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)

Contents

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) ⇒ 7
```

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

(`+` i_1 i_2) → <*integer*>

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 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:

$term^+$	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 values obey the respective class restrictions:

array, $array_1, \ldots array_j, \ldots$	`<basic-array>`
cons, $cons_1, \ldots cons_j, \ldots$	`<cons>`
list, $list_1, \ldots list_j, \ldots$	`<list>`
obj, $obj_1, \ldots obj_j, \ldots$	`<object>`
sequence, $sequence_1, \ldots sequence_j, \ldots$	`<basic-vector>` or `<list>` (see §17)
stream, $stream_1, \ldots stream_j, \ldots$	`<stream>`
string, $string_1, \ldots string_j, \ldots$	`<string>`
char, $char_1, \ldots char_j, \ldots$	`<character>`
function, $function_1, \ldots function_j, \ldots$	`<function>`
class, $class_1, \ldots class_j, \ldots$	`<class>`
symbol, $symbol_1, \ldots symbol_j, \ldots$	`<symbol>`
$x, x_1, \ldots x_j, \ldots$	`<number>`
$z, z_1, \ldots z_j, \ldots$	`<integer>`

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 *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 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:

[*s*]*dd*...*d*.*dd*...*d*
[*s*]*dd*...*d*.*dd*...*d*E[*s*]*dd*...*d*
[*s*]*dd*...*d*.*dd*...*d*e[*s*]*dd*...*d*
[*s*]*dd*...*d*E[*s*]*dd*...*d*
[*s*]*dd*...*d*e[*s*]*dd*...*d*

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

[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 `#(`$obj_1 \ldots obj_n$`)`.

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 §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*).