

INTERNATIONAL STANDARD

ISO/IEC
13816

First edition
1997-05-01

Information technology — Programming languages, their environments and system software interfaces — Programming language ISLISP

*Technologies de l'information — Langages de programmation, leurs
environnements et interfaces système — Langage de programmation
ISLISP*



Reference number
ISO/IEC 13816:1997(E)



Documents

Document Sharing Hub

Contents

1	Scope, Conventions and Compliance	1
1.1	Scope	1
1.2	Normative References	1
1.3	Notation and Conventions	1
1.4	Lexemes	4
1.4.1	Separators	5
1.4.2	Comments	5
1.5	Textual Representation	5
1.6	Reserved Identifiers	6
1.7	Definitions	6
1.8	Errors	9
1.8.1	Classes of error specification	9
1.8.2	Pervasive Error Types	9
1.9	Compliance of ISLISP Processors and Text	10
2	Classes	10
2.1	Metaclasses	11
2.2	Predefined Classes	13
2.3	Standard Classes	14
2.3.1	Slots	14
2.3.2	Creating Instances of Classes	14
3	Scope and Extent	14
3.1	The Lexical Principle	15
3.2	Scope of Identifiers	15
3.3	Some Specific Scope Rules	15
3.4	Extent	16
4	Forms and Evaluation	17
4.1	Forms	17
4.2	Function Application Forms	18
4.3	Special Forms	18
4.4	Defining Forms	19
4.5	Macro Forms	19
4.6	The Evaluation Model	19
4.7	Functions	20
4.8	Defining Operators	24

© ISO/IEC 1997

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

5	Predicates	26
5.1	Boolean Values	26
5.2	Class Predicates	26
5.3	Equality	26
5.4	Logical Connectives	29
6	Control Structure	30
6.1	Constants	30
6.2	Variables	31
6.3	Dynamic Variables	35
6.4	Conditional Expressions	36
6.5	Sequencing Forms	38
6.6	Iteration	39
6.7	Non-Local Exits	40
6.7.1	Establishing and Invoking Non-Local Exits	40
6.7.2	Assuring Data Consistency during Non-Local Exits	44
7	Objects	45
7.1	Defining Classes	45
7.1.1	Determining the Class Precedence List	48
7.1.2	Accessing Slots	48
7.1.3	Inheritance of Slots and Slot Options	49
7.2	Generic Functions	49
7.2.1	Defining Generic Functions	50
7.2.2	Defining Methods for Generic Functions	51
7.2.2.1	Agreement on Parameter Specializers and Qualifiers	53
7.2.2.2	Congruent Lambda-Lists for all Methods of a Generic Function	53
7.2.3	Inheritance of Methods	53
7.3	Calling Generic Functions	53
7.3.1	Selecting the Applicable Methods	54
7.3.2	Sorting the Applicable Methods	54
7.3.3	Applying Methods	55
7.3.3.1	Simple Method Combination	55
7.3.3.2	Standard Method Combination	55
7.3.4	Calling More General Methods	56
7.4	Object Creation and Initialization	57
7.4.1	Initialize-Object	58
7.5	Class Enquiry	59
8	Macros	60
9	Declarations and Coercions	61
10	Symbol class	63
10.1	Symbol Names	63
10.1.1	Notation for Symbols	64
10.1.2	Alphabetic Case in Symbol Names	64
10.1.3	nil and ()	65
10.2	Symbol Properties	65
10.3	Unnamed Symbols	66
11	Number class	67
11.1	Number class	67
11.2	Float class	76
11.3	Integer class	78

12 Character class	81
13 List class	83
13.1 Cons	83
13.2 Null class	85
13.3 List operations	86
14 Arrays	90
14.1 Array Classes	90
14.2 General Arrays	91
14.3 Array Operations	91
15 Vectors	94
16 String class	95
17 Sequence Functions	98
18 Stream class	101
18.1 Streams to Files	102
18.2 Other Streams	104
19 Input and Output	105
19.1 Argument Conventions for Input Functions	105
19.2 Character I/O	106
19.3 Binary I/O	110
20 Files	111
21 Condition System	113
21.1 Conditions	113
21.2 Signaling and Handling Conditions	114
21.2.1 Operations relating to Condition Signaling	114
21.2.2 Operations relating to Condition Handling	115
21.3 Data associated with Condition Classes	116
21.3.1 Arithmetic Errors	116
21.3.2 Domain Errors	117
21.3.3 Parse Errors	117
21.3.4 Simple Errors	117
21.3.5 Stream Errors	118
21.3.6 Undefined Entity Errors	118
21.4 Error Identification	118
22 Miscellaneous	120
Index	122

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13816 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Introduction

The programming language ISLISP is a member of the LISP family. It is the result of standardization efforts by ISO/IEC JTC 1/SC 22/WG 16.

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.

ISO/IEC JTC 1/SC 22/WG 16 wishes to thank the many specialists who contributed to this International Standard.

Information technology — Programming languages, their environments and system software interfaces — Programming language ISLISP

1.1 Scope

1. Positive Scope

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

2. Negative Scope

This International Standard 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.

1.2 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

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

1.3 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 International Standard 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* represents 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 International Standard (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 International Standard 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:

$$(+ \ 3 \ 4) \Rightarrow 7$$

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

$$(+ \ i_1 \ i_2) \rightarrow \langle \text{integer} \rangle$$

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 <i>argument*</i>) → <i>result-class</i>	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 International Standard 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:

<i>term</i> ⁺	denotes one or more occurrences of <i>term</i> ;
<i>term</i> [*]	denotes zero or more occurrences of <i>term</i> ;
[<i>term</i>]	denotes at most one occurrence of <i>term</i> , commonly one says that <i>term</i> is optional;
{ <i>term</i> ₁ <i>term</i> ₂ ...}	denotes grouping of <i>terms</i> .
<i>term</i> ₁ <i>term</i> ₂ ...	denotes grouping of alternative <i>terms</i> .

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

<i>array</i> , <i>array</i> ₁ , ... <i>array</i> _j , ...	<i><basic-array></i>
<i>cons</i> , <i>cons</i> ₁ , ... <i>cons</i> _j , ...	<i><cons></i>
<i>list</i> , <i>list</i> ₁ , ... <i>list</i> _j , ...	<i><list></i>
<i>obj</i> , <i>obj</i> ₁ , ... <i>obj</i> _j , ...	<i><object></i>
<i>sequence</i> , <i>sequence</i> ₁ , ... <i>sequence</i> _j , ...	<i><basic-vector></i> or <i><list></i> (see §17)
<i>stream</i> , <i>stream</i> ₁ , ... <i>stream</i> _j , ...	<i><stream></i>
<i>string</i> , <i>string</i> ₁ , ... <i>string</i> _j , ...	<i><string></i>
<i>char</i> , <i>char</i> ₁ , ... <i>char</i> _j , ...	<i><character></i>
<i>function</i> , <i>function</i> ₁ , ... <i>function</i> _j , ...	<i><function></i>
<i>class</i> , <i>class</i> ₁ , ... <i>class</i> _j , ...	<i><class></i>
<i>symbol</i> , <i>symbol</i> ₁ , ... <i>symbol</i> _j , ...	<i><symbol></i>
<i>x</i> , <i>x</i> ₁ , ... <i>x</i> _j , ...	<i><number></i>
<i>z</i> , <i>z</i> ₁ , ... <i>z</i> _j , ...	<i><integer></i>

In this International Standard 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 *namep* 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 International Standard, 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*.

1.4 Lexemes

An ISLISP text is built up from lexemes. Lexemes are built up from at least the following characters (see §12):

```
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 §13.1 and §8):

```
( ) , ' `
```

The following character tuples (where *n* is a sequence of digits) are individual lexemes (see §4.7, §8, and §14.1):

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

The textual representations of symbols (see §10), numbers (see §11), characters (see §12), and strings (see §16) 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 §16) or inside a corresponding pair of multiple escape characters (see §10) or for the character immediately following #\.

1.4.1 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.

1.4.2 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.

1.5 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 §1.4

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 ... *obj*_{*n*}). A dotted list (*i.e.*, a list whose last tail is not **nil**) appears as (*obj*₁ *obj*₂ ... *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:

$$\begin{aligned} &[s]dd\dots d.dd\dots d \\ &[s]dd\dots d.dd\dots dE[s]dd\dots d \\ &[s]dd\dots d.dd\dots de[s]dd\dots d \\ &[s]dd\dots dE[s]dd\dots d \\ &[s]dd\dots de[s]dd\dots d \end{aligned}$$

where *s* is either “+” or “–”, and *d* is one of “0”–“9”. For example: 987.12, +12.5E-13, -1.5E12, 1E32¹.

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

Vector A vector of class `<general-vector>` is written as `#(obj1 ... objn)`.

Array An array of class `<general-array*>` or `<general-vector>` can be written on input as `#na` (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 `(obj1 ... objn)`. If $n > 1$ and the dimensions are $n_1\ n_2\ \dots$, the structure is `(str1 ... strn1)`, where the `stri` are the structures of the n_1 subarrays, each of which has dimensions `(n2 ...)`. 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 §10. 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.

1.6 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**.

1.7 Definitions

For the purposes of this International Standard, the following definitions apply:

1.7.1 **abstract class**: A class that by definition has no direct instances.

1.7.2 **activation**: Computation of a function. 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.

1.7.3 **accessor**: Association of a reader and a writer for a slot of an instance.

1.7.4 **binding**: Binding has both a syntactic and a semantic aspect.

Syntactically, "binding" describes the relation between an identifier and a binding ISLISP form. The property of being bound can be checked textually by relating defining and applied identifier occurrences.

Semantically, "binding" describes the relation between a variable, its denoting identifier, and an object (or, the relation between a variable and a location). This 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.

1.7.5 **class**: Object, that determines the structure and behavior of a set of other objects, called its instances. The behavior is the set of operations that can be performed on an instance.

- 1.7.6 **condition**: An object that represents a situation that has been (or might be) detected by a running program.
- 1.7.7 **definition point**: An identifier represents an ISLISP object starting with its definition point, which is a textual point of an ISLISP text.
- 1.7.8 **direct instance**: 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.
- 1.7.9 **dynamic**: Having an effect that is determined only through program execution and that cannot, in general, be determined statically.
- 1.7.10 **dynamic variable**: A 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.
- 1.7.11 **evaluation**: Computation of a form prepared for execution which results in a value and/or a side effect.
- 1.7.12 **execution**: A sequence of (sometimes nested) activations.
- 1.7.13 **extension**: An implementation-defined modification to the requirements of this International Standard that does not invalidate any ISLISP text complying with this International Standard (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.
- 1.7.14 **form**: A single, syntactically valid unit of program text, capable of being prepared for execution.
- 1.7.15 **function**: An ISLISP object that is called with arguments, performs a computation (possibly having side-effects), and returns a value.
- 1.7.16 **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.
- 1.7.17 **identifier**: A lexical element (lexeme) which designates an ISLISP object. In the data structure representation of ISLISP texts, identifiers are denoted by symbols.
- 1.7.18 **immutable binding**: A binding is immutable if the relation between an identifier and the object represented by this identifier cannot be changed. It is a violation if there is attempt to change an immutable binding (error-id. *immutable-binding*).
- 1.7.19 **immutable object**: An object is immutable if it 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. 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.
- 1.7.20 **implementation defined**: A feature, possibly differing between different ISLISP processors, but completely defined for every processor.
- 1.7.21 **implementation dependent**: A 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.
- 1.7.22 **inheritance**: Relation between a class and its superclass which maps structure and behavior of the superclass onto the class. ISLISP supports a restricted form of multiple inheritance; *i.e.*, a class may have several superclasses at once.

- 1.7.23 **instance (of a class)**: Either a direct instance of a class or an instance of one of its subclasses.
- 1.7.24 **literal**: An object whose representation occurs directly in a program as a constant value.
- 1.7.25 **metaclass**: A class whose instances are themselves classes.
- 1.7.26 **method**: Case of a generic function for a particular parameter profile, which defines the class-specific behavior and operations of the generic function.
- 1.7.27 **object**: An object is anything that can be created, destroyed, manipulated, compared, stored, input, or output by the ISLISP processor. In particular, functions are ISLISP objects. 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*.
- 1.7.28 **operator**: the 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.
- 1.7.29 **parameter profile**: Parameter list of a method, where each formal parameter is accompanied by its class name. If a parameter is not accompanied by a class name, it belongs to the most general class.
- 1.7.30 **place**: Objects can be stored in places and retrieved later. 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**.
- 1.7.31 **position**:
- (a) argument position: Occurrence of a text unit as an element in a form excluding the first one.
 - (b) operator position: Occurrence of a text unit as the first element in a form.
- 1.7.32 **process**: The execution of an ISLISP text prepared for execution.
- 1.7.33 **processor**: A 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.
- 1.7.34 **program**: An aggregation of expressions to be evaluated, the specific nature of which depends on context. Within this International Standard, the term “program” is used only in an abstract way; there is no specific syntactic construct that delineates a program.
- 1.7.35 **scope**: The scope of an identifier is that textual part of a program where the meaning of that identifier is defined; *i.e.*, there exists an ISLISP object designated by this identifier.
- 1.7.36 **slot**: A named component of an instance which can be accessed using the slot accessors. The structure of an instance is defined by the set of its slots.
- 1.7.37 **text**: A text that complies with the requirements of this International Standard (*i.e.*, with the syntax and static semantics of ISLISP). An ISLISP text consists of a sequence of toplevel forms.
- 1.7.38 **toplevel form**: Any form that either is not nested in any other form or is nested only in **progn** forms.
- 1.7.39 **toplevel scope**: The scope in which a complete ISLISP text unit is processed.
- 1.7.40 **writer**: A method associated with a slot of a class, whose task is to bind a value with a slot of an instance of that class.

1.8 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 International Standard. 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 International Standard are not met. A violation shall be detected during preparation for execution.

1.8.1 Classes of error specification

The wording of error specification in this International Standard 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 expression shall stop. 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 International Standard 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.

1.8.2 Pervasive Error Types

Most errors are described in detail in the context 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 occurring 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 §21.4.

1.9 Compliance of ISLISP Processors and Text

An ISLISP processor complying with the requirements of this International Standard shall

- (a) accept and implement all features of ISLISP specified in this International Standard.
- (b) reject any text that contains any textual usage which this International Standard explicitly defines to be a violation (see §1.8).
- (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 International Standard; these extensions shall be described as being “extensions to ISLISP as specified by ISO/IEC 13816:1997(E).”

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 International Standard.

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

2 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 International Standard to be a direct superclass of C_2 (for example, by indenting C_2 under C_1 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 $C_2, \dots, C_{n-1}$ 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_1 is direct subclass of C_2 . 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.

2.1 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 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.

```

<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>    ;; Note: <null> also inherits from <symbol>
  <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>
  <symbol>
    <null>    ;; Note: <null> also inherits from <list>

```

Subclasses appear indented under superclasses.

Figure 1. Class Inheritance

2.2 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> 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 International Standard. However, additional implementation-specific subclass relationships may exist between implementation-specific classes and classes defined in this International Standard.
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.

2.3 Standard Classes

2.3.1 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 §7.1.3.

2.3.2 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.

3 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 §1.3) 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.

3.1 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.

3.2 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.

3.3 Some Specific Scope Rules

The **toplevel scope** is the scope of identifiers of required built-in functions, required built-in macros, and constants.

Reserved identifiers are not subject to the lexical principle, because they are not identifiers. They cannot be defined or bound. See §1.6.

```

(let ((a1 f-a1)
      ...
      (x f-x)      ...
      (z1 f-z1))
  ...
  (let ((a2 f-a2)
        ...
        (x f-x2)
        ...
        (z2 f-z2))
    ...
  )
  ...
)

```

; now a1...x...z1 are applicable, their scope begins here
 ; a1...x...z1 might be defined newly, but:
 ; the outer a1...x...z1 are still usable
 ; the inner a2...x...z2 are not yet usable
 ; 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

3.4 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-output-file
flet	tagbody	with-standard-input
for	with-error-output	with-standard-output
labels	with-open-input-file	
let	with-open-io-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.

4 Forms and Evaluation

4.1 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 International Standard 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 §6.7.1). 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, or an identifier, or a lambda expression. The identifier names a function, 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**.

4.2 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 International Standard describes a function application form in the following format:

<i>(function-name argument*) → result-class</i>	function
---	-----------------

This describes an ordinary function.

<i>(generic-function-name argument*) → result-class</i>	generic function
---	-------------------------

This describes a generic function.

<i>(local-function-name argument*) → result-class</i>	local function
---	-----------------------

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

4.3 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 §4.5 and §8). Special forms are recognized because they have a **special operator** in their operator position. The following are special operators:

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

There might be additional, implementation-defined special operators.

This International Standard describes the evaluation of special forms in the following format:

<i>(special-operator argument*) → result-class</i>	special operator
--	-------------------------

4.4 Defining Forms

A **defining form** is a toplevel special form (see §4.3) 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 International Standard describes defining forms in the following format:

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

4.5 Macro Forms

Macro forms are expanded during preparation for execution. It is implementation defined whether any operator described by this International Standard as a macro is implemented as a special operator (see §4.3).

For information on how macros are processed, see §8.

4.6 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 §8). 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 §1.8.2) (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)` $\Rightarrow$ 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 §1.8.2) (error-id. *undefined-function*). The *arguments* are evaluated in order from left to right, yielding objects (sometimes called “actual arguments”) to which the function will be applied. 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 §1.8.2 for descriptions of error situations that might occur during execution of the above cases.

4.7 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

<code>(functionp obj) $\rightarrow$ boolean</code>	function
---	-----------------

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

Example:

`(functionp (function car))` $\Rightarrow$ 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 <i>function-name</i>) → < <i>function</i> >	special operator syntax
#' <i>function-name</i> → < <i>function</i> >	

This special form denotes a reference to the function named by the identifier *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 §1.8.2) (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 <i>lambda-list</i> <i>form*</i>) → < <i>function</i> >	special operator
---	------------------

Where:

$$\textit{lambda-list} ::= (\textit{identifier}^* [\&\textit{rest} \textit{identifier}]) \mid (\textit{identifier}^* [: \textit{rest} \textit{identifier}])$$

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 §6.7.1).

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	((<i>function-name</i> <i>lambda-list form</i>[*])[*])	<i>body-forms</i>[*]	→	<object>	special operator
(flet	((<i>function-name</i> <i>lambda-list form</i>[*])[*])	<i>body-forms</i>[*]	→	<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 *funcs* (*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 function 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

```

<code>(apply function obj* list) → <object></code>	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 §1.5) (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

```

<code>(funcall function obj*) → <object></code>	function
---	-----------------

Activates the specified function *function* and returns the value that the function returns. The *i*th 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 *argument* 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
```

4.8 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 <i>name</i> <i>form</i>) → < <i>symbol</i> >	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 <i>name</i> <i>form</i>) → < <i>symbol</i> >	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 <i>name</i> <i>form</i>) → < <i>symbol</i> >	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)          ⇒ red
(dynamic *color*)                  ⇒ red
(defun what-color () (dynamic *color*))
    ⇒ what-color
(what-color)                       ⇒ red
(dynamic-let ((*color* 'green)) (what-color))
    ⇒ green

```

(defun <i>function-name</i> <i>lambda-list</i> <i>form*</i>) → < <i>symbol</i> >	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 (*i.e.*, those which are not contained in the lambda list) follow the rules of lexical scoping.

Example:

```

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

```

5 Predicates

5.1 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.

<code>t</code>	$\rightarrow$ <code><symbol></code>	named constant
<code>nil</code>	$\rightarrow$ <code><null></code>	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.

5.2 Class Predicates

The following functions are one-argument class membership predicates:

<code>basic-array-p</code>	<code>floatp</code>	<code>integerp</code>	<code>stringp</code>
<code>basic-array-p</code>	<code>functionp</code>	<code>listp</code>	<code>symbolp</code>
<code>basic-vector-p</code>	<code>general-array-p</code>	<code>null</code>	
<code>characterp</code>	<code>general-vector-p</code>	<code>numberp</code>	
<code>consp</code>	<code>generic-function-p</code>	<code>streamp</code>	

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

5.3 Equality

<code>(eq obj₁ obj₂)</code>	$\rightarrow$ <code>boolean</code>	function
<code>(eql obj₁ obj₂)</code>	$\rightarrow$ <code>boolean</code>	function

`eq` and `eql` test whether `obj1` and `obj2` 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 undefined if either `obj1` or `obj2` is a number or a character. For `eq1` the meaning for numbers and characters is defined as follows:

- If `obj1` and `obj2` are numbers, `eq1` tests whether they are direct instances of the same classes and have the same value.
If an implementation supports positive and negative zeros as *distinct* values, then `(eq1 0.0 -0.0)` returns `nil`. When the syntax `-0.0` is read and it is interpreted as the value `0.0` then `(eq1 0.0 -0.0)` returns `t`.
- If `obj1` and `obj2` are characters, `eq1` tests whether they are the same character (see `char=`).

Example:

<code>(eq1 () ())</code>	$\Rightarrow$ <code>t</code>
<code>(eq () ())</code>	$\Rightarrow$ <code>t</code>
<code>(eq1 '() '())</code>	$\Rightarrow$ <code>t</code>
<code>(eq '() '())</code>	$\Rightarrow$ <code>t</code>
<code>(eq1 'a 'a)</code>	$\Rightarrow$ <code>t</code>
<code>(eq 'a 'a)</code>	$\Rightarrow$ <code>t</code>
<code>(eq1 'a 'A)</code>	$\Rightarrow$ <code>t</code>
<code>(eq 'a 'A)</code>	$\Rightarrow$ <code>t</code>
<code>(eq1 'a 'b)</code>	$\Rightarrow$ <code>nil</code>
<code>(eq 'a 'b)</code>	$\Rightarrow$ <code>nil</code>
<code>(eq1 'f 'nil)</code>	$\Rightarrow$ <code>nil</code>
<code>(eq 'f 'nil)</code>	$\Rightarrow$ <code>nil</code>
<code>(eq1 2 2)</code>	$\Rightarrow$ <code>t</code>
<code>(eq 2 2)</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(eq1 2 2.0)</code>	$\Rightarrow$ <code>nil</code>
<code>(eq 2 2.0)</code>	$\Rightarrow$ <code>nil</code>
<code>(eq1 100000000 100000000)</code>	$\Rightarrow$ <code>t</code>
<code>(eq 100000000 100000000)</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(eq1 10.00000 10.0)</code>	$\Rightarrow$ <code>t</code>
<code>(eq 10.00000 10.0)</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(eq1 (cons 1 2) (cons 1 2))</code>	$\Rightarrow$ <code>nil</code>
<code>(eq (cons 1 2) (cons 1 2))</code>	$\Rightarrow$ <code>nil</code>
<code>(let ((x '(a))) (eq1 x x))</code>	$\Rightarrow$ <code>t</code>
<code>(let ((x '(a))) (eq x x))</code>	$\Rightarrow$ <code>t</code>
<code>(eq1 '(a) '(a))</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(eq '(a) '(a))</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(let ((x '(b)) (y '(a b))) (eq1 x (cdr y)))</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(let ((x '(b)) (y '(a b))) (eq x (cdr y)))</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(eq1 '(b) (cdr '(a b)))</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(eq '(b) (cdr '(a b)))</code>	$\Rightarrow$ <code>nil or t (implementation-defined)</code>
<code>(let ((p (lambda (x) x)) (eq1 p p))</code>	$\Rightarrow$ <code>t</code>
<code>(let ((p (lambda (x) x)) (eq p p))</code>	$\Rightarrow$ <code>t</code>
<code>(let ((x "a")) (eq1 x x))</code>	$\Rightarrow$ <code>t</code>

(let ((x "a")) (eq x x))	⇒ t
(eq1 "a" "a")	⇒ nil or t (implementation-defined)
(eq "a" "a")	⇒ nil or t (implementation-defined)
(let ((x "")) (eq1 x x))	⇒ t
(let ((x "")) (eq x x))	⇒ t
(eq1 "" "")	⇒ nil or t (implementation-defined)
(eq "" "")	⇒ nil or t (implementation-defined)
(eq1 #\a #\A)	⇒ nil
(eq #\a #\A)	⇒ nil
(eq1 #\a #\a)	⇒ t
(eq #\a #\a)	⇒ nil or t (implementation-defined)
(eq1 #\space #\Space)	⇒ t
(eq #\space #\Space)	⇒ nil or t (implementation-defined)
(eq1 #\space #\space)	⇒ t
(eq #\space #\space)	⇒ nil or t (implementation-defined)

(equal *obj₁* *obj₂*) → *boolean*

function

This function tests whether *obj₁* and *obj₂* are isomorphic—i.e., whether *obj₁* and *obj₂* denote the same structure with equivalent values. **equal** returns **t** if the test was satisfied, and **nil** if not. Specifically:

If *obj₁* and *obj₂* are instances of the same classes, **equal** returns **t** if they are **eq1**. Otherwise (if they are direct instances of the same classes but not **eq1**), the result is **t** if one of the following cases applies:

(a) lists: either *obj₁* and *obj₂* 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** *obj₁* *ind₁* ... *ind_n*)

(equal (**aref** *obj₁* *ind₁* ... *ind_n*)
(**aref** *obj₂* *ind₁* ... *ind_n*)) is satisfied.

Otherwise the value is **nil**.

obj₁ and *obj₂* 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

```

5.4 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 form1 form2 ... formn) ≡ (if form1 (and form2 ... formn) 'nil)2

```

Example:

²For the definition of **if**, see §6.4 below.

```

(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 “ $\vee$ ”). *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 form1 form2 ... formn) ≡ ((lambda (var)
                                (if var var (or form2 ... formn))) form1)
                                where var does not occur in form2 ... formn

```

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

```

6 Control Structure

6.1 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 <i>obj</i>) → < <i>object</i> >	special operator syntax
' <i>obj</i> → < <i>object</i> >	

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 1)	⇒ (car 1)
'(+ 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.

6.2 Variables

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

A **variable** is 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	<i>var</i>
dynamic bindings	(dynamic <i>var</i>)
the components of a basic-array	(aref <i>basic-array</i> <i>z</i> ₁ ... <i>z</i> _{<i>n</i>})
the components of a general array	(garef <i>general-array</i> <i>z</i> ₁ ... <i>z</i> _{<i>n</i>})
the components of a list	(elt <i>list</i> <i>z</i>)
the components of a vector	(elt <i>basic-vector</i> <i>z</i>)
the left component of a cons	(car <i>cons</i>)
the right component of a cons	(cdr <i>cons</i>)
a property of a symbol	(property <i>symbol</i> <i>property</i>)
a slot of an instance of a class	(accessor-name <i>instance</i>)

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 *place* 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 *form* 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 *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 () <i>body-form</i> *)	≡	(progn <i>body-form</i> *) ³
(let ((<i>var</i> ₁ <i>form</i> ₁) (<i>var</i> ₂ <i>form</i> ₂) ... (<i>var</i> _{<i>n</i>} <i>form</i> _{<i>n</i>}) <i>body-form</i> *)	≡	((lambda (<i>var</i> ₁ <i>var</i> ₂ ... <i>var</i> _{<i>n</i>}) <i>body-form</i> *) <i>form</i> ₁ <i>form</i> ₂ ... <i>form</i> _{<i>n</i>}) ⁴

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* ((*var 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* excluding nested regions of *var*, if any, 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 definitions enlarge the set of current valid identifiers perhaps shadowing previous definitions (in case some *var* was defined outside), and in this enlarged or modified environment the *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)
      (var2 form2)
      ...
      (varn formn)
      body-form*)
  ≡ (let ((var1 form1)
          (var2 form2)
          ...
          (varn formn)
          body-form*)...)

```

Example:

```

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

```

³For the definition of **progn** see §6.5 below.

⁴For the definition of **lambda** see 5.6.d.

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

6.3 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 <i>var</i>) → <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 <i>var</i>) <i>form</i>) → <object>	special form
---	---------------------

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 ((<i>var form</i>)*) <i>body-form</i> *) → <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.