Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**define-method-combination**

to bind additional variables. It is also possible to bypass the method group specifier mechanism and do everything in the body *forms*. This is accomplished by writing a single method group with **\*** as its only *qualifier-pattern*; the variable is then bound to a *list* of all of the *applicable methods*, in most-specific-first order.

The body *forms* compute and return the *form* that specifies how the methods are combined, that is, the e↵ective method. The e↵ective method is evaluated in the *null lexical environment* augmented with a local macro definition for **call-method** and with bindings named by symbols not *accessible* from the COMMON-LISP-USER *package*. Given a method object in one of the *lists* produced by the method group specifiers and a *list* of next methods, **call-method** will invoke the method such that **call-next-method** has available the next methods.

When an e↵ective method has no e↵ect other than to call a single method, some implementations employ an optimization that uses the single method directly as the e↵ective method, thus avoiding the need to create a new e↵ective method. This optimization is active when the e↵ective method form consists entirely of an invocation of the **call-method** macro whose first *subform* is a method object and whose second *subform* is **nil** or unsupplied. Each **define-method-combination** body is responsible for stripping o↵ redundant invocations of **progn**, **and**, **multiple-value-prog1**, and the like, if this optimization is desired.

The list (:arguments . *lambda-list*) can appear before any declarations or *documentation string*. This form is useful when the method combination type performs some specific behavior as part of the combined method and that behavior needs access to the arguments to the *generic function*. Each parameter variable defined by *lambda-list* is bound to a *form* that can be inserted into the e↵ective method. When this *form* is evaluated during execution of the e↵ective method, its value is the corresponding argument to the *generic function*; the consequences of using such a *form* as the *place* in a **setf** *form* are undefined. Argument correspondence is computed by dividing the :arguments *lambda-list* and the *generic function lambda-list* into three sections: the *required parameters*, the *optional parameters*, and the *keyword* and *rest parameters*. The *arguments* supplied to the *generic function* for a particular *call* are also divided into three sections; the required *arguments* section contains as many *arguments* as the *generic function* has *required parameters*, the optional *arguments* section contains as many arguments as the *generic function* has *optional parameters*, and the keyword/rest *arguments* section contains the remaining arguments. Each *parameter* in the required and optional sections of the :arguments *lambda-list* accesses the argument at the same position in the corresponding section of the *arguments*. If the section of the :arguments *lambda-list* is shorter, extra *arguments* are ignored. If the section of the :arguments *lambda-list* is longer, excess *required parameters* are bound to forms that evaluate to **nil** and excess *optional parameters* are *bound* to their initforms. The *keyword parameters* and *rest parameters* in the :arguments *lambda-list* access the keyword/rest section of the *arguments*. If the :arguments *lambda-list* contains **&key**, it behaves as if it also contained **&allow-other-keys**.

In addition, **&whole** *var* can be placed first in the :arguments *lambda-list*. It causes *var* to Objects **7–79**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**define-method-combination**

be *bound* to a *form* that *evaluates* to a *list* of all of the *arguments* supplied to the *generic function*. This is di↵erent from **&rest** because it accesses all of the arguments, not just the keyword/rest *arguments*.

Erroneous conditions detected by the body should be reported with

**method-combination-error** or **invalid-method-error**; these *functions* add any nec essary contextual information to the error message and will signal the appropriate error.

The body *forms* are evaluated inside of the *bindings* created by the *lambda list* and method group specifiers. Declarations at the head of the body are positioned directly inside of *bindings* created by the *lambda list* and outside of the *bindings* of the method group variables. Thus method group variables cannot be declared in this way. **locally** may be used around the body, however.

Within the body *forms*, *generic-function-symbol* is bound to the *generic function object*.

*Documentation* is attached as a *documentation string* to *name* (as kind

**method-combination**) and to the *method combination object*.

Note that two methods with identical specializers, but with di↵erent *qualifiers*, are not ordered by the algorithm described in Step 2 of the method selection and combination process described in Section 7.6.6 (Method Selection and Combination). Normally the two methods play di↵erent roles in the e↵ective method because they have di↵erent *qualifiers*, and no matter how they are ordered in the result of Step 2, the e↵ective method is the same. If the two methods play the same role and their order matters, an error is signaled. This happens as part of the *qualifier* pattern matching in **define-method-combination**.

If a **define-method-combination** *form* appears as a *top level form*, the *compiler* must make the *method combination name* be recognized as a valid *method combination name* in subsequent **defgeneric** *forms*. However, the *method combination* is executed no earlier than when the **define-method-combination** *form* is executed, and possibly as late as the time that *generic functions* that use the *method combination* are executed.

**Examples:**

Most examples of the long form of **define-method-combination** also illustrate the use of the related *functions* that are provided as part of the declarative method combination facility.

;;; Examples of the short form of define-method-combination

(define-method-combination and :identity-with-one-argument t)

(defmethod func and ((x class1) y) ...)

;;; The equivalent of this example in the long form is:

(define-method-combination and

**7–80** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**define-method-combination**

(&optional (order :most-specific-first))

((around (:around))

(primary (and) :order order :required t))

(let ((form (if (rest primary)

‘(and ,@(mapcar #’(lambda (method)

‘(call-method ,method))

primary))

‘(call-method ,(first primary)))))

(if around

‘(call-method ,(first around)

(,@(rest around)

(make-method ,form)))

form)))

;;; Examples of the long form of define-method-combination

;The default method-combination technique

(define-method-combination standard ()

((around (:around))

(before (:before))

(primary () :required t)

(after (:after)))

(flet ((call-methods (methods)

(mapcar #’(lambda (method)

‘(call-method ,method))

methods)))

(let ((form (if (or before after (rest primary))

‘(multiple-value-prog1

(progn ,@(call-methods before)

(call-method ,(first primary)

,(rest primary)))

,@(call-methods (reverse after)))

‘(call-method ,(first primary)))))

(if around

‘(call-method ,(first around)

(,@(rest around)

(make-method ,form)))

form))))

;A simple way to try several methods until one returns non-nil

(define-method-combination or ()

((methods (or)))

‘(or ,@(mapcar #’(lambda (method)

‘(call-method ,method))

methods)))

Objects **7–81**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**define-method-combination**

;A more complete version of the preceding

(define-method-combination or

(&optional (order ’:most-specific-first))

((around (:around))

(primary (or)))

;; Process the order argument

(case order

(:most-specific-first)

(:most-specific-last (setq primary (reverse primary)))

(otherwise (method-combination-error "~S is an invalid order.~@

:most-specific-first and :most-specific-last are the possible values." order)))

;; Must have a primary method

(unless primary

(method-combination-error "A primary method is required."))

;; Construct the form that calls the primary methods

(let ((form (if (rest primary)

‘(or ,@(mapcar #’(lambda (method)

‘(call-method ,method))

primary))

‘(call-method ,(first primary)))))

;; Wrap the around methods around that form

(if around

‘(call-method ,(first around)

(,@(rest around)

(make-method ,form)))

form)))

;The same thing, using the :order and :required keyword options

(define-method-combination or

(&optional (order ’:most-specific-first))

((around (:around))

(primary (or) :order order :required t))

(let ((form (if (rest primary)

‘(or ,@(mapcar #’(lambda (method)

‘(call-method ,method))

primary))

‘(call-method ,(first primary)))))

(if around

‘(call-method ,(first around)

(,@(rest around)

(make-method ,form)))

form)))

**7–82** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**define-method-combination**

;This short-form call is behaviorally identical to the preceding

(define-method-combination or :identity-with-one-argument t)

;Order methods by positive integer qualifiers

;:around methods are disallowed to keep the example small

(define-method-combination example-method-combination ()

((methods positive-integer-qualifier-p))

‘(progn ,@(mapcar #’(lambda (method)

‘(call-method ,method))

(stable-sort methods #’<

:key #’(lambda (method)

(first (method-qualifiers method)))))))

(defun positive-integer-qualifier-p (method-qualifiers)

(and (= (length method-qualifiers) 1)

(typep (first method-qualifiers) ’(integer 0 \*))))

;;; Example of the use of :arguments

(define-method-combination progn-with-lock ()

((methods ()))

(:arguments object)

‘(unwind-protect

(progn (lock (object-lock ,object))

,@(mapcar #’(lambda (method)

‘(call-method ,method))

methods))

(unlock (object-lock ,object))))

**Side Eects:**

The *compiler* is not required to perform any compile-time side-e↵ects.

**Exceptional Situations:**

Method combination types defined with the short form require exactly one *qualifier* per method. An error of *type* **error** is signaled if there are applicable methods with no *qualifiers* or with *qualifiers* that are not supported by the method combination type. At least one primary method must be applicable or an error of *type* **error** is signaled.

If an applicable method does not fall into any method group, the system signals an error of *type* **error** indicating that the method is invalid for the kind of method combination in use.

If the value of the :required option is *true* and the method group is empty (that is, no applicable methods match the *qualifier* patterns or satisfy the predicate), an error of *type* **error** is signaled.

If the :order option evaluates to a value other than :most-specific-first or :most-specific-last, an error of *type* **error** is signaled.

Objects **7–83**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**See Also:**

**call-method**, **call-next-method**, **documentation**, **method-qualifiers**, **method-combination-error**, **invalid-method-error**, **defgeneric**, Section 7.6.6 (Method Selection and Combination), Sec tion 7.6.6.4 (Built-in Method Combination Types), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

**Notes:**

The :method-combination option of **defgeneric** is used to specify that a *generic function* should use a particular method combination type. The first argument to the :method-combination option is the *name* of a method combination type and the remaining arguments are options for that type.

**find-method** *Standard Generic Function*

**Syntax:**

**find-method** *generic-function method-qualifiers specializers* &optional *errorp*

*! method*

**Method Signatures:**

**find-method** (*generic-function* **standard-generic-function**)

*method-qualifiers specializers* &optional *errorp*

**Arguments and Values:**

*generic-function*—a *generic function*.

*method-qualifiers*—a *list*.

*specializers*—a *list*.

*errorp*—a *generalized boolean*. The default is *true*.

*method*—a *method object*, or **nil**.

**Description:**

The *generic function* **find-method** takes a *generic function* and returns the *method object* that agrees on *qualifiers* and *parameter specializers* with the *method-qualifiers* and *specializers* arguments of **find-method**. *Method-qualifiers* contains the method *qualifiers* for the *method*. The order of the method *qualifiers* is significant. For a definition of agreement in this context, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

The *specializers* argument contains the parameter specializers for the *method*. It must correspond in length to the number of required arguments of the *generic function*, or an error is signaled. This means that to obtain the default *method* on a given *generic-function*, a *list* whose elements are the *class* **t** must be given.

**7–84** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

If there is no such *method* and *errorp* is *true*, **find-method** signals an error. If there is no such *method* and *errorp* is *false*, **find-method** returns **nil**.

**Examples:**

(defmethod some-operation ((a integer) (b float)) (list a b))

*!* #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>

(find-method #’some-operation ’() (mapcar #’find-class ’(integer float)))

*!* #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>

(find-method #’some-operation ’() (mapcar #’find-class ’(integer integer)))

*.* Error: No matching method

(find-method #’some-operation ’() (mapcar #’find-class ’(integer integer)) nil) *!* NIL

**Aected By:**

**add-method**, **defclass**, **defgeneric**, **defmethod**

**Exceptional Situations:**

If the *specializers* argument does not correspond in length to the number of required arguments of the *generic-function*, an an error of *type* **error** is signaled.

If there is no such *method* and *errorp* is *true*, **find-method** signals an error of *type* **error**.

**See Also:**

Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers)

**add-method** *Standard Generic Function*

**Syntax:**

**add-method** *generic-function method ! generic-function*

**Method Signatures:**

**add-method** (*generic-function* **standard-generic-function**)

(*method* **method**)

**Arguments and Values:**

*generic-function*—a *generic function object*.

*method*—a *method object*.

**Description:**

The generic function **add-method** adds a *method* to a *generic function*.

Objects **7–85**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

If *method* agrees with an existing *method* of *generic-function* on *parameter specializers* and *qualifiers*, the existing *method* is replaced.

**Exceptional Situations:**

The *lambda list* of the method function of *method* must be congruent with the *lambda list* of *generic-function*, or an error of *type* **error** is signaled.

If *method* is a *method object* of another *generic function*, an error of *type* **error** is signaled.

**See Also:**

**defmethod**, **defgeneric**, **find-method**, **remove-method**, Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers)

**initialize-instance** *Standard Generic Function*

**Syntax:**

**initialize-instance** *instance* &rest *initargs* &key &allow-other-keys *! instance*

**Method Signatures:**

**initialize-instance** (*instance* **standard-object**) &rest *initargs*

**Arguments and Values:**

*instance*—an *object*.

*initargs*—a *defaulted initialization argument list*.

**Description:**

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

The system-supplied primary *method* on **initialize-instance** initializes the *slots* of the *instance* with values according to the *initargs* and the :initform forms of the *slots*. It does this by calling the generic function **shared-initialize** with the following arguments: the *instance*, **t** (this indicates that all *slots* for which no initialization arguments are provided should be initialized according to their :initform forms), and the *initargs*.

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**.

**7–86** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**See Also:**

**shared-initialize**, **make-instance**, **slot-boundp**, **slot-makunbound**, Section 7.1 (Object Creation and Initialization), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

**class-name** *Standard Generic Function*

**Syntax:**

**class-name** *class ! name*

**Method Signatures:**

**class-name** (*class* **class**)

**Arguments and Values:**

*class*—a *class object*.

*name*—a *symbol*.

**Description:**

Returns the *name* of the given *class*.

**See Also:**

**find-class**, Section 4.3 (Classes)

**Notes:**

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 Section 4.3 (Classes).

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

Objects **7–87**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**(setf class-name)** *Standard Generic Function*

**Syntax:**

**(setf class-name)** *new-value class ! new-value*

**Method Signatures:**

**(setf class-name)** *new-value* (*class* **class**)

**Arguments and Values:**

*new-value*—a *symbol*.

*class*—a *class*.

**Description:**

The generic function (setf class-name) sets the name of a *class* object.

**See Also:**

**find-class**, *proper name*, Section 4.3 (Classes)

**class-of** *Function*

**Syntax:**

**class-of** *object ! class*

**Arguments and Values:**

*object*—an *object*.

*class*—a *class object*.

**Description:**

Returns the *class* of which the *object* is a *direct instance*.

**Examples:**

(class-of ’fred) *!* #<BUILT-IN-CLASS SYMBOL 610327300>

(class-of 2/3) *!* #<BUILT-IN-CLASS RATIO 610326642>

(defclass book () ()) *!* #<STANDARD-CLASS BOOK 33424745>

(class-of (make-instance ’book)) *!* #<STANDARD-CLASS BOOK 33424745>

(defclass novel (book) ()) *!* #<STANDARD-CLASS NOVEL 33424764>

(class-of (make-instance ’novel)) *!* #<STANDARD-CLASS NOVEL 33424764>

**7–88** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(defstruct kons kar kdr) *!* KONS

(class-of (make-kons :kar 3 :kdr 4)) *!* #<STRUCTURE-CLASS KONS 250020317>

**See Also:**

**make-instance**, **type-of**

**unbound-slot** *Condition Type*

**Class Precedence List:**

**unbound-slot**, **cell-error**, **error**, **serious-condition**, **condition**, **t**

**Description:**

The *object* having the unbound slot is initialized by the :instance initialization argument to **make-condition**, and is *accessed* by the *function* **unbound-slot-instance**.

The name of the cell (see **cell-error**) is the name of the slot.

**See Also:**

**cell-error-name**, **unbound-slot-object**, Section 9.1 (Condition System Concepts)

**unbound-slot-instance** *Function*

**Syntax:**

**unbound-slot-instance** *condition ! instance*

**Arguments and Values:**

*condition*—a *condition* of *type* **unbound-slot**.

*instance*—an *object*.

**Description:**

Returns the instance which had the unbound slot in the *situation* represented by the *condition*.

**See Also:**

**cell-error-name**, **unbound-slot**, Section 9.1 (Condition System Concepts)

Objects **7–89**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**7–90** Programming Language—Common Lisp