ISO Lisp Object System Specification

2. Functions in the Programmer Interface

Author: Richard P. Gabriel

Based on the document "Common Lisp Object System Specification" by Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Draft Dated: April 15, 1992

The author wishes to thank Patrick Dussud and Jon L White for their contributions to this document.

CONTENTS

Introduction

This chapter describes the functions, macros, special forms, and generic functions provided by the ISO Lisp Object System Programmer Interface. The Programmer Interface comprises the functions and macros that are sufficient for writing most object-oriented programs.

This chapter is reference material that requires an understanding of the basic concepts of the ISO Lisp Object System. The functions are arranged in alphabetical order for convenient reference.

The description of each function, macro, special form, and generic function includes its purpose, its syntax, the semantics of its arguments and returned values, and often an example and cross-references to related functions.

The syntax description for a function, macro, or special form describes its parameters. The following is an example of the format for the syntax description of a function:

Syntax:

F *x y* `&optional` *z* `&key` *k* [*Generic Function*]

This description indicates that the generic function **F** has two required parameters, *x* and *y*. In addition, there is an optional parameter *z* and a keyword parameter *k*.

The generic functions described in this chapter are all standard generic functions. They all use standard method combination.

The description of a generic function includes descriptions of the methods that are defined on that generic function by the ISO Lisp Object System. A ***method signature*** is used to describe the parameters and parameter specializers for each method. The following is an example of the format for a method signature:

Method Signature:

F (*x class*) (*y* `t`) `&optional` *z* `&key` *k* [*Primary Method*]

This signature indicates that this method on the generic function **F** has two required parameters, *x*, which must be an instance of the class *class*, and *y*, which can be any object. In addition, there is an optional parameter *z* and a keyword parameter *k*. This signature also indicates that this method on **F** is a primary method and has no qualifiers.

The syntax description for a generic function describes the lambda-list of the generic function itself, while the method signatures describe the lambda-lists of the defined methods.

Any implementation of the ISO Lisp Object System is allowed to provide additional methods on the generic functions described in this chapter.

It is useful to categorize the functions and macros according to their role in this standard:

- Tools used for simple object-oriented programming

  These tools allow for defining new classes, methods, and generic functions, and for making instances. Some tools used within method bodies are also listed here. Some of the macros listed here have a corresponding function that performs the same task at a lower level of abstraction.

  call-next-method
  defclass
  defgeneric
  defmethod
  initialize-instance
  make-instance
  next-method-p
  slot-boundp

- Functions underlying the commonly used macros

  class-name
  find-class
  slot-exists-p

- General ISO Lisp support tools

  class-of

Notation

This specification uses an extended Backus Normal Form (BNF) to describe the syntax of the Object System. This section discusses the syntax of BNF expressions. The primary extension used is the following:

$$[\![O]\!]$$

An expression of this form will appear whenever a list of elements is to be spliced into a larger structure and the elements can appear in any order. The symbol O represents a description of the syntax of some number of syntactic elements to be spliced; that description must be of the form

$$O_1 \mid \ldots \mid O_N$$

where each O_i can be either of the form S or of the form $S*$. The expression $[\![O]\!]$ means that a list of the form

$$(O_{i_1} \ldots O_{i_j}) \quad 1 \leq j$$

is spliced into the enclosing expression, such that if $n \neq m$ and $1 \leq n, m \leq j$, then either $O_{i_n} \neq O_{i_m}$ or $O_{i_n} = O_{i_m} = Q_k$, where for some $1 \leq k \leq N$, O_k is of the form Q_k*.

For example, the expression

$$(\texttt{x}\ [\![\ \texttt{A} \mid \texttt{B*} \mid \texttt{C}\ ]\!]\ \texttt{y})$$

means that at most one `A`, any number of `B`'s, and at most one `C` can occur in any order. It is a description of any of these:

```
(x y)
(x B A C y)
(x A B B B B B C y)
(x C B A B B B y)
```

but not any of these:

```
(x B B A A C C y)
(x C B C y)
```

In the first case, both `A` and `C` appear too often, and in the second case `C` appears too often.

A simple indirection extension is introduced in order to make this new syntax more readable:

$$\downarrow O$$

If O is a non-terminal symbol, the right-hand side of its definition is substituted for the entire expression $\downarrow O$. For example, the following BNF is equivalent to the BNF in the previous example:

$$(\texttt{x}\ [\![\ \downarrow O\ ]\!]\ \texttt{y})$$

$$O ::= \texttt{A} \mid \texttt{B*} \mid \texttt{C}$$

call-next-method *Function*

Purpose:

The function **call-next-method** can be used within the body of a method defined by a method-defining form to call the next method.

The function **call-next-method** returns the value or values returned by the method it calls. If there is no next method, the generic function an error is signaled.

The type of method combination used determines which methods can invoke **call-next-method**. The standard method combination type allows **call-next-method** to be used within primary methods and **:around** methods. The standard method combination type defines the next method as follows:

- If **call-next-method** is used in an **:around** method, the next method is the next most specific **:around** method, if one is applicable.
- If there are no **:around** methods at all or if **call-next-method** is called by the least specific **:around** method, other methods are called as follows:
  - All the **:before** methods are called, in most-specific-first order. The function **call-next-method** cannot be used in **:before** methods.
  - The most specific primary method is called. Inside the body of a primary method, **call-next-method** may be used to pass control to the next most specific primary method. An error is signaled if **call-next-method** is used and there are no more primary methods.
  - All the **:after** methods are called in most-specific-last order. The function **call-next-method** cannot be used in **:after** methods.

For further discussion of **call-next-method**, see the sections "Standard Method Combination" and "Built-in Method Combination Types."

Syntax:

call-next-method [*Function*]

Arguments:

call-next-method passes the current method's original arguments to the next method. Neither argument defaulting, nor using **setq**, nor rebinding variables with the same names as parameters of the method affects the values **call-next-method** passes to the method it calls.

Values:

The function **call-next-method** returns the value or values returned by the method it calls.

call-next-method

Remarks:

Further computation is possible after **call-next-method** returns.

The function **call-next-method** has lexical scope and indefinite extent.

The function **next-method-p** can be used to test whether there is a next method.

If **call-next-method** is used in methods that do not support it, an error is signaled.

See Also:

"Method Selection and Combination"

"Standard Method Combination"

"Built-in Method Combination Types"

next-method-p

class-name *Standard Generic Function*

Purpose:

The generic function **class-name** takes a class object and returns its name.

Syntax:

class-name *class* [*Generic Function*]

Method Signatures:

class-name (*class* `class`) [*Primary Method*]

Arguments:

The *class* argument is a class object.

Values:

The name of the given class is returned.

Remarks:

The name of an anonymous class is **nil**.

If S is a symbol such that S =`(class-name` C`)` and C =`(find-class` S`)`, then S is the proper name of C. For further discussion, see the section "Classes."

See Also:

"Classes"

find-class

class-of *Function*

Purpose:

The function **class-of** returns the class of which the given object is an instance.

Syntax:

class-of *object* [*Function*]

Arguments:

The argument to **class-of** may be any ISO Lisp object.

Values:

The function **class-of** returns the class of which the argument is an instance.

defclass *Macro*

Purpose:

The macro **defclass** defines a new named class. It returns the new class object as its result.

The syntax of **defclass** provides options for specifying initialization arguments for slots, for specifying default initialization values for slots, and for requesting that methods on specified generic functions be automatically generated for reading and writing the values of slots. No reader or writer functions are defined by default; their generation must be explicitly requested.

Defining a new class also causes a type of the same name to be defined. The predicate `(typep` *object class-name*`)` returns true if the class of the given object is *class-name* itself or a subclass of the class *class-name*. A class object can be used as a type specifier. Thus `(typep` *object class*`)` returns true if the class of the *object* is *class* itself or a subclass of *class*.

Syntax:

defclass *class-name* ({*superclass-name*}*) ({*slot-specifier*}*)[[↓ *class-option*]]

class-name::= *symbol*

superclass-name::= *symbol*

slot-specifier::= *slot-name* | (*slot-name* [[↓ *slot-option*]])

slot-name::= *symbol*

slot-option::= {`:reader` *reader-function-name*}* |
{`:writer` *writer-function-name*}* |
{`:accessor` *reader-function-name*}* |
{`:initform` *form*} |

reader-function-name::= *symbol*

writer-function-name::= *function-specifier*

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

class-option::= (`:metaclass` *class-name*)

Figure 2–1. Syntax for defclass

Arguments:

The *class-name* argument is a non-**nil** symbol. It becomes the proper name of the new class.

Each *superclass-name* argument is a non-**nil** symbol that specifies a direct superclass of the new class. The new class will inherit slots and methods from each of its direct superclasses, from their direct superclasses, and so on. See the section "Inheritance" for a discussion of how slots and methods are inherited.

Each *slot-specifier* argument is the name of the slot or a list consisting of the slot name followed by zero or more slot options. The *slot-name* argument is a symbol that is syntactically valid for use as a ISO Lisp variable name. If there are any duplicate slot names, an error is signaled.

The following slot options are available:

- The **:reader** slot option specifies that an unqualified method is to be defined on the generic function named *reader-function-name* to read the value of the given slot. The *reader-function-name* argument is a non-**nil** symbol. The **:reader** slot option may be specified more than once for a given slot.

- The **:writer** slot option specifies that an unqualified method is to be defined on the generic function named *writer-function-name* to write the value of the slot. The *writer-function-name* argument is a function specifier. The **:writer** slot option may be specified more than once for a given slot.

- The **:accessor** slot option specifies that an unqualified method is to be defined on the generic function named *reader-function-name* to read the value of the given slot and that an unqualified method is to be defined on the generic function named `(setf` *reader-function-name*`)` to be used with **setf** to modify the value of the slot. The *reader-function-name* argument is a non-**nil** symbol. The **:accessor** slot option may be specified more than once for a given slot.

- The **:initform** slot option is used to provide a default initial value form to be used in the initialization of the slot. The **:initform** slot option may be specified once at most for a given slot. This form is evaluated every time it is used to initialize the slot. The lexical environment in which this form is evaluated is the lexical environment in which the **defclass** form was evaluated. Note that the lexical environment refers both to variables and to functions. The dynamic environment is the dynamic environment in which **make-instance** was called. See the section "Object Creation and Initialization."

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

Each class option is an option that refers to the class as a whole or to all class slots. The following class option is available:

- The **:metaclass** class option is used to specify that instances of the class being defined are to have a different metaclass than the default provided by the system (the class <**standard-class**>). The *class-name* argument is the name of the desired metaclass. The **:metaclass** class option may be specified once at most.

Values:

The new class object is returned as the result.

Remarks:

Note the following rules of **defclass** for standard classes:

- It is not required that the superclasses of a class be defined before the **defclass** form for that class is evaluated.
- All the superclasses of a class must be defined before an instance of the class can be made.
- A class must be defined before it can be used as a parameter specializer in a **defmethod** form.

The Object System may be extended to cover situations where these rules are not obeyed.

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

The options to **defclass** can be extended. It is required that all implementations signal an error if they observe a class option or a slot option that is not implemented.

It is valid to specify more than one reader, writer, accessor, or initialization argument for a slot. No other slot option may appear more than once in a single slot description, or an error is signaled.

If no reader, writer, or accessor is specified for a slot, the slot cannot be accessed.

See Also:

"Classes"

"Inheritance"

"Determining the Class Precedence List"

"Object Creation and Initialization"

make-instance

initialize-instance

defgeneric *Macro*

Purpose:

The macro **defgeneric** is used to define a generic function or to specify options and declarations that pertain to a generic function as a whole.

If `(fboundp` *function-specifier*`)` is **nil**, a new generic function is created. If `(symbol-function` *function-specifier*`)` is a generic function or *function-specifier* names a non-generic function, a macro, or a special form, an error is signaled.

Each *method-description* defines a method on the generic function. The lambda-list of each method must be congruent with the lambda-list specified by the *lambda-list* option. If this condition does not hold, an error is signaled. See the section "Congruent Lambda-Lists for All Methods of a Generic Function" for a definition of congruence in this context.

The macro **defgeneric** returns the generic function object as its result.

Syntax:

defgeneric *function-specifier lambda-list* [[↓ *option* | *method-description**]] [*Macro*]

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

lambda-list::= ({*var*}*)

option::= `(:method-combination` *symbol*`)` |
`(:generic-function-class` *class-name*`)` |
`(:method-class` *class-name*`)`

method-description::= `(:method` {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*`)`

method-qualifier::= *non-nil-atom*

specialized-lambda-list::= ({*var* | (*var parameter-specializer-name*)}*
`[&rest` *var*`]`)

parameter-specializer-name::= *symbol*

method-qualifier::= *non-nil-atom*

Arguments:

The *function-specifier* argument is a non-**nil** symbol or a list of the form `(setf` *symbol*`)`.

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

The following options are provided. A given option may occur only once, or an error is signaled.

- The **:generic-function-class** option may be used to specify that the generic function is to have a different class than the default provided by the system (the class <**standard-generic-function**>). The *class-name* argument is the name of a class that can be the class of a generic function.
- The **:method-class** option is used to specify that all methods on this generic function are to have a different class from the default provided by the system (the class <**standard-method**>). The *class-name* argument is the name of a class that is capable of being the class of a method.
- The **:method-combination** option is followed by a symbol that names a type of method combination.

The *method-description* arguments define methods that will be associated with the generic function. The *method-qualifier* and *specialized-lambda-list* arguments in a method description are the same as for **defmethod**.

The *form* arguments specify the method body. The body of the method is enclosed in an implicit block. If *function-specifier* is a symbol, this block bears the same name as the generic function. If *function-specifier* is a list of the form `(setf` *symbol*`)`, the name of the block is *symbol*.

Values:

The generic function object is returned as the result.

Remarks:

The effect of the **defgeneric** macro is as if the following three steps were performed: first, methods defined by previous **defgeneric** forms are removed; second, the generic function is created if needed; and finally, methods specified by the current **defgeneric** form are added to the generic function.

If no method descriptions are specified and a generic function of the same name does not already exist, a generic function with no methods is created.

The *lambda-list* argument of **defgeneric** specifies the shape of lambda-lists for the methods on this generic function. All methods on the resulting generic function must have lambda-lists that are congruent with this shape. If a **defgeneric** form is evaluated and some methods for that generic function have lambda-lists that are not congruent with that given in the **defgeneric** form, an error is signaled. For further details on method congruence, see "Congruent Lambda-Lists for All Methods of a Generic Function"

Implementations can extend **defgeneric** to include other options. It is required that an imple-

mentation signal an error if it observes an option that is not implemented locally.

See Also:

"Congruent Lambda-Lists for All Methods of a Generic Function"

defmethod

defmethod *Macro*

Purpose:

The macro **defmethod** defines a method on a generic function.

If `(fboundp` *function-specifier*`)` is **nil**, a generic function is created with default values for the argument precedence order (each argument is more specific than the arguments to its right in the argument list), for the generic function class (the class <**standard-generic-function**>), for the method class (the class <**standard-method**>), and for the method combination type (the standard method combination type). The lambda-list of the generic function is congruent with the lambda-list of the method being defined. If *function-specifier* names a non-generic function, a macro, or a special form, an error is signaled.

If a generic function is currently named by *function-specifier*, where *function-specifier* is a symbol or a list of the form `(setf` *symbol*`)`, the lambda-list of the method must be congruent with the lambda-list of the generic function. If this condition does not hold, an error is signaled. See the section "Congruent Lambda-Lists for All Methods of a Generic Function" for a definition of congruence in this context.

Syntax:

defmethod *function-specifier* {*method-qualifier*}* *specialized-lambda-list* [*Macro*]

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

method-qualifier::= *non-nil-atom*

specialized-lambda-list::= ({*var* | (*var parameter-specializer-name*)}*
`[&rest` *var*`]`)

parameter-specializer-name::= *symbol*

Arguments:

The *function-specifier* argument is a non-**nil** symbol or a list of the form `(setf` *symbol*`)`. It names the generic function on which the method is defined.

Each *method-qualifier* argument is an object that is used by method combination to identify the given method. A method qualifier is a non-**nil** atom. The method combination type may further restrict what a method qualifier may be. The standard method combination type allows for unqualified methods or methods whose sole qualifier is the keyword **:before**, the keyword **:after**, or the keyword **:around**.

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

The *form* arguments specify the method body. The body of the method is enclosed in an implicit block. If *function-specifier* is a symbol, this block bears the same name as the generic function. If *function-specifier* is a list of the form (`setf` *symbol*), the name of the block is *symbol*.

Values:

The result of **defmethod** is the method object.

Remarks:

The class of the method object that is created is that given by the method class option of the generic function on which the method is defined.

If the generic function already has a method that agrees with the method being defined on parameter specializers and qualifiers, **defmethod** replaces the existing method with the one now being defined. See the section "Agreement on Parameter Specializers and Qualifiers" for a definition of agreement in this context.

The parameter specializers are derived from the parameter specializer names as described in the section "Introduction to Methods."

The expansion of the **defmethod** macro "refers to" each specialized parameter. This includes parameters that have an explicit parameter specializer name of <**object**>. This means that a compiler warning does not occur if the body of the method does not refer to a specialized parameter. Note that a parameter that specializes on <**object**> is not synonymous with an unspecialized parameter in this context.

See Also:

"Introduction to Methods"

"Congruent Lambda-Lists for All Methods of a Generic Function"

"Agreement on Parameter Specializers and Qualifiers"

find-class *Function*

Purpose:

The function **find-class** returns the class object named by the given symbol in the given environment.

Syntax:

find-class *symbol* `&optional` *errorp* [*Function*]

Arguments:

The first argument to **find-class** is a symbol.

If there is no such class and the *errorp* argument is not supplied or is non-**nil**, **find-class** signals an error. If there is no such class and the *errorp* argument is **nil**, **find-class** returns **nil**. The default value of *errorp* is **t**.

Values:

The result of **find-class** is the class object named by the given symbol.

Remarks:

The class associated with a particular symbol can be changed by using **setf** with **find-class**. The results are undefined if the user attempts to change the class associated with a symbol that is defined as a type specifier by *ISO Lisp*. See the section "Integrating Types and Classes."

initialize-instance *Standard Generic Function*

Purpose:

The generic function **initialize-instance** is called by **make-instance** to initialize a newly created instance. The generic function **initialize-instance** is called with the new instance and the defaulted initialization arguments.

The system-supplied primary method on **initialize-instance** initializes the slots of the instance with values according to the initialization arguments and the **:initform** forms of the slots.

Syntax:

initialize-instance *instance* [*Generic Function*]

Method Signatures:

initialize-instance (*instance* <standard-object>) [*Primary Method*]

Arguments:

The *instance* argument is the object to be initialized.

Values:

The modified instance is returned as the result.

Remarks:

Programmers can define methods for **initialize-instance** to specify actions to be taken when an instance is initialized. If only **:after** methods are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of **initialize-instance**.

See Also:

"Object Creation and Initialization"

make-instance

slot-boundp

make-instance *Standard Generic Function*

Purpose:

The generic function **make-instance** creates and returns a new instance of the given class.

The generic function **make-instance** may be used as described in the section "Object Creation and Initialization."

Syntax:

make-instance *class* [*Generic Function*]

Method Signatures:

make-instance (*class* <`standard-class`>) [*Primary Method*]

make-instance (*class* `symbol`) [*Primary Method*]

Arguments:

The *class* argument is a class object or a symbol that names a class.

If the second of the above methods is selected, that method invokes **make-instance** on the arguments (`find-class` *class*).

The initialization arguments are checked within **make-instance**. See the section "Object Creation and Initialization."

Values:

The new instance is returned.

See Also:

"Object Creation and Initialization"

defclass

initialize-instance

class-of

next-method-p *Function*

Purpose:

The locally defined function **next-method-p** can be used within the body of a method defined by a method-defining form to determine whether a next method exists.

Syntax:

next-method-p [*Function*]

Arguments:

The function **next-method-p** takes no arguments.

Values:

The function **next-method-p** returns true or false.

Remarks:

Like **call-next-method**, the function **next-method-p** has lexical scope and indefinite extent.

See Also:

call-next-method

slot-boundp *Function*

Purpose:

The function **slot-boundp** tests whether a specific slot in an instance is bound.

Syntax:

slot-boundp *instance slot-name* [*Function*]

Arguments:

The arguments are the instance and the name of the slot.

Values:

The function **slot-boundp** returns true or false.

Remarks:

The function **slot-boundp** allows for writing **:after** methods on **initialize-instance** in order to initialize only those slots that have not already been bound.

slot-exists-p *Function*

Purpose:

The function **slot-exists-p** tests whether the specified object has a slot of the given name.

Syntax:

slot-exists-p *object slot-name* [*Function*]

Arguments:

The *object* argument is any object. The *slot-name* argument is a symbol.

Values:

The function **slot-exists-p** returns true or false.