is not a number. *obj* is printed as a float.

This effect is implemented by `(format-float` *output-stream obj*`)`.

`~O` Octal: An error shall be signaled if *obj* is not an integer. *obj* is printed in octal radix (radix 8).

This effect is implemented by `(format-integer` *output-stream obj* `8)`.

`~`n`R` Radix: An error shall be signaled if *obj* is not an integer. *obj* is printed in radix n (which must be between 2 and 36, inclusive).

This effect is implemented by `(format-integer` *output-stream obj n*`)`.

`~S` S-expression: *obj* is any object. This format directive outputs the textual representation of *obj*, with escape characters as needed. That is, the output generated using this format directive is suitable for input to the function `read`.

This effect is implemented by `(format-object` *output-stream obj* `t)`.

`~`n`T` Tab: output enough spaces to move to column n (where column 0 represents the left margin). If already at or beyond column n, one space is output. If an implementation cannot determine the current column position, the behavior is implementation defined, but at least one space will be output.

This effect is implemented by `(format-tab` *output-stream n*`)`.

`~X` Hexadecimal: An error shall be signaled if *obj* is not an integer. *obj* is printed in hexadecimal radix (radix 16).

This effect is implemented by `(format-integer` *output-stream obj* `16)`.

`~%` newline: output a `#\newline` character;

This effect is implemented by `(format-char` *output-stream* `#\newline)`.

`~&` conditional newline: output a `#\newline` character if it cannot be determined that the output stream is at the beginning of a fresh line;

This effect is implemented by `(format-fresh-line` *output-stream*`)`.

`~~` tilde: output a tilde (~).

This effect is implemented by `(format-char` *output-stream* `#\~)`.

Example:

```
(format output-stream "No result")  ⇒ nil
 Output is: No result

(format output-stream "The result is ~A and nothing else." "meningitis")
  ⇒ nil
 Output is: The result is meningitis and nothing else.

(format output-stream "The result i~C" #\s)
  ⇒ nil
 Output is: The result is

(format output-stream "The results are ~S and ~S." 1 #\a)
  ⇒ nil
 Output is: The results are 1 and #\a.

(format output-stream "Binary code ~B" 150)
  ⇒ nil
 Output is: Binary code 10010110

(format output-stream "permission ~O" 493)
  ⇒ nil
 Output is: permission 755

(format output-stream "You ~X ~X" 2989 64206)
  ⇒ nil
 Output is: You BAD FACE

(progn
  (format output-stream "~&Name ~10Tincome ~20Ttax~%")
  (format output-stream "~A ~10T~D ~20T~D" "Grummy" 23000 7500))
  ⇒ nil
 Output is: Name      income    tax
            Grummy    23000     7500

(format output-stream "This will be split into~%two lines.")
  ⇒ nil
 Output is: This will be split into
            two lines.

(format output-stream "This is a tilde: ~~")
  ⇒ nil
 Output is: This is a tilde: ~
```

27.3 Binary I/O

The following operations are used for binary I/O. An error shall be signaled if an attempt is made to perform a binary I/O operation on a stream that does not handle such operations.

(read-byte *input-stream* [*eos-error-p* [*eos-value*]]) → <*integer*> **function**

Reads a byte from the *input-stream* and returns it. The number of bits in a byte is determined by the stream element type of the *input-stream*; see `open-input-file`.

See §27.1 for information about how *input-stream*, *eos-error-p*, and *eos-value* are treated.

Example:

```
;; This example assumes 8-bit byte codes are stored in files.
(defglobal byte-example (open-output-stream "byte-ex"))
  ⇒ byte-example

(format byte-example "hello")       ⇒ nil
(close byte-example)                ⇒ implementation-defined
(setq byte-example (open-input-stream "byte-ex" 8))
  ⇒  implementation-defined

(read-byte  byte-example)           ⇒ 104  (implementation-defined)
(read-byte  byte-example)           ⇒ 101  (implementation-defined)
(read-byte  byte-example)           ⇒ 108  (implementation-defined)
(read-byte  byte-example)           ⇒ 108  (implementation-defined)
(read-byte  byte-example)           ⇒ 111  (implementation-defined)
```

(write-byte *z* *output-stream*) → <*integer*> **function**

Writes *z* to the *output-stream* and returns it. An error shall be signaled if *z* is not an integer in the range appropriate to the stream element type of *output-stream* or if *output-stream* is not a stream capable of handling output operations (error-id. *domain-error*).

Example:

```
(let ((out-str (open-output-stream "byte-example" 8)))
    (write-byte #b101 out-str)
    (close out-str))                ⇒ implementation-defined
```

28 Files

(probe-file *filename*) → *boolean* **function**

Returns `t` if the file specified by *filename* exists; otherwise, returns `nil`. An error shall be signaled if *filename* is not a string (error-id. *domain-error*).

Example:

```
(probe-file "notexist.lsp")         ⇒  nil
(defglobal new-file (open-output-file "notexist.lsp"))
  ⇒ new-file

(close new-file)                    ⇒ implementation-defined
(probe-file "notexist.lsp")         ⇒ t
```

(file-position *stream*) → <*integer*> **function**

Returns the ***file position*** associated with *stream*.

A file position is a non-negative integer that represents a position in the stream. For binary streams, the file position represents the number of preceding bytes in the stream. It is increased by one each time one of the following is done:

```
(read-byte stream)
(write-byte z stream)
```

For character streams, the file position is increased by an implementation-defined non-negative amount each time one of the following is done:

```
(format stream...)
(format-char stream char)
(format-float stream float)
(format-fresh-line stream)
(format-integer stream integer radix)
(format-object stream obj escape-p)
(format-tab stream column)
(report-condition condition stream)
(read-char stream)
(read-line stream)
(read stream)
```

The amount may depend on the output and on the file position itself. It is implementation-defined which integer represents the first element of the file. An error shall be signaled if *stream* is not a stream to or from a file (error-id. *domain-error*).

Example:

```
;; This example assumes 8-bit byte codes are stored in files.
(defglobal example (open-output-file "example.lsp"))
  ⇒ example

(format example "hello")            ⇒ nil
```

```
(close example)                          ⇒ implementation-defined
(setq example (open-input-stream "example.lsp" 8))
  ⇒ implementation-defined

(file-position example)                  ⇒ 0 (implementation-defined)
(read-byte example)                      ⇒ 104 (implementation-defined)
(file-position example)                  ⇒ 1 (implementation-defined)
```

`(set-file-position` *stream z*`)` → <*integer*> **function**

Attempts to change the file position (see `file-position`) of the stream *stream* to z. If it is not possible to move to the exact position z, some implementation-defined motion within the file might still be performed. The value returned is the new file position, which might or might not be z.

An error shall be signaled if *stream* is not a stream to or from a file, or if z is not a non-negative integer (error-id. *domain-error*).

<u>`Example:`</u>

```
(set-file-position example 4)       ⇒ 4
```

`(file-length` *filename element-class*`)` → <*object*> **function**

Returns the length of the file named by *filename*, or returns `nil` if the length cannot be determined. The *element-class* determines the units. An error shall be signaled if *filename* is not a string (error-id. *domain-error*).

<u>`Example:`</u>

```
(file-length "file27.dat" 8)        ⇒ 25
;; Implementations are not required to support byte size 2.
(file-length "file27.dat" 2)        ⇒ 100
```

29 Condition system

The condition system, sometimes called the “error system,” is a facility which permits problem situations detected at runtime to be represented and resolved while still under the control of a conforming program.

29.1 Conditions

When a problem situation is detected, a representation of that situation called a ***condition*** (or sometimes a "condition object" to emphasize its nature as an ordinary ISLisp object) is constructed and the situation represented by the condition is announced by a process called ***signaling***. This signaling process allows a dynamically established ***handler*** an opportunity to resolve the problem.

Figure 1 shows an inheritance graph for the various condition classes.

Some condition classes require initialization arguments when using `create` so that associated data can be provided. For more information, see §29.3.

Conditions that represent situations involving dynamically detected program errors are called ***error conditions***. Error conditions and those conditions that represent implementation limitations that may not be symptomatic of program errors are collectively called ***serious conditions***.

Note: In some dialects of Lisp a meaning is assigned to the idea of conditions that are not serious. Such conditions are beyond the scope of this document; hence the use of the class name `<serious-condition>` as the most general kind of condition defined herein.

29.2 Signaling and handling conditions

When a condition is signaled, the ***active handler*** is called with one argument, a condition which represents the situation. An initial active handler will have been established by the system; it will provide some implementation-defined action (such as return to toplevel, program exit, or entry into an interactive debugger). User programs may also establish handlers (see `with-handler`).

At any given time, only one handler is active. Establishing a new handler with `with-handler` shadows any previously active handler. This newly established handler is active throughout execution of its associated body of code unless shadowed by another use of `with-handler`.

If called, a handler function will execute in the dynamic environment of the call to `signal-condition`, *except* that the handler context is re-bound to match the dynamic handler state that was current at the point the handler function was established as the active handler.

Note: This means that handlers are not expected to handle errors in themselves. If a programmer wishes to have a handler handle its own errors, he might use `labels` to allow the function a way to refer to itself and might have the function re-establish itself as a handler within its own body.

When a handler is called, it must handle the condition by transferring control to a point outside of the call to `signal-condition`. Such a transfer of control might be made explicitly by use of `go`, `throw`, or `return-from` or implicitly by use of an abstract operation such as `continue-condition` that has an equivalent effect. The consequences are undefined if the handler returns normally; the handler is required to transfer control.

A handler may defer to previously established handlers by calling `signal-condition` on the condition object which it received as an argument.

29.2.1 Operations relating to condition signaling

(error *error-string* *obj**) → <*object*> **function**

An error shall be signaled.

error-string and the *objs* are advice to the implementation about how the error message might be textually described (using `format`), but whether or not that advice is used is implementation defined.

This is equivalent to:

```
(signal-condition
  (create (class <simple-error>)
          'format-string error-string
          'format-arguments (list obj*)))
  nil)
```

(cerror *continue-string* *error-string* *obj**) → <*object*> **function**

Like `error`, but the error that it signals is "continuable" (see `continue-condition`). The extra argument *continue-string* describes what happens if this function returns.

This is equivalent to:

```
(signal-condition
  (create (class <simple-error>)
          'format-string error-string
          'format-arguments (list obj*))
  (let ((str (create-string-output-stream)))
    (format str continue-string obj*)
    (get-output-stream-string str)))
```

(signal-condition *condition* *continuable*) → <*object*> **function**

Invokes the condition handling system on *condition*.

If *continuable* is `nil`, the results of attempting to "continue" (see `continue-condition`) are not defined except that the call to `signal-condition` will not return normally.

If *continuable* is not `nil`, it will be possible to return from the call to `signal-condition` (see `continue-condition`). In this case, the specific value of *continuable* may be a string indicating the effect of continuing, or it may be the symbol `t`, indicating that an implementation-defined string such as `"Continue with no special action."` is to be used.

Example:

```
(signal-condition (create (class <simple-error>)
                          'format-string "A ~A problem occurred."
                          'format-arguments '(bad))
                  nil)
```

29.2.2 Operations relating to condition handling

(`ignore-errors` *form**) → <*object*> **special operator**

Establishes a handler for `<error>`, such that if an error occurs during execution of *forms*, `ignore-errors` will immediately return `nil`. Then it executes *forms*, returning the value returned by the last *form* (or `nil` if there were no *forms*) if execution terminates normally.

(`report-condition` *condition stream*) → <*condition*> **generic function**

Presents a natural language description of *condition* to *stream*. This generic function may be specialized for user-defined condition classes.

(`condition-continuable` *condition*) → <*object*> **function**

Returns `nil` if *condition* is not continuable, or a string describing the effect of continuing otherwise.

(`continue-condition` *condition* [*value*]) *transfers control and data* **function**

"Continues" from *condition* by finding the call to `signal-condition` and arranging for it to perform a normal return of the *value*, which defaults to `nil`.

The consequences are undefined if the *condition* is not continuable.

(`with-handler` *handler form**) → <*object*> **special operator**

Evaluates *handler*, which must yield a function (called the "handler function"). The handler function is established as active handler (see §29.2) and then the *forms* are executed. If execution of *forms* finishes normally, the value of the last *form* (or `nil` if there are no *forms*) is returned.

29.3 Data associated with condition classes

Some of the condition classes defined by ISLisp permit data to be associated with a condition object at its time of creation and later retrieved. Initialization arguments and accessors for such classes are defined here.

29.3.1 Arithmetic errors

`<arithmetic-error>` `operation` *operation*

`operands` *operands*

The *operation* is the function that was being performed, and the *operands* is a list of the arguments it received.

(`arithmetic-error-operation` *arithmetic-error*) → <*function*> **function**
(`arithmetic-error-operands` *arithmetic-error*) → <*list*> **function**

These functions return the *operation* and *operands* supplied as data when creating the *arithmetic-error*. An error shall be signaled if *arithmetic-error* is not a condition of class `<arithmetic-error>` (error-id. *domain-error*).

29.3.2 Domain errors

`<domain-error>` `object` *object*

`expected-class` *expected-class*

The *object* is the offending object, and the *expected-class* is the class that it was expected to be.

(`domain-error-object` *domain-error*) → <*object*> **function**
(`domain-error-expected-class` *domain-error*) → <*class*> **function**

These functions return the *object* and *expected-class* supplied as data when creating the *domain-error*. An error shall be signaled if *domain-error* is not a condition of class `<domain-error>` (error-id. *domain-error*).

29.3.3 Parse errors

`<parse-error>` `string` *string*

`expected-class` *expected-class*

The *string* is the string that was being parsed, and the *expected-class* is the class that the textual notation in the *string* was expected to represent.

(`parse-error-string` *parse-error*) → *<string>* **function**
(`parse-error-expected-class` *parse-error*) → *<class>* **function**

These functions return the *string* and *expected-class* supplied as data when creating the *parse-error*. An error shall be signaled if *parse-error* is not a condition of class `<parse-error>` (error-id. *domain-error*).

29.3.4 Simple errors

`<simple-error>` `format-string` *format-string*

`format-arguments` *format-arguments*

The *format-string* (a string) and *format-arguments* (a list of objects) are passed through to `format` to construct the error message. Each object in the list given as *format-arguments* becomes a separate data argument, *obj*, in the call to `format`.

(`simple-error-format-string` *simple-error*) → *<string>* **function**
(`simple-error-format-arguments` *simple-error*) → *<list>* **function**

These functions return the *format-string* and *format-arguments* supplied as data when creating the *simple-error*. An error shall be signaled if *simple-error* is not a condition of class `<simple-error>` (error-id. *domain-error*).

29.3.5 Stream errors

`<stream-error>` `stream` *stream*

The *stream* is the stream on which the error occurred.

(`stream-error-stream` *stream-error*) → *<stream>* **function**

Returns the *stream* supplied as data when creating the *stream-error*. An error shall be signaled if *stream-error* is not a condition of class `<stream-error>` (error-id. *domain-error*).

29.3.6 Undefined entity errors

`<undefined-entity>` `name` *name*

`namespace` *namespace*

The *name* is a symbol representing the identifier which was undefined. The *namespace* is one of the symbols `variable`, `dynamic-variable`, `function`, or `class`.

(`undefined-entity-name` *undefined-entity*) → <*symbol*> **function**
(`undefined-entity-namespace` *undefined-entity*) → <*symbol*> **function**

These functions return the *name* and *namespace* supplied as data when creating the *undefined-entity*. An error shall be signaled if *undefined-entity* is not a condition of class `<undefined-entity>` (error-id. *domain-error*).

The result of `undefined-entity-namespace` will be one of the symbols `variable`, `dynamic-variable`, `function`, or `class`.

29.4 Error identification

The following is a summary of all named errors in the language and the semantics associated with each error.

arity-error Errors of this kind occur when a function is activated with a number of arguments that is not compatible with the number of parameters permitted by the function's definition. Errors of this kind are represented as conditions of class `<program-error>`.

cannot-create-array Errors of this kind occur when a request is made to allocate an array that cannot be allocated. Errors of this kind are represented as conditions of class `<storage-exhausted>`.

cannot-create-cons Errors of this kind occur when a request is made to allocate a cons that cannot be allocated. Errors of this kind are represented as conditions of class `<storage-exhausted>`.

cannot-create-list Errors of this kind occur when a request is made to allocate a list that cannot be allocated. Errors of this kind are represented as conditions of class `<storage-exhausted>`.

cannot-create-sequence Errors of this kind occur if a function that produces a sequence (*e.g.,* `subseq`) cannot allocate that sequence. Errors of this kind are represented as conditions of class `<storage-exhausted>`.

cannot-create-string Errors of this kind occur when a request is made to allocate a string that cannot be allocated. Errors of this kind are represented as conditions of class `<storage-exhausted>`.

cannot-create-vector Errors of this kind occur when a request is made to allocate a vector that cannot be allocated. Errors of this kind are represented as conditions of class `<storage-exhausted>`.

cannot-parse-number Errors of this kind occur when the *string* parameter received by the `parse-number` function cannot be classified as the textual representation of a number. Errors of this kind are represented as conditions of class `<parse-error>`.

control-error Errors of this kind occur when an attempt is made to leave a block more than once or when there is no outstanding catcher for a *catch tag*. Errors of this kind are represented as conditions of class `<control-error>`.

division-by-zero Errors of this kind occur when an attempt is made to divide by zero. Errors of this kind are represented as conditions of class `<division-by-zero>`.

domain-error Errors of this kind occur when the object given as argument to a standard function for which an argument class restriction is in effect is not an instance of the class to which the argument is restricted. Errors of this kind are represented as conditions of class `<domain-error>`.

end-of-stream Errors of this kind occur when an attempt to read a character or byte at the end-of-stream when *eos-error-p* argument is true, or when an attempt to read a more complex object is about to begin (*e.g.,* by `read` or `read-line`) but the end-of-stream is seen before parsing of that object has finished. Errors of this kind are represented as conditions of class `<end-of-stream>`.

immutable-binding Errors of this kind occur when an attempt is made to change an immutable binding. Errors of this kind are represented as conditions of class `<program-error>`.

improper-argument-list Errors of this kind occur when the last argument given to the `apply` function is not a proper list. Errors of this kind are represented as conditions of class `<program-error>`.

index-out-of-range Errors of this kind occur when the index given to a function that accesses an element in a sequence (such as `elt`) is an integer outside the range of the sequence. Errors of this kind are represented as conditions of class `<program-error>`.

not-an-input-stream Errors of this kind occur when an attempt is made to read from a stream which is not an input stream. Errors of this kind are represented as conditions of class `<domain-error>`.

not-an-output-stream Errors of this kind occur when an attempt is made to write to a stream which is not an output stream. Errors of this kind are represented as conditions of class `<domain-error>`.

unbound-variable Errors of this kind occur when an attempt is made to access an unbound variable. Errors of this kind are represented as conditions of class `<unbound-variable>`.

undefined-entity Errors of this kind occur when the entity denoted by an identifier does not exist when an access to that entity is made. Errors of this kind are represented as conditions of class `<undefined-entity>`.

undefined-function Errors of this kind occur when a function does not exist at its activation point. Errors of this kind are represented as conditions of class `<undefined-function>`.

Some errors that can occur have not been named in this document, and others might be added by the implementation. The above list should not be taken as an exhaustive list of all possible errors in the language.

30 Miscellaneous

(identity *obj*) → <*object*> **function**

Returns an object that is the same as *obj* under `eql`. *obj* may be any ISLISP object.

Example:

```
(identity '(a b c))                ⇒ (a b c)
```

(get-universal-time) → <*integer*> **function**

Returns an approximation to the "current time" in Universal Time Format. The units are seconds. Universal Time Format represents time as an integer number of seconds since the beginning (*i.e.,* midnight), January 1, 1900 UT (ignoring leap seconds). If `get-universal-time` is called twice, the first value shall be less than or equal to the second value.

No implementation is required to have a way to verify that the time returned is correct. However, an error shall be signaled if an implementation can determine that the time it would return would not be correct (*e.g.,* it can determine that the clock was never initialized).

Example:

```
(get-universal-time)               ⇒ 2901312000
```

(get-internal-run-time) → <*integer*> **function**
(get-internal-real-time) → <*integer*> **function**

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

`get-internal-run-time` returns as an *integer* the current run time in *internal time units*. The precise meaning of this quantity is implementation defined. The difference between the values of two calls to this function is the amount of time between the two calls during which computational effort was expended on behalf of the executing program.

(internal-time-units-per-second) → <*integer*> **function**

`internal-time-units-per-second` returns the number of time units per second for the implementation.

Index