The Meta Class System MCS

A Portable Object System for Common Lisp

Version 1.3

– Documentation –

Harry Bretthauer

Jürgen Kopp

Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD)
Forschungsgruppe Expertensysteme
Schloß Birlinghoven
5205 Sankt Augustin 1

June 1991

Contents

1 Introduction

1.1 History

This paper describes an object-oriented extension to Common Lisp called the Meta Class System (MCS). MCS has been developed during the last three years. We based our first version on ObjVlisp of Pierre Cointe [Coi88], Micro Flavor System (MFS) of Thomas Christaller [Chr88] and Micro Common Flavors (MCF) of Franco di Primio [dP88] following the message passing paradigm. In successive versions, major revisions have been made moving MCS from message passing to generic functions and integrating the functionality of CLOS, the Common Lisp Object System [BDG+88], and TEΛOΣ, the object system of LE-LISP version 16 [ILO91] and EULISP [PN91]. Additionally, MCS provides a metaobject protocol which the user can specialize. Some of the earlier versions have also been ported to Scheme [Hay89].
MCS has been used successfully for the development of the commercial software product babylon, which is an expert system development workbench, including a hybrid knowledge representation language and many tools like editors, browsers, etc. Based on experiences made building such a large and complex software system MCS has been improved on abstract classes and mixin classes to support a better style of object-oriented modelling [BCK89]. Furthermore, these improvements allow significant internal optimizations.
Here, we describe the current state of the latest version that is still under development. Nevertheless, major parts of MCS have achieved a stable state, at least from the application programmers point of view. MCS is more efficient in time and space than comparable systems we know, including commercial CLOS implementations.
The paper is divided into five sections. The first section describes the underlying language concepts including compatibility with CLOS. The next three sections go into details about the application programming, system programming and the programming environment. Technical differences between MCS and CLOS are listed in the appendix.
The reader of this paper should be familiar with Lisp [SJ90] and with object-oriented programming, e.g. CLOS [Kee89] or TEΛOΣ.

1.2 Availability

The sources of MCS are available as freeware via anonymous FTP from host gmdzi.gmd.de (IP address: 129.26.1.90) in the directory /pub/lisp/mcs .
MCS is a portable Common Lisp program; its source code requires 321 KB, the binaries need ca. 330 KB for Macintosh Allegro Common Lisp, ca. 530 KB for Allegro Common Lisp on a Sun SPARC-Station, and ca. 500 KB for Lucid Common Lisp on a Sun SPARC-Station.

1.3 Acknowledgments

First, we want to thank all authors and users of papers, books, and software systems concerning LISP and object-oriented programming (ObjVlisp, Flavors, MFS, MCF, CLOS, and TEΛOΣ), who contributed many ideas to our work.
We would also like to thank all colleagues of the babylon project at GMD and VW-GEDAS. They have been using MCS since two years and made continuously suggestions for improving our concepts and implementation, we would not get otherwise. Jürgen Walther provided a very helpful user interface to support program development for Apple's Macintosh Allegro Common Lisp.
We enjoyed working in the EUREKA project ELSY (EU 157) together with colleagues at Bull, CRIL, GMD, ILOG and INRIA. Especially, Harley Davis and Greg Nuyens gave us useful hints from experience with LE-LISP and TEΛOΣ.
The EULISP working group was an excellent forum to discuss and learn much about the programming language LISP, its design and its implementation.
Furthermore, we thank Thomas Christaller, Jürgen Christoffel, Tom Gordon, Wolfgang Gräter, Erich Rome, Angi Voß, and Ralf Zimmer for their thoughtful reviews of this paper.

2 Language Concepts

MCS was developed with strong respect of language design principles of ObjVlisp to have a minimal kernel, which is uniform, reflective, and extendible. It took from CLOS[1] and TEΛOΣ the idea to integrate function application and message passing into generic functions.
The key concept that distinguishes object-oriented programming languages from languages supporting abstract data types is called inheritance. In MCS inheritance has a well-defined semantics to serve a well known modelling technique called specialization and generalization, i.e. commonalities and differences of objects are organized by answering the question: Which properties of objects are more general and which are more specific?
This view of inheritance is not taken as a matter of course [Str88]. The old debate about what inheritance is or isn't [Bra83] is still alive. Rooting in AI, especially in knowledge representation, the metaphor of inheritance has been approved for "standard" programming languages, e.g. C++. Many insights are going to be reinvented by a different community from a different point of view. So far, more attention has been paid for code reusing than for programs reflecting more explicitly underlying models in a problem domain. Corresponding requirements of inheritance related language constructs and their usage appears sometimes strange from the modelling point of view.
A typical example demonstrating that is an integer stack defined using multiple inheritance:

> Assuming there are classes `stack` and `integer`, the class `integer-stack`, i.e. a stack with integer entries, is defined inheriting from `integer` and `stack`.
>
> While it seems "naturally" to view an integer stack as a special stack, it is "confusing" to view it as a special integer.
> In this example, inheritance serves two different modelling techniques: specialization and aggregation.

Requirements of different programming tasks have been analyzed to classify specific language constructs. Thus, MCS provides three interfaces: the application programmer interface, the system programmer interface and the programming environment interface.
The application programmer interface is intended for simple, nevertheless powerful object-oriented programming using the interface as it is. The system programmer interface provides the functionality for defining the application programmer interface, as well as for defining new interfaces or language extensions specializing MCS itself. It is also called the metaprogramming interface (Metaobject Protocol in CLOS), i.e. the output of a program will be used to write another program. It also includes features needed to implement the programming environment, which is separated as an interface of its own. The distinction of requirements for program development and application is indispensable to achieve efficiency in complex software systems.
In the following sections we describe the main concepts of MCS in more detail.

2.1 Objects

In MCS, every object is an instance of a specific class. Classes are first-class objects.

2.2 Classes and Inheritance

A class represents a set of objects, called its instances. A class defines the structure of its instances, i.e. which set of slots each instance contains. A class also defines the behavior of their instances, i.e. a set of methods specialized on the class. The structure and behavior of a class can be specialized defining subclasses.
The class defining form includes the *class name*, a list of its *superclasses*, a set of *slot specifications* and a set of *class options*.
A class is called a metaclass if its instances are themselves classes.
Classes have to be defined top down, i.e. when defining a class all its superclasses need to be defined already.
Each class name is also a valid Common Lisp type specifier.

[1]Parts of this documentation are taken from the CLOS and EULISP specification .

2.2.1 Classifying Classes

From the implementation point of view, we distinguish built-in (primitive) vs. defined classes, instantiable vs. abstract (non-instantiable) classes, single vs. multiple inherited classes.
From the modelling point of view, we distinguish base classes defining substantive properties like buffer, window, etc. vs. mixin classes defining adjective properties of base classes like printable, editable, or bordered, titled, etc.
How are these orthogonal views related to each other? The implementation distinctions are the basis of some significant optimizations. The modelling distinctions lead to clear and understandable programs. We observed that multiple inheritance is mostly used to let a base class specialize many mixin classes, or a mixin class specialize many mixin classes. The property that base classes are organized in a single inheritance manner when ignoring the mixin classes for a moment, i.e. each base class specializes exactly one other base class, seems to be a strong indicator of well designed class hierarchies.
Mixin classes may not be instantiated separately, i.e. they are abstract, while some base classes have direct instances. Base classes, as well as mixin classes, are all defined, i.e. not primitive.
Thus, to serve the above modelling and implementation requirements, we provide the following instantiable metaclasses:

- `abstract-base-class` is an `abstract single-inherited defined class`,
- `base-class` is an `instantiable single-inherited defined class`,
- `mixin-class` is an `abstract multiple-inherited defined class`,
- `abstract-built-in-class` is an `abstract built-in class`,
- `built-in-class` is an `instantiable built-in class`.

There is a general constructor `defclass` to define new classes. Its metaclass option determines which kind of class has to be created. By default it is a base class (instance of base-class or a subclass of it). Specific constructors create specific classes: `defabstract` creates abstract base classes (instances of `abstract-base-class` or a subclass of it) and `defmixin` creates mixin classes (instances of `mixin-class` or a subclass of it).

2.2.2 Base Classes

Base classes are intended to represent the most important and rather substantive properties of objects. A base class may be abstract or instantiable. It may specialize at most one other base class (the rightmost one in the list of direct superclasses) and many other mixin classes. For abstract base classes this can only be an abstract base class and for instantiable base classes it could be any base class. All direct superclasses of a base class have to be orthogonal, i.e. neither is a superclass of the other and there exists no class (called join node) which is a superclass of two of them except the root.

2.2.3 Mixin Classes

Mixin classes are intended to represent additional and rather adjective properties of objects. A mixin class may only specialize mixin classes. All the direct superclasses of a mixin class have to be orthogonal (see base classes), i.e. there is no join node except the root.

2.2.4 Built-in Classes

For each of the following predefined types of Common Lisp there exists a built-in class in MCS. The built-in classes (or types) define a hierarchy as shown in the following figure.
All names of built-in and user defined classes can be used as type names in `typep`. In order to avoid redefinition of Common Lisp system functions, MCS provides special functions like `obj-typep`, `obj-type-of` and `obj-subtypep` to handle built-in and defined classes.

2.2.5 Controlling Inheritance

Each class has a class precedence list, which is used for controlling the inheritance of slots and methods. The class precedence list contains the class and all its superclasses. Since join nodes are not allowed the class precedence list is computed by concatenating the class precedence lists of the direct superclasses of a class.

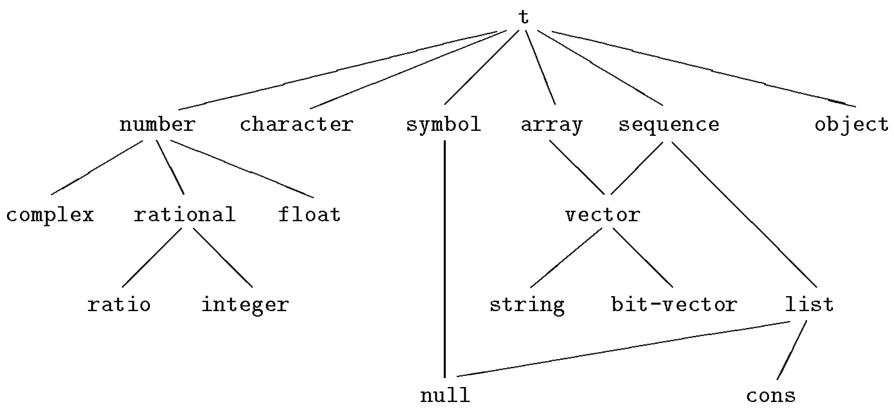

Figure 1: Specialization hierarchy of built-in classes.

2.3 Slots

The components of an object are called slots. Each slot of an object is associated with a slot definition in the objects class. Slots are inherited, this means that an instance of a class will contain all the slots defined directly in the class as well as all of those defined in the class's superclasses. For slots of the same name the most specific default initial value is taken and all given types are collected to represent the type of the slot.
In MCS, all slots are local. Furthermore, if an initialization argument is specified its value has to be the same symbol as the slot name. So the `:initarg` option may only appear once, otherwise an error is signaled.

2.4 Generic Functions

A generic function is a function whose behavior is determined by the classes of its arguments. Each potential behavior is defined by a method.
Generic functions replace the `send` construct of other object oriented languages based on message passing, e.g. Flavors [WM81], CommonLoops [BKK+86], MFS [Chr88], and MCF [dP88].
MCS does not support local generic functions.
In the current version of MCS only the standard method combination of CLOS is implemented. Even though it is possible to define new method combinations, we do not recommend to do so. In the context of inheritance, we prefer to support behavior specialization rather than behavior composition (combination) which is orthogonal to specialization. We suggest to define only primary methods and to use `call-next-method` for calling the more general method from inside a more specific one.

2.4.1 Methods

A method determines the behavior of a generic function for a particular sequence of classes, called the method's signature or specializers. Methods are not functions themselves; they only contain pieces of code for a generic function.
Like slots, methods are inherited. This means, if a method is applicable for a class `C1` it is also applicable for all subclasses of `C1`. For subclasses, new methods may be defined; they are called more specific than those of the superclasses. However, the more general method is accessible from the more specific through the `call-next-method` form. So the behavior can be inherited and specialized in subclasses.
There are some restrictions for the definition of a method:

- The lambda-list of a method can only mention `&rest`. In MCS `&optional`, `&aux`, `&key` and `&allow-other-keys` are not supported.
- In the *specialized-lambda-list* of a method a specialized parameter is a list (*variable-name parameter-specializer-name*), where *parameter-specializer-name* has to be a class name, i.e. (`eql` *form*) is not

allowed in MCS.

- Since there is only the standard method combination the auxiliary methods have only one single qualifier of `:around`, `:before`, or `:after`.
- Methods in MCS do not support multiple values from Common Lisp.
- It is not allowed to rebind lambda variables of a method in its method body.

2.5 Creating Instances

Instances of a class are created with the function `make-instance`. Additionally, there exists a macro called `make-instance-fast` that optimizes the creation of instances, if the class object and the initargs are known at compile time.

2.6 Compatibility to CLOS

The application programmer interface of MCS is mainly a sublanguage of CLOS with some restrictions and a few extensions.
All restrictions are motivated by goals like

- simple and understandable programming style
- efficiency and portability of MCS and its applications
- balancing supply and demand of language features.

Besides these few extensions, which can be easily mapped to CLOS (see section 3.5), every MCS program runs also in a CLOS implementation. The system programmer interface is similar to the draft of the metaobject protocol (MOP) of CLOS. The appendix contains a collection of all restrictions and extensions to CLOS.

3 Application Programmer Interface

This section describes the interface for programming applications using simple object-oriented features of MCS, i.e. avoiding metaobject programming.

3.1 Expressions

3.1.1 Definitions

(**defclass** *class-name* (*superclasses*) (*slot-specification**) *class-option**) → *undefined* **Macro**

class-name ::= *symbol*
superclasses ::= *mixin-class-name** [*base-class-name*]
mixin-class-name ::= *symbol*
base-class-name ::= *symbol*
slot-specification ::= *slot-name* | (*slot-name slot-option**)
slot-name ::= *symbol*
slot-option ::= {`:accessor` *reader-function-name*}* |
{`:reader` *reader-function-name*}* |
{`:writer` *writer-function-name*}* |
{`:initform` *form*} |
{`:initarg` *slot-name*} |
{`:type` *type-specifier*} |
{`:documentation` *string*}
reader-function-name ::= *symbol*
writer-function-name ::= *symbol*
class-option ::= (`:documentation` *string*)

The macro `defclass` defines a new base class named *class-name*. If a class with the same name already exists, the redefinition is ignored and MCS signals a warning.[2]

The syntax of `defclass` provides options for specifying initialization arguments for slots, for specifying default initial values for slots, and for requesting that methods on specified generic functions be automatically generated for reading and writing the values of slots. Reader or writer functions are not defined by default, their generation must be requested explicitly.

The Definition of a new class causes the definition of a new type of the class's name. The predicate `(obj-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*. The predicate `(typep` *object class-name*`)` returns the same but works more slowly.

Each *mixin-class-name* and the *base-class-name* argument specifies a direct superclass. The *base-class-name* has to be the name of a base class. The new class will inherit slots and methods from each of its direct superclasses, from their direct superclasses, and so on. The superclasses of a class must be defined prior to the evaluation of the class definition. Furthermore a class must be defined before it can be used as a parameter specializer in a `defmethod` form.

Each *slot-specification* 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. 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

[2]This is the default behavior of MCS which can be changed within the programming environment. See section 5.3.

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 during the initialization of the slot. The `:initform` slot option may be specified at most once 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 to both variables and functions. The dynamic environment is the dynamic environment in which `make-instance` was called.

- The `:initarg` slot option declares an initialization argument named as the slot and specifies that this initialization argument may initialize the given slot. If the initialization argument is used in a call to `make-instance`, its value will be stored in the given slot, and the slot's `:initform` slot option, if any, is not evaluated. If the initialization argument specified for a given slot is not used, then the slot is initialized according to the `:initform` slot option, if specified, or will be unbound. The `:initarg` slot option can be specified only once for a given slot. The initialization argument has to be identical to *slot-name*.[3]

- The `:type` slot option specifies that the contents of the slot should always be of the specified data type. MCS does not check the type when writing the slot.

- The `:documentation` slot option provides a documentation string for the slot.

If reader, writer, or accessor is not specified for a slot, the slot can only be accessed by the macro `slot-value`.
Slot specifications of MCS do not support the slot option `:allocation`, i.e. shared slots[4] are not available.

A class option refers to the class as a whole. Within the application programmer interface there is only a single class option:

- The `:documentation` class option causes a documentation string to be attached to the class name. The documentation type of this string is `type`. The form `(documentation` *class-name* `'type)` may be used to retrieve the documentation string. The `:documentation` class option may be specified at most once.

In MCS there exists no class option `:default-initargs`.
See also: `defabstract, defmixin, slot-value, make-instance, initialize-instance`.

(**defabstract** *class-name* (*superclasses*) (*slot-specification**) *class-option**) → *undefined* **Macro**

class-name ::= *symbol*
superclasses ::= *mixin-class-name** [*abstract-class-name*]
mixin-class-name ::= *symbol*
superclass-name ::= *symbol*

The macro `defabstract`[5] defines a new abstract base class named *class-name*. If a class with the same name already exists, the redefinition is ignored and MCS signals a warning.[2] Each *mixin-class-name* has to be the name of a mixin class and *abstract-class-name* has to be the name of a non-instantiable base class (i.e. an abstract class) if it is specified. All superclasses have to be defined prior to the abstract class. *Slot-specification* and *class-option* have the same meaning as in `defclass`.
See also: `defclass, defmixin`.

(**defmixin** *class-name* (*superclasses*) (*slot-specification**) *class-option**) → *undefined* **Macro**

class-name ::= *symbol*
superclasses ::= *mixin-class-name**
mixin-class-name ::= *symbol*

The macro `defmixin`[5] defines a new mixin class named *class-name* and returns the new class object as its result. If a class with the same name already exists, the redefinition is ignored and MCS signals a warning.[2]

[3] In CLOS you can define more than one `:initarg` that can be any symbol.
[4] Use class variables or communication objects instead!
[5] This function does not exist in CLOS.

Each *mixin-class-name* has to be the name of a mixin class if it is specified. All superclasses have to be defined prior to the mixin class.
Slot-specification and *class-option* have the same meaning as in `defclass`.
See also: `defabstract, defclass`.

(**defconstructor** *constructor-name* (*variable**) [*documentation*] *form**) → *undefined* **Macro**

The macro `defconstructor`[5] should be used to define constructors of classes. It is applied like `defun`. Within the *form* arguments `make-instance` should be used to create instances. The constructor should be defined after the associated class definition and its `initialize-instance` methods for potential optimizations.

(**defgeneric** *function-specifier lambda-list gfn-option**) → *undefined* **Macro**

function-specifier ::= {*symbol* | `(setf` *symbol*`)`}
lambda-list ::= ({*var*}* [`&rest` *var*])
gfn-option ::= (*:method-combination symbol*) |
(*:method-class class-name*) |
(*:documentation string*)

The macro `defgeneric` is used to define a generic function and 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, the redefinition is ignored and MCS signals a warning. If *function-specifier* names a non-generic function, a macro, or a special form, an error is signaled.
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 with the exception that only `&rest` is allowed as a keyword.
The following generic function options are provided. A given option should occur at most once.

- 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 `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. In the current version only `standard` is available.

- The `:documentation` argument associates a documentation string with the generic function. The documentation type of this string is `function`. The form `(documentation` *function-specifier* `'function)` may be used to retrieve this string.

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.
See also: `defmethod`.

(**defmethod** *function-specifier* [*qualifier*] *specialized-lambda-list* [*doc*] *form**) → *undefined* **Macro**

function-specifier ::= {*symbol*} | `(setf` *symbol*`)`
qualifier ::= `:before` | `:after` | `:around`
specialized-lambda-list ::= ({*var* | (*var parameter-specializer-name*)}* [`&rest` *var*])
parameter-specializer-name ::= *symbol*
doc ::= *string*

The macro `defmethod` defines a method on a generic function.
If `(fboundp` *function-specifier*`)` is `nil`, a generic function is created with `method` and `standard` as default values for method class and method combination respectively. 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. Otherwise, an error is signaled.

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 being defined.
Each *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 the range of method qualifiers. The standard method combination type allows for unqualified (primary) methods or methods whose sole qualifier is the keyword `:before`, `:after`, or `: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 and that only `&rest` is allowed. 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 parameter specializer name is not specified for a given required parameter, the parameter specializer defaults to the class named `t`.
The *form* arguments specify the method body.
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.
Within a method body it is not allowed to rebind variables with the same names as the required parameters of the method.
See also: `defgeneric, call-next-method`.

3.1.2 Specializing Methods

(call-next-method) → *object* **Macro**

The macro `call-next-method`[6] can be used within the body of a method defined by a method-defining form to call the next method.
The macro `call-next-method` returns the value returned by the method it calls. If there is no next method, the generic function `no-next-method` is called.
The type of method combination used determines which methods may 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 macro `call-next-method` may not 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. The generic function `no-next-method` is called 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 macro `call-next-method` may not be used in `:after` methods.

Further computations are possible after `call-next-method` returns. The macro `next-method-p` can be used to test whether there is a next method.
See also: `defmethod, next-method-p`.

(next-method-p) → *boolean* **Macro**

The locally defined macro `next-method-p`[7] can be used within the body of a method defined by a method-defining form to determine whether a next method exists. It takes no arguments and returns true or false.
See also: `defmethod, call-next-method`.

[6] In CLOS `call-next-method` is a function with a `&rest` argument.
[7] In CLOS `next-method-p` is a function.

3.2 Objects

3.2.1 Operations

(**initialize-instance** *object* `&rest` *initargs*) → *object* **Generic Function**

`initialize-instance` (*object* `object`) `&rest` *initargs* **Method**

The generic function `initialize-instance` is called by `make-instance` to initialize a newly created instance, which is returned as value.

The *object* argument is the object to be initialized. The *initargs* argument is a possible empty sequence of alternating initialization argument names and values.

The system-supplied primary method on `initialize-instance` initializes all slots of the instance in the following way. Each slot with a supplied initialization argument is initialized according to the initialization argument value. The other slots are initialized according to the default initial value forms (`:initform`), if specified for the slots, or will be unbound. The method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid.

Programmers can define methods on `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: `make-instance, slot-boundp, slot-makunbound`.

(**reinitialize-instance** *object* `&rest` *initargs*) → *object* **Generic Function**

`reinitialize-instance` (*object* `object`) `&rest` *initargs* **Method**

The generic function `reinitialize-instance` can be used to change the values of slots according to initialization arguments. This generic function is called by the Meta-Object Protocol. It can also be called by users.

The *object* argument specifies the object to be initialized and the *initargs* argument is a possible empty sequence of alternating initialization argument names and values. As its result, it returns the modified object.

The system-supplied primary method for `reinitialize-instance` checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid.

See also: `initialize-instance, slot-boundp, slot-makunbound`.

(**slot-value** *object slot-name*) → *object* **Macro**

The macro `slot-value`[8] returns the value contained in the slot *slot-name* of the given object *object*. If there is no slot with that name, `slot-missing` is called. If the slot is unbound, `slot-unbound` is called.

The macro `setf` can be used with `slot-value` to change the value of a slot.

MCS may optimize `slot-value` within methods.

See also: `slot-missing, slot-unbound, slot-makunbound`.

(**slot-boundp** *object slot-name*) → *boolean* **Function**

The function `slot-boundp` tests whether a specific slot of an instance is bound. It takes an object and the name of a slot and returns true or false.

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.

If the object does not have a slot of the given name, `slot-missing` is called.

(**slot-makunbound** *object slot-name*) → *object* **Function**

The function `slot-makunbound` resets a slot of an object to the unbound state. It takes an instance and the name of a slot and returns the instance.

If the object does not have a slot of the given name, `slot-missing` is called.

(**obj-describe** *object*) → *undefined* **Generic Function**

`obj-describe` (*object* `object`) **Method**

`obj-describe` (*object* `t`) **Method**

The generic function `obj-describe`[9] prints information about a given object on the standard output. The

[8] In CLOS `slot-value` is a function.

[9] The function has the prefix `obj-` in order to avoid the redefinition of a Common Lisp system function.

object argument may be any Common Lisp object. Programmers may define methods for their own classes if they do not wish to inherit the standard method of `object`. The generic function `obj-describe` does not return any value.

(**print-object** *object stream*) → *object* **Generic Function**
`print-object` (*object* `object`) *stream* **Method**
`print-object` (*object* `method`) *stream* **Method**
The generic function `print-object` writes the printed representation of an object to a stream. The function `print-object` is called by the print system; it should not be called by the user. Users can write methods for `print-object` for their own classes if they do not wish to inherit an implementation-supplied method.
`print-object` takes a Lisp object and a stream - it cannot be `t` or `nil` - and returns *object*.

(**obj-typep** *object type*) → *boolean* **Function**
The function `obj-typep`[9] takes an object and a type specifier and returns `t` if *object* is a subtype of *type*. `typep` works too, but more slowly.

(**obj-type-of** *object*) → *type* **Function**
The function `obj-type-of`[9] takes an object and returns its type (class-name).

3.3 Classes

3.3.1 Creating Instances

(**make-instance** *class-name* `&rest` *initargs*) → *object* **Function**
The function `make-instance` creates and returns a new instance of the given class.
The *class-name* argument is a symbol that names a class. The remaining arguments form a possible empty sequence of alternating initialization argument names and values.
The initialization arguments are checked within `make-instance`.
See also: `initialize-instance`.

3.4 A short Example

```
;;; define an abstract class graphical-object

(defabstract graphical-Object () ())


;;; define a base class point

(defclass Point (graphical-Object)
  ((x :initform 0
      :initarg x
      :accessor point-x
      :type     integer)
   (y :initform 0
      :initarg y
      :accessor point-y
      :type     integer)))


;;; the protocol for colored objects

(defgeneric color-of (obj))
(defgeneric change-color (obj color))

;;; define a mixin-class colored

(defmixin colored ()
  ((color :initform 'black :initarg color)))
```

```lisp
;;; default methods for uncolored objects

(defmethod color-of (obj)
  'gray)

(defmethod change-color (obj color)
  (warn "Can't change the color of an uncolored object."))

;;; methods for colored objects

(defmethod color-of ((obj colored))
  (slot-value obj 'color))

(defmethod change-color ((obj colored) color)
  (setf (slot-value obj 'color) color))


;;; the protocol for moveable objects

(defgeneric move-object (obj x y))

;;; define a mixin-class moveable

(defmixin moveable ()
  ((x :accessor object-x :type integer)
   (y :accessor object-y :type integer)))

;;; default methods for unmoveable object

(defmethod move-object (obj x y)
  (warn "Can't move an unmoveable object."))

;;; methods for moveable object

(defmethod move-object ((obj moveable) x y)
  (setf (object-x obj) x)
  (setf (object-y obj) y)
  obj)


;;; define special point classes

(defclass colored-Point (colored Point) ())

(defclass moveable-colored-Point (moveable colored Point) ())


;;; create some instances of point classes

(setf p1 (make-instance 'Point
                         'x 25 'y 10))

(setf p2 (make-instance 'colored-Point
                        'x 1 'y 1 'color 'red))

(setf p3 (make-instance 'moveable-colored-Point
                        'color 'blue))
```

3.5 Mapping the Application Programmer Interface to CLOS

```
(defmacro defabstract (class-name direct-superclasses direct-slots &rest options)
  `(defclass ,class-name ,direct-superclasses
     ,direct-slots
     ,@options))
```

```
(defmacro defmixin (class-name direct-superclasses direct-slots &rest options)
  `(defclass ,class-name ,direct-superclasses
     ,direct-slots
     ,@options))
```

```
(defmacro defconstructor (function-name lambda-list &rest body)
  `(defun ,function-name  ,lambda-list ,@body) )
```

```
(defmethod obj-describe (object)
  (describe object))
```

```
(defun obj-typep (object type)
  (typep object type))
```

```
(defun obj-type-of (object)
  (type-of object))
```

4 System Programmer Interface

This section describes the system programmer interface of MCS. The first section describes the expressions and the following ones list the operations ordered according to the specialization of objects (objects, classes, slot definitions, etc.). The description of the system programmer interface includes the application programmer interface, but we omit the details of items that have already been described in section 3.1.

4.1 Expressions

4.1.1 Definitions

(**defclass** *class-name* (*superclasses*) (*slot-specification**) *class-option**) → *class* **Macro**

class-option ::= (`:documentation` *string*) |
(`:metaclass` *class-name*)

This is almost identical to `defclass` of section 3.1. Additionally, the return value is specified to be the new class object. Furthermore, one more class option is available:

- The `:metaclass` class option is used to specify that instances of the class being defined should have a different metaclass than the default `base-class`. The *class-name* argument is the name of the desired metaclass. The `:metaclass` class option may be specified once at most.

  With `defclass` it is also possible to define abstract or mixin classes by using `abstract-base-class` or `mixin-class` as metaclass respectively.

See also: section 3.1.

(**defabstract** *class-name* (*superclasses*) (*slot-specification**) *class-option**) → *class* **Macro**

Slot-specification and *class-option* have the same meaning as in `defclass`, with the restriction that an argument given to the `:metaclass` option can only be a subclass of `abstract-base-class`.
See also: section 3.1.

(**defmixin** *class-name* (*superclasses*) (*slot-specification**) *class-option**) → *class* **Macro**

Slot-specification and *class-option* have the same meaning as in `defclass`, with the restriction that an argument given to the `:metaclass` option can only be a subclass of `mixin-class`.
See also: section 3.1.

(**defconstructor** *constructor-name* (*variable**) [*documentation*] *form**) → *symbol* **Macro**

See section 3.1.

(**defgeneric** *function-specifier lambda-list gfn-option**) → *function* **Macro**

Here, the same as for `defgeneric` of section 3.1 holds, except that it returns the generic function object as its result.
See also: section 3.1.

(**defmethod** *function-specifier* [*qualifier*] *specialized-lambda-list* [*doc*] *form**) → *function* **Macro**

Here, the same as for `defmethod` of section 3.1 holds, except that it returns the generic function object as its result.
See also: section 3.1.

(**defreader** *reader-name class-name slot-name*) → *function* **Macro**

The function `defreader`[10] defines a reader called *reader-name* for reading the slot *slot-name* of instances of class *class-name*. `defreader` forms are part of the macro expansion of a `defclass` form if readers are specified for some slots.
See also: `make-reader`.

(**defwriter** *writer-name class-name slot-name*) → *function* **Macro**

The function `defwriter`[10] defines a writer called *writer-name* for writing the slot *slot-name* of instances of class *class-name*. `defwriter` forms are part of the macro expansion of a `defclass` form if writers are specified for some slots.
See also: `make-writer`.

[10] This function does not exist in the draft Metaobject Protocol (MOP) of CLOS.

4.1.2 Specializing Methods

(**call-next-method**) → *object* **Macro**

See section 3.1.

(**next-method-p**) → *boolean* **Macro**

See section 3.1.

4.2 Objects

4.2.1 Access Functions

(**class-of** *object*) → *class* **Function**

The function `class-of` returns the class of which *object* is an instance. *Object* may be any Common Lisp object.

(**slot-value** *object slot-name*) → *object* **Macro**

The macro `slot-value`[11] returns the value of the slot *slot-name* of the given object *object*. If there is no slot with that name, `slot-missing` is called. If the slot is unbound, `slot-unbound` is called.
The macro `setf` can be used with `slot-value` to change the value of a slot.
MCS may optimize `slot-value` within methods.
See also: `slot-missing, slot-unbound, slot-makunbound`.

(**slot-boundp** *object slot-name*) → *boolean* **Function**

The function `slot-boundp` tests whether a specific slot of an instance is bound. It takes an object and the name of a slot and returns true or false.
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.
If the object does not have a slot of the given name, `slot-missing` is called.

(**slot-exists-p** *object slot-name*) → *boolean* **Function**

The function `slot-exists-p` tests whether the specified object has a slot of the given name. It takes an object and the name of a slot and returns true or false.
See also: `find-slot`.

(**slot-makunbound** *object slot-name*) → *object* **Function**

The function `slot-makunbound` resets a slot of an object to the unbound state. It takes an instance and the name of a slot and returns the instance.
If the object does not have a slot of the given name, `slot-missing` is called.

4.2.2 Initialization

(**initialize-instance** *object* `&rest` *initargs*) → *object* **Generic Function**

`initialize-instance` (*object* `object`) `&rest` *initargs* **Method**

The generic function `initialize-instance` is called by `make-instance` to initialize a newly created instance, which is returned as value.
The *object* argument is the object to be initialized. The *initargs* argument is a possible empty sequence of alternating initialization argument names and values.
The primary method for `object` initializes all slots of the instance in the following way. Each slot with a supplied initialization argument is initialized according to the initialization argument value. The other slots are initialized according to the default initial value forms (`:initform`), if specified for these slots, or will be unbound. The method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid.
Programmers can define methods on `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`.

[11] In CLOS `slot-value` is a function.

See also: `make-instance, slot-boundp, slot-makunbound`.

(**reinitialize-instance** *object* `&rest` *initargs*) → *object* **Generic Function**

`reinitialize-instance` (*object* `object`) `&rest` *initargs* **Method**

The generic function `reinitialize-instance` can be used to change the values of local slots according to initialization arguments. This generic function is called by the Meta-Object Protocol. It can also be called by users.

The *object* argument is the object to be initialized and the *initargs* argument is a possible empty sequence of alternating initialization argument names and values. As its result, it returns the modified object.

The primary method of `reinitialize-instance` for `object` initializes only those slots for which initialization arguments are supplied. The other slots remain unchanged. The method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid.

See also: `initialize-instance, slot-boundp, slot-makunbound`.

4.2.3 Miscellaneous Functions

(**change-class** *object new-class*) → *object* **Generic Function**

`change-class` (*object* `object`) *new-class* **Method**

The generic function `change-class` changes the class of *object* to a *new-class* if possible, i.e. if the old and the new class are both redefinable. It destructively modifies and returns the object.

A slot of the old class, that is not unbound, keeps its old value if a slot of the same name is present in the new class[12]. If there is no slot of the same name in the new class the slot is discarded. All other slots are initialized according to their default initial value forms (`:initform`).

See also: Updating Instances.

(**obj-describe** *object*) → *undefined* **Generic Function**

`obj-describe` (*object* `object`) **Method**

The generic function `obj-describe`[9] prints information about a given object on the standard output. The *object* argument may be any Common Lisp object. Programmers may define methods for their own classes if they do not wish to inherit the standard method of `object`. The generic function `obj-describe` does not return any value.

(**print-object** *object stream*) → *object* **Generic Function**

`print-object` (*object* `object`) *stream* **Method**

The generic function `print-object` writes the printed representation of an object to a stream. The function `print-object` is called by the print system; it should not be called by the user. Users can write methods for `print-object` for their own classes if they do not wish to inherit an implementation-supplied method. `print-object` takes a Lisp object and a stream - it cannot be `t` or `nil` - and returns *object*.

(**obj-typep** *object type*) → *boolean* **Function**

The function `obj-typep`[9] takes an object and a type specifier and returns `t` if the type of *object* is a subtype of *type*. `typep` works too, but slower.

(**obj-type-of** *object*) → *type* **Function**

The function `obj-type-of`[9] takes an object and returns its type (class-name).

(**obj-copy** *object*) → *object* **Function**

The function `obj-copy`[10] takes an instance of a defined class and returns a flat copy of it. If *object* is an instance of a built-in class an error is signaled.

4.3 Classes

4.3.1 Class Hierarchy

The following pictures depict the system class hierarchy of MCS without the built-in classes referring to the Common Lisp types (see page 3). The dashed boxes in the following picture surround complementary mixin classes, i.e. a class, for example, can either be abstract or instantiable, but not both.

[12] In CLOS an unbound slot remains unbound, even if there is a default initial value form in *new-class* for the slot.

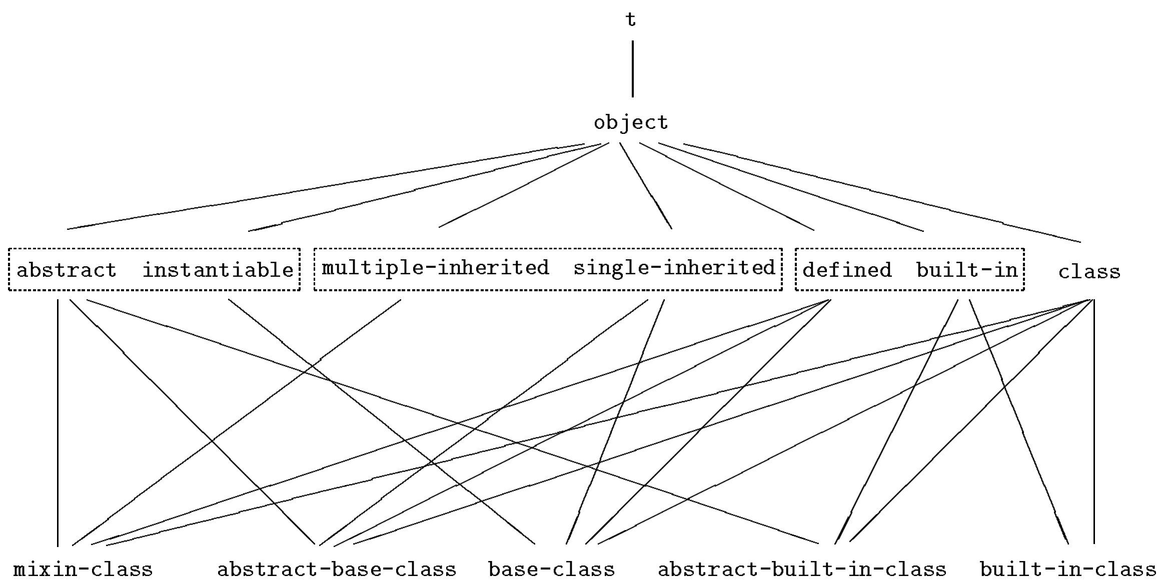

Figure 2: Specialization hierarchy of metaclasses.

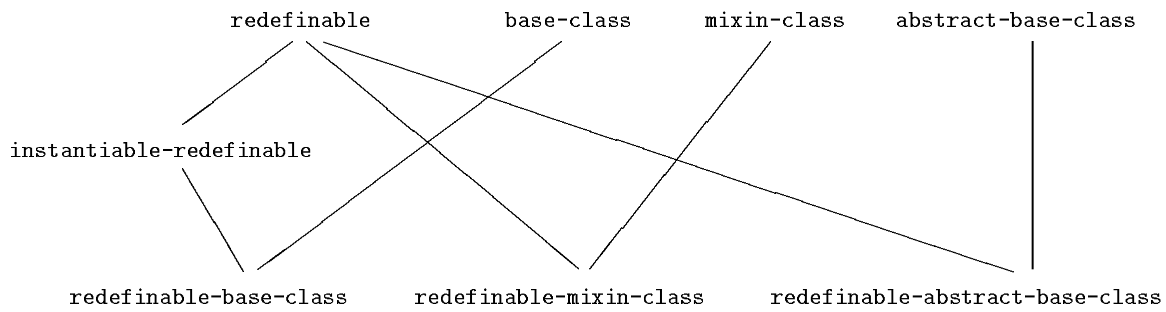

Figure 3: Specialization hierarchy of metaclasses for redefinition.

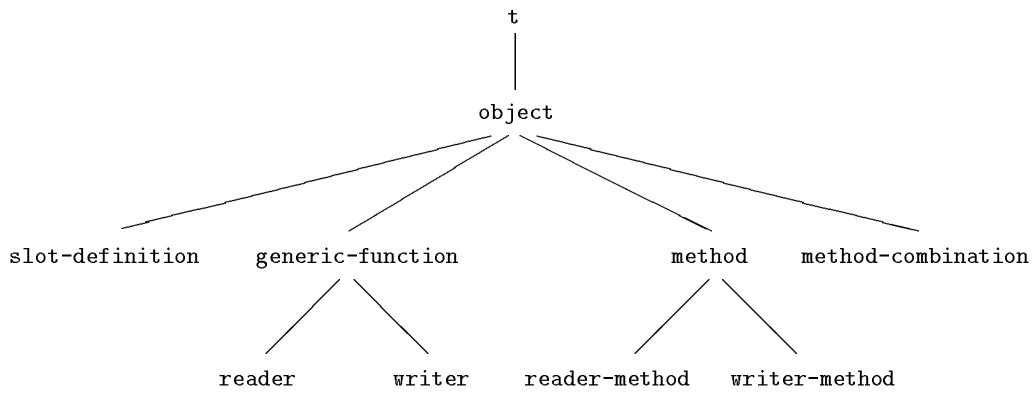

Figure 4: Specialization hierarchy of other classes.

4.3.2 Initialization Argument Names

The following table shows which system classes define initialization argument names.

Classes	Initargs
`class`	`name superclasses slots initargs`
`slot-definition`	`name initform initfunction slot-class type`
`generic-function`	`name lambda-list method-class method-combination discriminating-function`
`method`	`name qualifiers lambda-list specializers function`

4.3.3 Predicates

(**abstract-p** *object*) → *boolean* **Function**

The function `abstract-p`[5] returns `t` if its argument is an abstract class or `nil` otherwise.

(**class-p** *object*) → *boolean* **Function**

The function `class-p`[5] returns `t` if its argument is a class or `nil` otherwise.

(**mixin-p** *object*) → *boolean* **Function**

The function `mixin-p`[5] returns `t` if its argument is a mixin class or `nil` otherwise.

(**metaclass-p** *object*) → *boolean* **Function**

The function `metaclass-p`[5] returns `t` if its argument is a metaclass or `nil` otherwise.

4.3.4 Operations on Class Names

(**find-class** *symbol* `&rest` *errorp*) → {*class* | `nil`} **Function**

The function `find-class`[13] returns the class object named by the given symbol. If there is no such class and *errorp* is `nil`, `find-class` returns `nil`. If there is no such class and *errorp* is non-`nil`, an error is signaled.
The class associated with a particular symbol can be changed by using `setf` with `find-class`.

(**ensure-class** *class-name metaclass-name initargs*) → *class* **Function**

The function `ensure-class`[14] is used to define a new or to redefine an existing class. It takes the class name, the metaclass name and a list of alternating initialization argument names and values for the class and returns a class object. It is used within `defclass`, but it does not create readers, writers nor a type to be used by the function `typep`.

(**obj-subtypep** *type1 type2*) → *boolean* **Function**

The function `obj-subtypep`[9] takes two type specifiers and returns `t` if *type1* is a subtype of *type2*, otherwise `nil` is returned. Both arguments have to be type specifiers.
See also: `obj-typep, obj-type-of`.

4.3.5 Access Functions

(**class-direct-methods** *class*) → *list* **Generic Function**

`class-direct-methods` (*class* `redefinable`) **Method**

The generic function `class-direct-methods`[10] takes a class object and returns a list of its direct methods.

(**class-direct-subclasses** *class*) → *list* **Generic Function**

`class-direct-subclasses` (*class* `redefinable`) **Method**

The generic function `class-direct-subclasses` takes a class object and returns a list of its direct subclasses.

[13] In CLOS `find-class` has two optional parameters, *errorp* and *environment*.
[14] In the draft MOP of CLOS `ensure-class` has other arguments.

(**class-direct-superclasses** *class*) → *list* **Generic Function**

`class-direct-superclasses` (*class* `redefinable`) **Method**

The generic function `class-direct-superclasses` takes a class object and returns a list of its direct superclasses.

(**class-name** *class*) → *symbol* **Generic Function**

`class-name` (*class* `class`) **Method**

The generic function `class-name`[15] takes a class object and returns its name.
See also: `find-class`

(**class-precedence-list** *class*) → *list* **Generic Function**

`class-precedence-list` (*class* `class`) **Method**

The generic function `class-precedence-list` takes a class object and returns its class precedence list which is a list of classes.
See also: `compute-class-precedence-list`.

(**class-direct-slots** *class*) → *list* **Generic Function**

`class-direct-slots` (*class* `redefinable`) **Method**

The generic function `class-direct-slots` takes a class object and returns a list of its direct slot definitions.

(**class-slots** *class*) → *list* **Generic Function**

`class-slots` (*class* `class`) **Method**

The generic function `class-slots` takes a class object and returns a list of its effective slot definitions.

(**class-direct-initargs** *class*) → *list* **Generic Function**

`class-direct-initargs` (*class* `redefinable`) **Method**

The generic function `class-direct-initargs`[10] takes a class object and returns a list of its direct initargs.

(**class-initargs** *class*) → *list* **Generic Function**

`class-initargs` (*class* `class`) **Method**

The generic function `class-initargs`[10] takes a class object and returns a list of all its initargs.

(**find-slot** *class slot-name* `&rest` *errorp*) → {*slot-definition* | `nil`} **Generic Function**

`find-slot` (*class* `class`) *slot-name* `&rest` *errorp* **Method**

The generic function `find-slot`[10] takes a class object and a slot name and returns the slot definition object of *slot-name*. If there is no such slot definition and *errorp* is `nil`, `find-slot` returns `nil`. If there is no such slot definition and *errorp* is non-`nil`, an error is signaled.
See also: `slot-exists-p`.

(**class-prototype** *class*) → *object* **Generic Function**

`class-prototype` (*class* `instantiable`) **Method**

The generic function `class-prototype` takes a class and returns a prototype instance of the class.

4.3.6 Creating Instances

(**make-instance** *class* `&rest` *initargs*) → *object* **Function**

The function `make-instance` creates and returns a new instance of the given class.
The *class* argument is a class object or a symbol that names a class. The remaining arguments form a possible empty sequence of alternating initialization argument names and values.
`make-instance` checks the initialization arguments.
See also: `initialize-instance`.

(**allocate-instance** *class initargs*) → *object* **Generic Function**

`allocate-instance` (*object* `instantiable`) *initargs* **Method**

`allocate-instance` (*object* `instantiable-redefinable`) *initargs* **Method**

[15] In CLOS there exists also a generic function for setting the name of a class.

The generic function `allocate-instance`[16] is called by `make-instance` to create a new, uninitialized instance of a class. The *class* argument is a class object and the remaining arguments form a possible empty sequence of alternating initialization argument names and values.

(**initialize-instance** *class* `&rest` *initargs*) → *object* **Generic Function**

`initialize-instance` (*class* `defined`) `&rest` *initargs* **Method**

The generic function `initialize-instance` is called by `make-instance` to initialize a newly created instance, which is returned as value.

The *class* argument is the object to be initialized. The *initargs* argument is a possible empty sequence of alternating initialization argument names and values.

First, the primary method for `defined` calls the next `initialize-instance` method. Second, it computes the direct slot definitions; finally, it calls `finalize-inheritance`.

See also: `make-instance, slot-boundp, slot-makunbound`.

(**reinitialize-instance** *class* `&rest` *initargs*) → *object* **Generic Function**

`reinitialize-instance` (*class* `defined`) `&rest` *initargs* **Method**

The generic function `reinitialize-instance` can be used to change the values of local slots according to initialization arguments. This generic function is called by the Meta-Object Protocol. It can also be called by users.

The *class* argument is the object to be initialized and the *initargs* argument is a possible empty sequence of alternating initialization argument names and values. As its result, it returns the modified object.

The primary method for `defined` does the same work as `initialize-instance` for `defined` does, except that the next `reinitialize-instance` method is called instead.

See also: `initialize-instance, slot-boundp, slot-makunbound`.

4.3.7 Inheritance

(**finalize-inheritance** *class direct-superclasses plist*) → *class* **Generic Function**

`finalize-inheritance` (*class* `class`) *direct-superclasses plist* **Method**

`finalize-inheritance` (*class* `defined`) *direct-superclasses plist* **Method**

`finalize-inheritance` (*class* `instantiable`) *direct-superclasses plist* **Method**

`finalize-inheritance` (*class* `redefinable`) *direct-superclasses plist* **Method**

The generic function `finalize-inheritance`[17] is called to perform the inheritance computation for `class`, i.e. it calls `validate-superclass`, `compute-class-precedence-list`, `compute-slots`, `compute-initargs`, etc.

(**compute-class-precedence-list** *class direct-superclasses*) → *list* **Generic Function**

`compute-class-precedence-list` (*class* `class`) *direct-superclasses* **Method**

The generic function `compute-class-precedence-list`[18] is called to compute the class precedence list of *class*. It takes the class *class* and a list of direct superclasses and returns a list of classes.

(**compute-effective-slot-definition** *class name slot-definitions*) → *slot-definition* **Generic Function**

`compute-effective-slot-definition` (*class* `defined`) *name slot-definitions* **Method**

The generic function *compute-effective-slot-definition* is called to combine the direct slot definition object of *class* and its superclasses into a single effective slot definition object which is returned as value. The *name* argument is the name of the slot. The *slot-definitions* argument is a list of direct slot definition objects.

See also: `compute-slots, effective-slot-definition-class`.

(**compute-slots** *class direct-superclasses direct-slot-definitions*) → *list* **Generic Function**

`compute-slots` (*class* `defined`) *direct-superclasses direct-slot-definitions* **Method**

The generic function `compute-slots`[19] is called to compute the set of slots accessible in instance of *class*.

[16] In the draft MOP of CLOS the *initargs* argument of `allocate-instance` is a `&rest` argument.

[17] In the draft MOP of CLOS the *direct-superclasses* and *plist* argument of `finalize-inheritance` do not exist.

[18] In the draft MOP of CLOS the *direct-superclasses* argument of `compute-class-precedence-list` does not exist.

[19] In the draft MOP of CLOS the *direct-superclasses* and *direct-slot-definitions* argument of `compute-slots` do not exist.

It takes a *class*, its *direct-superclasses* and its *direct-slot-definitions* and returns the list of effective slot definitions for *class*.
See also: `compute-effective-slot-definition, effective-slot-definition-class`.

(**compute-initargs** *class direct-superclasses direct-initargs*) → *list* **Generic Function**
`compute-initargs` (*class* `defined`) *direct-superclasses direct-initargs* **Method**
The generic function `compute-initargs`[5] is called to compute the set of initialization arguments for instances of *class*. It takes a *class*, its *direct-superclasses* and its *direct-initargs* and returns the list of initialization arguments for *class*.

(**direct-slot-definition-class** *class canonical-slot-specification*) → *class* **Generic Function**
`direct-slot-definition-class` (*class* `defined`) *canonical-slot-specification* **Method**
This generic function is called to determine the class of each direct slot definition object of *class*. When a class is initialized, each *canonical-slot-specification* must be converted to a direct slot definition object. The direct slot definition class is returned.

(**effective-slot-definition-class** *class slot-definitions*) → *class* **Generic Function**
`effective-slot-definition-class` (*class* `defined`) *slot-definitions* **Method**
This generic function takes a *class* and a list of slot definitions (*slot-definitions*) and returns the effective slot definition class.

(**validate-superclass** *class superclass lastp*) → *boolean* **Generic Function**
`validate-superclass` (*class* `abstract`) (*superclass* `instantiable`) *lastp* **Method**
`validate-superclass` (*class* `multiple-inherited`) (*superclass* `single-inherited`) *lastp* **Method**
`validate-superclass` (*class* `multiple-inherited`) (*superclass* `multiple-inherited`) *lastp* **Method**
`validate-superclass` (*class* `single-inherited`) (*superclass* `multiple-inherited`) *lastp* **Method**
`validate-superclass` (*class* `single-inherited`) (*superclass* `single-inherited`) *lastp* **Method**
`validate-superclass` (*class* `redefinable`) (*superclass* `redefinable`) *lastp* **Method**
The generic function `validate-superclass`[20] is called to determine whether *superclass* is suitable for use as a superclass of *class*.
The *lastp* argument indicates that *superclass* is the last superclass given in the list of direct superclasses of *class*.

4.3.8 Creating Accessors

(**make-reader** *class slot reader-name*) → *function* **Generic Function**
`make-reader` (*class* `multiple-inherited`) (*slot* `slot-definition`) *reader-name* **Method**
`make-reader` (*class* `single-inherited`) (*slot* `slot-definition`) *reader-name* **Method**
The generic function `make-reader`[10] creates a slot reader called *reader-name* for *slot*. It may be applied on instances of *class*. `make-reader` is called within `defreader` and returns the function for reading the slot.
See also: `defreader`.

(**make-writer** *class slot writer-name*) → *function* **Generic Function**
`make-writer` (*class* `multiple-inherited`) (*slot* `slot-definition`) *writer-name* **Method**
`make-writer` (*class* `single-inherited`) (*slot* `slot-definition`) *writer-name* **Method**
The generic function `make-writer`[10] creates a slot writer called *writer-name* for *slot*. It may be applied on instances of *class*. `make-writer` is called within `defwriter` and returns the function for reading the slot.
See also: `defwriter`.

4.3.9 Exception Protocol

(**slot-missing** *class object slot-name operation* `&rest` *new-value*) → *undefined* **Generic Function**
`slot-missing` (*class* `defined`) (*object* `object`) *slot-name operation* `&rest` *new-value* **Method**

[20] In the draft MOP of CLOS the *lastp* argument of `validate-superclass` does not exist.

The generic function `slot-missing`[21] is invoked when an attempt is made to access a slot of an object whose class is `defined`. The default method signals an error.
The generic function `slot-missing` is not intended to be called by programmers. Programmers may write methods for it.
The required arguments of `slot-missing` are the class of the object that is being accessed, the object, the slot name, and a symbol that indicates the operation that caused `slot-missing` to be invoked.
If a method written for `slot-missing` returns a value, this value is returned as the value of the original function call.
The generic function `slot-missing` may be called during evaluation of `slot-value`, `(setf slot-value)`, `slot-boundp`, and `slot-makunbound`. For each of these operations the corresponding symbol for the *operation* argument is `slot-value`, `setf`, `slot-boundp`, and `slot-makunbound` respectively.
See also: `slot-value, slot-boundp, slot-makunbound`.

(**slot-unbound** *class object slot-name*) → *object* **Generic Function**
`slot-unbound` (*class* `defined`) (*object* `object`) *slot-name* **Method**
The generic function `slot-unbound` is called when an attempt is made to read an unbound slot of an instance whose class is `defined`. The default method signals an error.
The generic function `slot-unbound` is not intended to be called by programmers. Programmers may write methods for it.
The arguments of `slot-unbound` are the class of the instance whose slot was accessed, the instance itself, and the name of the slot.
If a method written for `slot-unbound` returns a value, this value is returned as the value of the original function call.
An unbound slot may occur if no `:initform` form was specified for the slot and the slot value has not been set, or if `slot-makunbound` has been applied on the slot.
See also: `slot-makunbound`.

4.4 Slot Definitions

4.4.1 Access Functions

(**slot-definition-initform** *slot-definition*) → *form* **Generic Function**
`slot-definition-initform` (*slot-definition* `slot-definition`) **Method**
The generic function `slot-definition-initform` takes a slot definition object and returns its initform.

(**slot-definition-initfunction** *slot-definition*) → *function* **Generic Function**
`slot-definition-initfunction` (*slot-definition* `slot-definition`) **Method**
The generic function `slot-definition-initfunction` takes a slot definition object and returns its initfunction.

(**slot-definition-name** *slot-definition*) → *symbol* **Generic Function**
`slot-definition-name` (*slot-definition* `slot-definition`) **Method**
The generic function `slot-definition-name` takes a slot definition object and returns its name.

(**slot-definition-type** *slot-definition*) → *type* **Generic Function**
`slot-definition-type` (*slot-definition* `slot-definition`) **Method**
The generic function `slot-definition-type` takes a slot definition object and returns its type.

4.5 Generic Functions

4.5.1 Predicates

(**generic-p** *object*) → *boolean* **Function**
The function `generic-p`[5] returns `t` if its argument is a generic function or `nil` otherwise.
See also: `method-p`.

[21] In CLOS the *new-value* argument of `slot-missing` is an optional parameter.

4.5.2 Operations on Generic Function Names

(**find-gfn** *symbol errorp*) → {*generic-function* | `nil`} **Function**

The function `find-gfn`[5] [22] returns the generic function object named by the given symbol. If there is no such generic function and *errorp* is `nil`, `find-gfn` returns `nil`. If there is no such generic function and *errorp* is non-`nil`, an error is signaled.

The generic function associated with a particular symbol can be changed by using `setf` with `find-gfn`.

4.5.3 Access Functions

(**find-method** *gfn qualifiers specializers* `&rest` *errorp*) → {*method* | `nil`} **Generic Function**

`find-method` (*gfn* `generic-function`) *qualifiers specializers* `&rest` *errorp* **Method**

The generic function `find-method`[23] takes a generic function *gfn* and returns the method object that corresponds to the *qualifiers* and *specializers* arguments of `find-method`.

The *qualifiers* argument is a list of the method qualifiers for the method. The order of the method qualifiers is significant.

The *specializers* argument is a list of the parameter specializers for the method. It must correspond in length to the number of required arguments of the generic function, otherwise an error is signaled.

(**generic-function-lambda-list** *generic-function*) → *lambda-list* **Generic Function**

`generic-function-lambda-list` (*generic-function* `generic-function`) **Method**

The generic function `generic-function-lambda-list` takes a generic function object and returns its lambda list.

(**generic-function-method-class** *generic-function*) → *class* **Generic Function**

`generic-function-method-class` (*generic-function* `generic-function`) **Method**

The generic function `generic-function-method-class` takes a generic function object and returns its method class.

(**generic-function-methods** *generic-function*) → *list* **Generic Function**

`generic-function-methods` (*generic-function* `generic-function`) **Method**

The generic function `generic-function-methods` takes a generic function object and returns a list of its methods.

(**generic-function-name** *generic-function*) → *symbol* **Generic Function**

`generic-function-name` (*generic-function* `generic-function`) **Method**

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

4.5.4 Initialization

(**initialize-instance** *generic-function* `&rest` *initargs*) → *object* **Generic Function**

`initialize-instance` (*generic-function* `generic-function`) `&rest` *initargs* **Method**

The generic function `initialize-instance` is called by `make-instance` to initialize a newly created instance, which is the returned as value.

The *generic-function* argument is the object to be initialized. The *initargs* argument is a possible empty sequence of alternating initialization argument names and values.

First, the primary method for `generic-function` calls the next `initialize-instance` method. Afterwards, it initializes the signature and the discriminating function of the generic function.

See also: `make-instance, slot-boundp, slot-makunbound`.

4.5.5 Updating Methods

(**add-method** *generic-function method*) → *generic-function* **Function**

[22] In CLOS with `(symbol-function generic-function-name)` the generic function object is obtained but in MCS only the funcallable function.

[23] In CLOS the *errorp* argument of `find-method` is an optional parameter.

The function `add-method`[24] adds a method object to a generic function object. It destructively modifies the generic function and returns the modified generic function as its result.
The *generic-function* argument is a generic function object. The *method* argument is a method object[25]. If the given method corresponds to an existing method of the generic function on parameter specializers and qualifiers, the existing method is replaced.
See also: `defmethod, defgeneric, find-method, remove-method`.

(**remove-method** *generic-function method*) → *generic-function* **Generic Function**
`remove-method` (*generic-function* `generic-function`) **Method**
(*method* `method`)
The generic function `remove-method` removes *method* from *generic-function*. It destructively modifies the specified generic function and returns the modified generic function object as its result. The result of `remove-method` is `eq` to the *generic-function* argument.
See also: `find-method, undefmethod`.

4.5.6 Applicable Methods

(**compute-applicable-methods** *generic-function function-arguments*) → *list* **Function**
Given a generic function *generic-function* and a set of arguments *function-arguments*, the function `compute-applicable-methods` returns the set of methods that are applicable for those arguments in order of precedence.

4.5.7 Exception Protocol

(**no-applicable-method** *generic-function* `&rest` *function-arguments*) → *undefined* **Generic Function**
`no-applicable-method` (*generic-function* `generic-function`) `&rest` *arguments* **Method**
The generic function `no-applicable-method` is called when a generic function is invoked and no method on that generic function is applicable. The default method signals an error. The generic function `no-applicable-method` is not intended to be called by programmers. Programmers may write methods for it.
The *generic-function* argument of `no-applicable-method` is the generic function object on which no applicable method was found. The *arguments* argument is a sequence of the arguments to that generic function.

(**no-next-method** *generic-function method* `&rest` *function-arguments*) → *undefined* **Generic Function**
`no-next-method` (*generic-function* `generic-function`)(*method* `method`) `&rest` *arguments* **Method**
The generic function `no-next-method` is called by `call-next-method` when there is no next method. The system-supplied method on `no-next-method` signals an error.
The *generic-function* argument is the generic function object to which the method that is the second argument belongs. The *method* argument is the innermost method that contained the call to `call-next-method` for which there is no next method. The *arguments* argument is the list of the arguments to the call of *generic-function*.
See also: `call-next-method`.

4.6 Methods

4.6.1 Predicates

(**method-p** *object*) → *boolean* **Function**
The function `method-p`[5] returns `t` if its argument is a method or `nil` otherwise.
See also: `generic-p`.

[24] In CLOS `add-method` is a standard generic function.
[25] In CLOS the congruence of the lambda-lists is checked.

4.6.2 Access Functions

(**method-function** *method*) → *function* **Generic Function**

`method-function` (*method* `method`) **Method**

The generic function `method-function` takes a method object and returns its function.

(**method-generic-function** *method*) → *generic-function* **Generic Function**

`method-generic-function` (*method* `method`) **Method**

The generic function `method-generic-function` takes a method object and returns its generic function.

(**method-lambda-list** *method*) → *lambda-list* **Generic Function**

`method-lambda-list` (*method* `method`) **Method**

The generic function `method-lambda-list` takes a method object and returns its lambda list.

(**method-name** *method*) → *symbol* **Generic Function**

`method-name` (*method* `method`) **Method**

The generic function `method-name` takes a method object and returns its name.

(**method-qualifiers** *method*) → *list* **Generic Function**

`method-qualifiers` (*method* `method`) **Method**

The generic function `method-qualifiers` takes a method object and returns a list of its qualifiers.

(**method-specializers** *method*) → *list* **Generic Function**

`method-specializers` (*method* `method`) **Method**

The generic function `method-specializers` takes a method object and returns a list of its specializers (i.e. classes). Specializers are often called signature.

4.7 Mapping the Macro to the Functional Interface

This section shows how the macro calls of `defclass`, `defgeneric`, `defmethod`, `defreader` and `defwriter` will expand.

```
(defclass c1 () ((x :initform 42
                    :type integer
                    :initarg x
                    :accessor x
                    :user-defined-option (my-expression))))

-> (progn
     (ensure-class 'c1
                   'standard-base-class
                   (list 'name 'c1
                         'superclasses (list (find-class 'standard-object))
                         'slots (list (list 'name 'x
                                            'initform '42
                                            'initfunction #'(lambda () 42)
                                            'type '(and integer)
                                            ':user-defined-option (my-expression)))
                         'initargs '(x)))
     ...
     (deftype c1 ... )
     ...
     (defreader x c1 x)
     (defwriter (setf x) c1 x)
     ... )



(defgeneric foo (arg1 arg2)
  (:class my-generic-function)
```

```
  (:method-class my-method))

-> (define-function foo
     ...
     (ensure-gfn 'foo
                 '(arg1 arg2)
                 (find-method-combination 'standard)
                 'class (find-class 'my-generic-function 'signal-error)
                 'method-class (find-class 'my-method 'signal-error))
     ... )



(defmethod foo ((arg1 c1) arg2)
  42)

-> (define-function foo
     ...
     (let ((%gfn (ensure-gfn 'foo
                             '(arg1 arg2)
                             (find-method-combination '()))))
       (let ((method (make-instance (or (%%gfn-method-class %gfn)
                                        (find-class 'method))
                                    'name 'foo
                                    'lambda-list '(arg1 arg2)
                                    'specializers (mapcar
                                                   #'(lambda (cl-name)
                                                       (find-class cl-name 'signal-error))
                                                   '(c1 t))
                                    'qualifiers '()
                                    'function #'(lambda ... 42 ... ))))
         (add-method %gfn method)))
     ... )



(defreader c1-x c1 x)

-> (define-function c1-x
     (let ((class (find-class 'c1)))
       (make-reader class (%find-slot class 'x) 'c1-x)))



(defwriter (setf c1-x) c1 x)

-> (progn
     (defsetf c1-x (object) (value)
       ... )
     (define-function |(SETF C1-X)|
       (let ((class (find-class 'c1)))
         (make-writer class (%find-slot class 'x) '|(SETF C1-X)|))))



(defwriter set-x c1 x)

-> (progn
     (define-function set-x
       (let ((class (find-class 'c1)))
         (make-writer class (%find-slot class 'x) 'set-x))))
```

5 Programming Environment

This section describes some functions which should only be used for development purposes and not within a MCS program.

5.1 Expressions

5.1.1 Definitions

(**undefgeneric** *function-specifier lambda-list gfn-option**) → *generic-function* **Macro**

function-specifier::= {*symbol* | `(setf` *symbol*`)`}
lambda-list::= ({*var*}+ [`&rest` *var*])

The macro `undefgeneric`[5] removes a generic function and all of its methods. The *gfn-option* arguments are ignored. For details about the arguments see `defgeneric`.

(**undefmethod** *function-specifier* [*qualifier*] *specialized-lambda-list* [*doc*] *form**) → *method* **Macro**

function-specifier ::= {*symbol*} | `(setf` *symbol*`)`
qualifier ::= `:before` | `:after` | `:around`
specialized-lambda-list::= ({*var* | (*var parameter-specializer-name*)}+ [`&rest` *var*])
parameter-specializer-name::= *symbol*

The macro `undefmethod`[5] removes the specified method from its generic function. The *doc* and *form* arguments are ignored. For details about the arguments see `defmethod`.

5.2 Operations

5.2.1 Objects

(**obj-documentation** *object* `&rest` *doc-options*) → *string* **Generic Function**

`obj-documentation` (*object* `cons`) `&rest` *doc-options* **Method**
`obj-documentation` (*class* `base-class`) `&rest` *doc-options* **Method**
`obj-documentation` (*method* `method`) `&rest` *doc-options* **Method**
`obj-documentation` (*gfn* `generic-function`) `&rest` *doc-options* **Method**
`obj-documentation` (*symbol* `symbol`) `&rest` *doc-options* **Method**

(**|(setf obj-documentation)|** *new-value object* `&rest` *doc-options*) → *string* **Generic Function**

`(setf obj-documentation)` *new-value* (*object* `cons`) `&rest` *doc-options* **Method**
`(setf obj-documentation)` *new-value* (*symbol* `symbol`) `&rest` *doc-options* **Method**

The generic function `obj-documentation`[9] returns the documentation string associated with the given MCS object if it is available; otherwise it returns `nil`.
The generic function `|(setf obj-documentation)|` is used to update a documentation.

(**apropos-protocol** *object substring*) → *list* **Generic Function**

`apropos-protocol` (*object* `object`) *substring* **Method**

The generic function `apropos-protocol`[5] returns a list of generic function names which include the user-supplied substring *substring* and which are applicable to *object*.

(**object-protocol** *object*) → *list* **Generic Function**

`object-protocol` (*object* `object`) **Method**

The generic function `object-protocol`[5] returns a list of generic function names which are applicable to *object*. *Object* can be any defined object.

5.2.2 Classes

(**class-direct-protocol** *class*) → *list* **Generic Function**

`class-direct-protocol` (*class* `class`) **Method**

The generic function `class-direct-protocol`[5] returns a list of generic function names which have *class* as a specializer.

(**class-protocol** *class*) → *list* **Generic Function**

`class-protocol` (*class* `class`) **Method**

The generic function `class-protocol`[5] returns a list of generic function names which are applicable to instances of *class*.

5.3 Redefinition

5.3.1 Redefining Classes

MCS supports the redefinition of classes and lazy updating of their subclasses and all affected instances by special metaclasses: `redefinable-base-class`, `redefinable-mixin-class`, and `redefinable-abstract-base-class`.

They can be used explicitly in the functional interface as well as in the macro constructors `defclass`, `defabstract`, and `defmixin` via the `:metaclass` class option. They can also be used implicitly by these macro constructors. The default metaclass referred to is controlled by the function `redefine-mode`, which associates the special class names `standard-base-class`, `standard-mixin-class`, and `standard-abstract-base-class` with corresponding classes. These special class names are used as default metaclass in `defclass`, `defmixin`, and `defabstract`, respectively.

There may exist redefinable and non-redefinable classes during development or run-time of a program. Subclasses of a redefinable class should always be redefinable, otherwise they could not be updated to the new definition of its superclass.

A redefinable class can change its class (this is a metaclass), but it can not be changed to a non-redefinable class or vice versa.

(**redefine-mode** `&rest` *switch*) → *boolean* **Function**

The function is used for setting the redefine mode on or off, or for getting its actual value if *switch* is not supplied. The *switch* argument has to be `t` or `nil` for setting the mode on or off, respectively.

According to *switch*, the function associates the names `standard-xxx-class` to `redefinable-xxx-class` or `xxx-class`, respectively, where `xxx` can be one of `abstract`, `base` or `mixin`.

5.3.2 Updating Subclasses

A class is called non-finalized if one of its superclasses has been redefined. A non-finalized class will be updated when applying `finalize-inheritance` on it. This is done implicitly, when creating an instance of that class or one of its subclasses as well as a new subclass of it. A non-finalized class will not be updated when accessing its slots or applying an operation on it.

5.3.3 Updating Instances

An instance whose class has been redefined is called an obsolete instance. It will be updated as soon as one of its slots is accessed or a generic function is applied on it. This is done in the following way: A bound slot keeps its old value if there is a slot of the same name in the old and new class definition[26]. If there is not a slot of the same name in the new class definition the slot is discarded. All other slots are initialized according to their default initial value forms (`:initform`) of the new class definition.

All instances remain garbage collectable.

See also: `change-class`.

5.3.4 Generic Functions

See `defgeneric` and `undefgeneric`.

5.3.5 Methods

See `defmethod` and `undefmethod`.

[26] In CLOS an unbound slot remains unbound, even if there is a default initial value form for this slot in the new class definition.

Appendix

MCS extensions to CLOS:

`defabstract`
`defconstructor`
`defmixin`

Limitations in MCS:

`:initarg`
`add-method` is not a generic function
`call-next-method` is not a function
`next-method-p` is not a function
no general multiple inheritance
no foreward-referenced classes
lambda lists of generic functions and methods
only standard method combination

Renamed functions in MCS:

`obj-describe`
`obj-subtypep`
`obj-typep`
`obj-type-of`

Different lambda lists in CLOS:

`find-class`
`find-method`
`slot-missing`

Not available in MCS but in CLOS:

`define-method-combination`
eql specializers for methods
`generic-flet`
`generic-labels`
local generic functions
multiple values as return value of methods
rebinding of a methods lambda-variables
`shared-initialize`
shared slots
`structure-class`
`with-added-methods`
`with-slots`
`with-accessors`
`:default-initargs`
`:allocation`
`&allow-other-keys`
`&aux`
`&key`
`&optional`

MCS extensions to the MOP of CLOS:

`abstract-p`
`class-direct-initargs`
`class-direct-methods`
`class-initargs`
`class-p`
`defreader`
`defwriter`
`find-gfn`
`find-slot`
`generic-p`
`make-reader`
`make-writer`
`metaclass-p`
`method-p`
`mixin-p`
`obj-copy`

Different lambda lists in MOP of CLOS:

`allocate-instance`
`compute-class-precedence-list`
`compute-slots`
`ensure-class`
`finalize-inheritance`
`validate-superclass`

MCS extensions for the programming environment :

`apropos-protocol`
`class-direct-protocol`
`class-protocol`
`object-protocol`
`undefgeneric`
`undefmethod`

References

[BCK89] H. Bretthauer, Th. Christaller, and J. Kopp. Multiple vs. Single Inheritance in Object-oriented Programming Languages. What do we really want? Arbeitspapiere der GMD 415, GMD, Sankt Augustin, November 1989.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. *Sigplan Notices*, 23(Special Issue), September 1988.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops, Merging Lisp and Object-Oriented Programming. In *OOPSLA 86 Conference Proceedings*, volume 21 of *SIGPLAN Notices (Special Issue)*, pages 17–29. ACM, November 1986.

[Bra83] Ronald J. Brachman. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer*, 16(10):30–36, October 1983.

[Chr88] Th. Christaller. Eine Einführung in LISP. In Th. Christaller, H.-W. Hein, and M. M. Richter, editors, *Künstliche Intelligenz. Theoretische Grundlagen und Anwendungsfelder*, volume 159 of *Informatik-Fachberichte*, pages 1–35. Springer, Berlin, 1988.

[Coi88] Pierre Cointe. The ObjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 155–176. North Holland, Amsterdam, 1988.

[dP88] Franco di Primio. Micro Common Flavors. Arbeitspapiere der GMD 295, GMD, Sankt Augustin, February 1988.

[Hay89] Christopher Haynes. Draft Standard for the Scheme Programming Language. August 1989.

[ILO91] ILOG S.A., Gentilly. *Le-Lisp version 16, Alpha Release 1.6 Manual*, February 1991.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp. A Programmer's Guide to CLOS*. Addison-Wesley, Reading, Massachusetts, 1989.

[PN91] Julian Padget and Greg Nuyens, editors. *The EuLisp Definition, Version 0.7*. March 1991. unpublished.

[SJ90] Guy L. Steele Jr. *Common Lisp - The Language, Second Edition*. Digital Press, Bedford, Massachusetts, 1990.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, pages 10–20, May 1988.

[WM81] Daniel Weinreb and David Moon. *LISP Machine Manual*. Symbolics Inc., Cambridge, Massachusetts, 4. edition, July 1981.

Index