
Common Lisp Object System Specification

3. Metaobject Protocol

File Date: 1987-12-04 [http://saildart.org/MOP.2\[CLS,LSP\]](http://saildart.org/MOP.2[CLS,LSP])

This document was written by Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Contributors to this document include Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White.

CONTENTS

CONTENTS	3-2
Status of Document	3-5
Terminology	3-6
Introduction	3-7
Class Organization in the CLOS Kernel	3-9
The Classes in the CLOS Kernel	3-12
standard-class	3-13
forward-referenced-class	3-13
built-in-class	3-13
structure-class	3-13
standard-slot-description	3-13
standard-class-slot-description	3-13
standard-method	3-14
standard-accessor-method	3-14
standard-reader-method	3-14
standard-writer-method	3-14
standard-generic-function	3-14
The Named Class Definition Protocol	3-16
expand-defclass	3-16
add-named-class	3-17
class-for-redefinition	3-18
compatible-metaclass-change-p	3-19
The Named Method Definition Protocol	3-21
The Slot Parsing Protocol	3-23
slot-description-class	3-23
make-instance of slot-descriptions	3-23
The Class Update Protocol	3-25
update-class	3-25
default-class-supers	3-27
check-super-metaclass-compatibility	3-27
propagate-class-update	3-28
update-class-locally	3-28
finalize-inheritance	3-29
Adding and Removing Accessor Methods	3-29
The Slot Inheritance Protocol	3-31
The Instance Structure Protocol	3-33
Instance Allocation	3-33
Index Level Instance Allocation	3-34
Symbolic Level Instance Allocation	3-34
Slot Level Instance Allocation	3-35

Instance Access	3-35
Index Level Instance Access	3-35
Symbolic Level Instance Access	3-36
index-in-instance	3-36
Optimized Symbolic Level Instance Access	3-37
Slot Level Instance Access	3-38
slot-value-using-class	3-38
slot-boundp-using-class	3-38
slot-makunbound-using-class	3-39
slot-exists-p-using-class	3-39
Example of Using the Instance Structure Protocol	3-39
The Instance Access Optimization Protocol	3-42
The Generic Function Update Protocol	3-44
The Method Lookup Protocol	3-45
Support Functions for Method Lookup	3-46
Example of using the Method Lookup Protocol	3-47

Status of Document

WARNING: This document is uneven in its coverage. Some sections are only sketched out, and others go into significantly more detail. All suggestions for improvements in writing and/or content gratefully accepted.

Terminology

For almost all the generic functions described below, there is only one method defined in CLOS for each generic function. The specializers for these methods are usually one or more of `standard-class`, `standard-method`, `standard-generic-function`, and `standard-object`. In the case where there is only one method specified by the CLOS kernel, we use the term "standard method" to refer to this method. In the section listing all the generic functions, we actually provide the full signature for the methods.

Introduction

This document describes the Common Lisp Object System kernel (CLOS kernel). The Common Lisp Object System kernel comprises those classes, methods and generic functions which implement the Common Lisp Object System system described in chapters 1 and 2. The purpose of the CLOS kernel is to make accessible the "real" CLOS interpreter as an object oriented program written in CLOS. By "real" we mean two things. First, the interpreter we describe is the one that is running; that is, changes made to the generic functions described will effect the operations of the system. Secondly, a "real" interpreter includes not only the direct operations of the language, but provides access to mechanisms to control and support compile and run time optimizations usually hidden from users. We return to this below.

The CLOS kernel provides three levels of accessibility for users. First, it defines a programmatic interface that supports alternative interfaces for the construction of programs. Programmers need not construct and evaluate forms using the interface macros *defmethod*, *defgeneric* and *defclass*. Instead they can invoke directly the generic functions that implement some or all of the behavior associated with these forms.

Secondly, the CLOS kernel provides users a detailed model of the implementation. The kernel classes define the data structures used, and the generic functions and methods define the behaviors of the kernel. In this way it is like the metacircular interpreter for Lisp or Scheme. By understanding those interpreters, users can predict the behavior of the system. But the CLOS kernel is not just a model of the implementation. The generic functions specified are the ones that actually implement the system. The data structures are instances of classes in the CLOS kernel, and the operations of the system depend on the types and contents of these instances. Because these generic functions, classes and instances comprise the metacircular interpreter, we refer to them collectively as the metaobjects of CLOS. The *metaobject protocol* refers to the operations (generic function with methods) defined on these metaobjects.

Finally, the CLOS kernel allows for detailed control and extensions to the object system. In Lisp, this type of control is usually achieved by building an interpreter that tests many conditions explicitly, and providing the users with ways of manipulating the conditions. In this distributed, object oriented interpreter, the standard behavior is supported through methods specialized to the standard objects in the system. Specialization of the behavior of the interpreter is achieved by defining classes that are subclasses of those provided by Common Lisp Object System, and specializing some of the generic functions described. By defining methods on the generic function specialized on these new subclasses, programmers can build extensions or alternatives to the basic CLOS framework. This requires programming only the few methods that implement the difference in behavior – a standard way of extending an object-oriented program. The remaining functionality is shared. The defined classes and generic functions thus provided a distributed extensible metacircular interpreter for the Common Lisp Object System.

This chapter is organized as follows. Section *Standard Classes in CLOS* provides a detailed

description of the classes that are provided with the CLOS system. Many of these classes were mentioned in chapters 1 and 2, but here we give a full specification of their structure, and related accessors.

After describing the classes, we provide a functionally organized view of the kernel. We call each piece of this kernel a *Protocol*, a set of closely related generic functions that implement a particular feature of the Common Lisp Object System. We group these descriptions of generic functions, providing details in each section of the operation of each generic function. In describing each generic function, we distinguish two different contracts. The first is the one that a caller of the generic function is concerned with. This includes specification of the arguments for the function, the value returned, and expected side effects. A second part of the description focusses on particular methods, usually the standard method. For these methods, we specify other generic functions that are called and more explicit side effects. Finally, there are some invariants to be maintained to ensure proper operation of the kernel. This may require that some generic functions be called whenever certain conditions hold.

The section *Named Class Definition Protocol* specifies the generic function for the expansion of the *defclass* user interface macro, and the generic functions that support the behavior of named standard-classes. The closely related *Slot Parsing Protocol* creates and initializes slot description objects. The underlying anonymous classes are manipulated through the *Class Update Protocol* and the *Slot Inheritance Protocol*.

Allocation and access to instance structure is supported at three levels in the Common Lisp Object System kernel. The *Slot Level* connects the user functions *slot-value*, *slot-boundp* and *slot-makunbound*, and *allocate-instance* with the metaobjects of the CLOS kernel. At the other extreme, the *Index Level* provides a mechanism for the manipulation of indexed blocks of storage. Connecting these two is the *Symbolic Level* that maps slot names into indices. It also provides the appropriate level to provide a hook for optimized storage access.

Generic functions and methods are supported by three protocols. First is the *Named Method and Generic Function Protocol*, which supports the expansion of the user interface macros *defmethod* and *defgeneric*. Secondly, there is the *Generic Function Update Protocol*, part of which has been described earlier (add-method, remove-method and get-method). The other part of this protocol supports maintenance of the method database, and the link from classes to generic functions specialized on them. The *Method Lookup Protocol* supports generation of the code for generic functions, and the computation of effective methods for appropriate combinations of arguments.

Class Organization in the CLOS Kernel

The purpose of this section is to present the design rationale for why we defined `standard-class` as one class rather than as a subclass of the minimal class and some mixin classes for other features. The reason this belongs here is because you need to be able to understand the philosophy behind the design of the current class structure to understand how to program with it. In the following section, we present the standard classes defined in the Common Lisp Object System kernel.

In the design of the lattice of these classes, there is a major design choice that deserves some discussion. The two choices that we want to contrast are the mixin-base versus common-base designs. (The style implemented in CLOS is the common-base design.) For this part of this discussion we will only talk about design of the standard metaclasses, and then extend it to the other metaobjects.

In the mixin-base design style, features of classes would be partitioned among a number of different mixins, each of which supports a particular capability. `standard-class` would then be a direct subclass of an appropriate set of these mixins, `built-in-class` would have another set of superclasses, etc. For example, there would be a class, called say `primitive-class`, that contained just those slots and methods that any object that could be a class must have. Other mixins, including something like `obsoletable-metaclass`, `instance-slot-metaclass`, `class-slot-metaclass`, (or should it be `slotted-class`) `initarg-metaclass` would also be direct superclasses of `standard-class`. These would each bring in to `standard-class` a few slots and behaviors that were appropriate to that type of class. Other metaclasses, such as `built-in-class` and `structure-class` would have the appropriate subset of these and other mixins.

In the common-base design style, the class `standard-class` is the root of the metaclass lattice, and all other metaclasses are subclasses of this class. In this design, all the features in `standard-class` are implemented directly for `standard-class`. For subclasses of `standard-class` like `built-in-class`, features that are not allowed must be "turned off" by specializing the appropriate generic function.

It would be nice to have a picture here. We just put defclass forms for now because drawing pictures is so hard.

Common base:

```
(defclass standard-class ()  
  (..))  
(defclass built-in-class (standard-class)  
  (..))  
(defclass structure-class (standard-class)  
  (..))  
(defclass forward-referenced-class (standard-class)  
  (..))
```

```
(defclass funcallable-standard-class (standard-class)
  (..))
```

Mixin base:

```
(defclass primitive-class ()
  (..))
(defclass nameable-class (primitive-class)
  (..))
(defclass built-in-class (nameable-class)
  (..))
(defclass structure-class (slotted-class nameable-class)
  (..))
(defclass slotted-class (primitive-class)
  (..))
(defclass standard-class (obsoletable-class slotted-class)
  (..))
(defclass funcallable-standard-class (standard-class)
  (..))
```

Arguments for mixin-base style:

By dividing the functionality of standard-class into many mixins, one can potentially define mixins that capture behavior associated with a form of type abstraction. This might allow easier understanding of the capabilities of any class that includes a subset of these type mixins, since the behaviors of each mixin should be independent and additive. Use of these "type" mixins to add behavior to a class implies that there is no need to turn off unwanted behavior, since all behaviors are invoked through specific generic functions. Unwanted behaviors would have no applicable method for classes that did not include such a mixin. Test for the availability of a particular behavior can be done using typep, since behaviors are intimately tied to classes.

Arguments for common-base style:

All the classes in CLOS are implemented as instances of standard-class. By including in one class all the behavior useful for standard-class, one gets a minimal self supporting kernel. This is minimal in several senses. First, the bootstrap implementation of classes need only have the single class standard-class that is its own class. Secondly, once one has understood the features of standard-class one has a complete view of the underlying implementation. Finally, this requires fewer classes in the kernel.

Using the common-base style, one does not have to make and defend decisions about the division of behaviors among mixins. For example, should there be one mixin or two that implement the notions of class and instance slots. Should there be a separate mixin for the standard initializa-

tion mechanism. Should there be a separate mixin that support default-initforms. Having the mixins is fine if you have cut the world exactly right for the variation of class behavior one wants. However, if not, then one has the same type of problem with overriding unwanted behavior with multiple mixins.

The test for availability of a particular behavior can be done by defining a generic-function that acts as a predicate, making the default method return NIL, and defining a method for any class that supports the behavior subclasses which "turn off" that behavior must redefine this method. Although this requires some extra mechanism, it allows independent implementation of a behavior in two different classes without insisting that to support the same behavior one needs to share a common superclass.

Another reason why we want to encourage users to make their classes subclasses of either standard-class or structure-class is that we don't specify a portable mechanism for extending the type system. In other words, the user must use the mechanism provided by standard-class or structure class for determining the class of objects. Finally, there is an argument that the current Common Lisp type system is not partitioned into mixins that support common behavior (e.g. where is ceiling for float and rational). Hence trying to use this style in the rest of CLOS wouldn't make the system be uniform in any way.

For us the balance of these arguments weighed in favor of the common-base style. Thus for each of the classes in the kernel below, there is a standard-x class at the top of the lattice that captures the common behavior. For unwanted behaviors, there are methods on generic-functions that signal an appropriate error. For example, for built-in-class, an error is signalled if the user tries to update the class and provides slots for the class. In the description of the generic function in the kernel, each method that has an error case is indicated.

The Classes in the CLOS Kernel

In defining the classes that make up the kernel of CLOS, we find it useful to distinguish three categories of information associated with a class. The distinction of structural, intrinsic and interface is based on the how the information is stored, and consequently how it is retrieved and modified.

Structural Access.

Structural information is explicitly stored in a slot of an instance, it is accessed using the slot-value function. The kernel itself uses slot-value and (setf slot-value) to read and write the information. This means that changes to the the value in a slot will affect the behavior of the kernel. Even so, the kernel often expects certain updating protocols to be followed when the value of a slot is changed. In these cases, the updating protocol insures that parts of the kernel which depend on the value of the slot will behave properly after the change. An example of this is the class-direct-superclasses slot of classes. In order for the kernel methods to behave properly the value of this slot must be changed with update-class generic function rather than (setf slot-value).

Intrinsic Access:

Information in this category is accessed by calling a generic function. The kernel itself also calls this generic function to access the information. This means that a user who has defined her own class as a subclass of a kernel class can specialize these generic functions to affect the behavior of kernel code which is inherited in her own class. Intrinsic accessors fall into one of two categories:

Reading and writing done with a generic function and setf of the generic function respectively. Examples of this kind are:

class-name (setf class-name)

Reading and writing done with different generic functions. Examples of this kind are:

class-direct-subclasses add-direct-subclass remove-direct-subclass

Interface Access

There is certain kinds of information which is maintained by the CLOS kernel but which cannot be modified directly. This includes various kinds of derivative information such as back-pointers. Examples include:

class-direct-subclasses class-direct-methods class-direct-generic-functions

The kernel itself does not call these generic functions. Other programs, including parts of a given implementation's programming environment may call these generic functions. A user who has defined her own class as a subclass of a kernel class may specialize these generic functions to affect all callers of the generic function.

The following table shows the slots, intrinsic accessors and interface accessors of the kernel classes. Note that implementations are free to add other slots, intrinsic or interface accessors to any of the classes but these must not conflict with the kernel specification as described here.

Each of the intrinsic accessors, interface accessors and interface accessors needs more documentation. For now, infer the 'obvious' behavior.

standard-class

Supers: (object)

Slots: direct-superclasses direct-slots direct-class-options class-precedence-list slots class-options inheritance-finalized-p

Intrinsic Accessors: class-name (setf class-name) class-prototype (setf class-prototype) class-direct-subclasses add-direct-subclass remove-direct-subclass

Interface Accessors: class-direct-methods class-direct-generic-functions

forward-referenced-class

Supers: (standard-class)

built-in-class

Supers: (standard-class)

structure-class

Supers: (standard-class)

standard-slot-description

Supers: (object)

Slots: name initform initfunction accessors readers type

Intrinsic Accessors:

Interface Accessors: slotd-allocation

standard-class-slot-description

Supers: (standard-slot-description)

Slots: value

Intrinsic Accessors:

Interface Accessors: slotd-allocation

standard-method

Supers: (object)

Slots: specializers qualifiers function

Intrinsic Accessors: method-generic-function method-lambda-list

Interface Accessors: documentation

standard-accessor-method

Supers: (standard-method)

Slots: slot-name

standard-reader-method

Supers: (standard-accessor-method)

standard-writer-method

Supers: (standard-accessor-method)

standard-generic-function

Supers: (function object)

Slots: methods method-class argument-precedence-order method-combination-type method-combination-arguments

Intrinsic Accessors: generic-function-name add-method remove-method get-method

The following is a DAG for the classes in CLOS, as described above. Implementations are free to interpolate additional classes, provided that the order of inheritance of the classes specified is the same.

```
T
  standard-object
    standard-generic-function (also has function as superclass)
    standard-method
      standard-accessor-method
      standard-reader-method
      standard-writer-method
    standard-slot-description
      standard-class-slot-description
      standard-structure-slot-description
    standard-class
      built-in-class
      structure-class
      forward-referenced-class
      funcallable-standard-class
```

The following is the DAG of classes that correspond to the Common Lisp data types. The classes that have multiple supers are indicated with a *; the most specific super is the one that comes first (highest) in the figure below.

```
T
  function
  number
    rational
      ratio
      integer
    complex
  character
  array
    vector*
      string
      bit-vector
  symbol
    null*
  sequence
    vector*
      string
      bit-vector
  list
    cons
    null*
```

The Named Class Definition Protocol

Here there should be a little summary which describes the basic function of the named class definition protocol. It covers all the relevant generic functions: `expand-defclass`

```
add-named-class
class-for-redefinition
compatible-metaclass-change-p
slot-description-class
```

expand-defclass

In order to allow metaclasses to effect the processing of the defclass form, the expansion of the form is controlled by a generic function:

```
expand-defclass (prototype-instance name superclasses slots options environment)
```

In the syntactic processing, the `:metaclass` option is used as described in [section defclass] to determine the metaclass of the class being defined. The prototype instance of that metaclass is used as the first argument to `expand-defclass`. The remaining arguments are determined by parsing the defclass form is parsed to extract the class-name, superclass list, slot descriptions and class options. This parsing is in accordance with the specification given in chapter 2. During this parsing, the syntax of the individual slot options and individual class options is also checked to make sure it is legal. If there are any syntactic errors in the defclass form an error is signalled. Thus `expand-defclass` has control over the expansion of the defclass form but cannot change the basic syntax of defclass. Note that the `:metaclass` option will not appear in the class options passed to defclass since it is communicated by the class of the prototype-instance argument. The environment is the `&environment` argument which was passed to the defclass macro function.

```
EXPAND-DEFCLASS ((prototype-instance standard-class) name direct-superclasses direct-slots options environment)
```

The standard method for `expand-defclass` expands the defclass form into a form which includes a call to the generic function `add-named-class`. The expanded form may include other implementation-dependent code, but it will include a call to `add-named-class`. The form which calls `add-named-class` will behave as if it was:

```
'(progn
  (eval-when (compile)
    (add-named-class
      (class-prototype
        (class-named
          ',(class-name
            (class-of prototype-instance))))))
```

```

      ',name
      ',direct-superclasses
      ',direct-slots
      ',options
      ',environment))
(eval-when (load eval)
  (add-named-class
   (class-prototype
    (class-named
     ',(class-name
        (class-of prototype-instance))))
   ',name
   ',direct-superclasses
   ',direct-slots
   ',options
   nil)))

```

Example of specialization:

Suppose one wanted to have a metaclass which treated classes defined with `defclass` differently than classes defined with `add-named-class`. This metaclass might want to supplement the class options with a special marker which said that the call to `add-named-class` was the result of a `defclass` expansion. The following code would have that effect:

```

(defmethod expand-defclass ((proto-instance my-class)
                           name superclasses slots options environment)
  (call-next-method my-class
                    name
                    superclasses
                    slots
                    (cons '(defclass t) options)
                    environment))

```

add-named-class

`add-named-class` (prototype-instance name direct-superclasses direct-slots options environment)

This generic function is the programmatic interface for defining named classes. The `prototype-instance` argument should be the class-prototype of the class of the class being defined. The `direct-superclasses` argument can be a list of either symbols (class names) or class objects. The `slots` argument should be a list of slot specifications as they would appear in a `defclass` form. The `options` should be a list of the class options as they would appear in a `defclass` form. The envi-

ronment is the environment in which the definition should take place, this is used to distinguish between compiler and non compiler environments. If there is no class with the given name, a new class is created. If there is already a class with the given name the results depends on the metaclass.

ADD-NAMED-CLASS ((prototype-instance standard-class) name direct-superclasses direct-slots options environment)

this is all a mess, it doesn't lay out what happens in clear order, it just talks about one possible path through the method, it needs to be fleshed out and rationalized.

The standard method on add-named-class calls the generic function class-for-redefinition to get the class object to use for the new definition. The standard method for class-for-redefinition returns the old class object if there was one of that name, if it exists, else a new object of the same class as the proto-instance.

The standard method for add-named-class calls the generic function parse-class-slot to convert the lists describing slots into slot description objects. (see section [[[Processing slot descriptions]]]).

The standard method for add-named-class calls update-class to store the direct information (superclasses, slots, options), and to update the class lattice of subclasses to take into account the new definition (see section [[[Updating classes]]]). Finally, add-named-class stores this class in the class name table, making sure that name is now the proper name of the newly updated class. If the metaclass of the proto-instance is different than that of a previously defined class, the standard method on class-for-redefinition checks whether it is legal to change the metaclass of the existing instance. It does this using

class-for-redefinition

class-for-redefinition (prototype-instance old-class)

class-for-redefinition is called by the standard method on add-named-class when there is already a class with the given name. Class-for-redefinition is supposed to determine what class object should be used for the new definition and do any work which might be needed to prepare that object and the old class object for the redefinition. The standard-method on add-named-class expects class-for-redefinition to return the class to be used for the new definition. This can be the same as the existing class.

CLASS-FOR-REDEFINITION ((metaclass-prototype standard-class) (old-class standard-class))

this is also incomplete, it only talks about one possible path through the method it needs to be fleshed out and rationalized. For example, what happens if the metaclass isn't different. What does class-for-redefinition do. How and when does the obsolete mechanism get triggered etc.

The standard-method on class-for-redefinition ...

If the metaclass of the proto-instance is different than that of a previously defined class, the standard method on class-for-redefinition checks whether it is legal to change the metaclass of the existing instance. It does this using

Example of specialization of class-for-redefinition:

Sometimes, a user wants to declare that certain classes, when they are defined, should have a particular metaclass. This can be the case when someone takes a program which is already written and wants to compile and load it using an optimizing metaclass. The user explicitly does not want to have to edit the original defclass forms to specify the metaclass option; the user would like to use a simple macro to make this declaration. Something like:

```
(defclass-optimized A)
```

Given that the optimizing metaclass already exists and is called `optimized-class`, this can be done using class-for-redefinition. The following code will work.

```
(defclass forward-referenced-optimized-class (forward-referenced-class)
  ())

(defmethod class-for-redefinition
  ((existing-class forward-referenced-optimized-class)
   (proposed-new-class standard-class)
   name
   supers
   slots
   options)
  (change-class existing-class
    (class-prototype (class-named 'optimized-class)))
  existing-class)

(defmacro defclass-optimized (class-name)
  '(add-named-class
    (class-prototype (class-named 'forward-referenced-optimized-class))
    ',class-name
    ()
    ()
    ()
    ()))
```

compatible-metaclass-change-p

```
compatible-metaclass-change-p (class proto-new-class)
```

This is called by the standard method ... when ... it is supposed to ...

`compatible-metaclass-change-p ((old-class standard-class) (prototype-new-class standard-class))`

the standard method...

which signals an error if such a change is not allowed. Otherwise the standard method on `class-for-redefinition` calls `change-class` to update the existing instance to the new metaclass.

[[[[*Expanding the defgeneric form*]]]]

The defgeneric form is expanded into a call to ensure-generic-function, followed by a call to defmethod for each method-description clause in the defgeneric form. The behavior of ensure-generic-function is described in Chapter 2.

The Named Method Definition Protocol

The generic function `expand-defmethod` is used to compute the expansion of `defmethod` forms.

`expand-defmethod` (proto-method name qualifiers lambda-list body environment)

Whatever value `expand-defmethod` returns will be used as the expansion of the `defmethod`. Before `expand-defmethod` is called, the `defmethod` form is parsed according to the syntax defined in the CLOS spec, so methods on `expand-defmethod` can't be used to change the syntax of `defmethod`, but can be used to change the expansion for methods of a particular class. Note that for many uses, it is more appropriate to define a special method on `expand-defmethod-body`.

The arguments of the standard method for `expand-defmethod` are as follows:

proto-method:

An instance of the class of method this `defmethod` form is supposed to define. This class is the one specified by the generic function's `:method-class` option. It can be the prototype instance of the method class.

name:

The name argument to the `defmethod` form. It is the name of the generic function that this method should be added to.

qualifiers:

A list of the method qualifiers as specified in the `defmethod` form

lambda-list:

The specialized lambda-list as specified in the `defmethod` form

body:

The body as specified in the `defmethod` form.

environment:

The lexical environment the defmethod form appeared in. This is what the defmethod macro got as its &environment argument.

For a typical defmethod like:

```
(defmethod move :before ((p position) x y) "Move the position to x,y and update the display"  
(setf (pos-x p) x) (setf (pos-y p) y) (update-display))
```

The arguments would be:

```
name: MOVE qualifiers: (:BEFORE) lambda-list: ((p position) x y) body: ((setf (pos-x p) x)  
(setf (pos-y p) x) (update-display)) environment: <some structure or NIL>
```

— Example of specialization:

Suppose the user wants some methods to broadcast to other machines, but not have calls to those same generic functions that are broadcast to rebroadcast.

```
(defmethod expand-defmethod ((proto-method broadcast-method) name qualifiers lambda-list  
body environment) (call-next-method name qualifiers (add-key-argument lambda-list '(broad-  
caster nil broadcast-p)) '(multiple-value-prog1 (progn ,@body) (or broadcast-p ,(broadcast-call  
name lambda-list))) environment))
```

The standard method on expand-defmethod calls the generic function expand-defmethod-body. This generic function gets the opportunity to do extra processing of the body of the method. This processing can include things like inserting declarations, wrapping a special lexical environment around the body etc.

```
expand-defmethod-body (mex-method generic-function-name body env)
```

The mex-method argument is an instance of the same method class that the defmethod form will define (the same class as the method-instance argument to expand-defmethod). Unlike the method-instance argument, the mex-method argument has the qualifiers, lambda-list, and specializers slots filled in. This provides a general mechanism for expand-defmethod to communicate information about the method that will be defined to expand-defmethod-body. If defmethod is being evaluated at load time (as opposed to compile time), the mex object is in fact the method that will be returned by the evaluation of the defmethod form.

add-named-method and friends need to go in here. add-named-method, ensure-generic-function, get-method

The Slot Parsing Protocol

Standard classes store information about two distinct sets of slots. The first is the set of slots defined in the class proper. The second is the total set of slots the class has, this includes inherited and locally defined slots. Both of these sets are stored as lists of slot description objects.

As part of defining a class, the slot specifications which appear in the defclass form must be converted to a list of slot description objects. This conversion process is done using the slot parsing protocol.

The slot parsing protocol is quite simple. It only contains only three steps: normalization of the slot-specification, a call to the generic function slot-description-class and a call to make-instance.

Normalization of the slot specifications converts the three kinds of slot specifications which can appear in defclass to a single 'pure plist' form. This is done as follows:

if the slot-specification is a symbol: <slot-name> the normalized slot specification is: (:name <slot-name>)

if the slot-specification is a list like: (<slot-name> . <slot-options-and-values>) the normalized-slot-specification looks like: (:name <slot-name> . <slot-options-and-values>)

none of this talks about how and when the :slot-initform-function value gets filled in. defclass does it, other callers of the slot-parsing protocol are expected to do it as well

slot-specifications are always parsed with respect to the class they specify a slot for. This allows the class the slot description is being produced for to control the class of the slot description object itself. This slot-description-class generic function is called with the class and the normalized slot specification to determine the class of slot description which should be produced for the class.

Once the appropriate class for the slot description has been determined, the actual parsing is achieved by applying make-instance to the class and the normalized slot specification.

This means that the legal set of slot option names for a given class of slot-description is the same as the legal set of initarg names for that class. See lambda-list-congruence rules.

slot-description-class

The slot-description-class generic function..

SLOT-DESCRIPTION-CLASS ((class standard-class) normalized-slot-specification)

slot-description-class

make-instance of slot-descriptions

The standard method ...

MAKE-INSTANCE ((class standard-slot-description))

This fills in the slots of the class standard-slot-description with the appropriate values in the initargs

MAKE-INSTANCE ((class standard-class-slot-description))

This fills in the slots and also fills in the class-value slot.

The Class Update Protocol

This protocol supports the inheritance contracts between the direct-slots and direct-superclasses of a class, and all the slots and superclasses of that class and its subclasses. It consists of three parts: entry, change propagation, and finalization of inheritance. This protocol works on classes as anonymous objects, and makes no use of the names of classes.

The entry part of the class update protocol is implemented by the generic functions `update-class`, `default-class-supers` and `legal-class-option-p`. A call to `update-class` is the only guaranteed consistent way to update the slots of standard class. The keyword arguments of `update-class` allow specification of new direct superclasses, new direct slots and new options. The contract of `update-class` is to normalize its the arguments, store the new information locally, and cause this class and its subclasses to be updated using the newly provided information.

The change propagation part of this protocol is implemented by the generic functions `propagate-class-update` and `update-class-locally`. It ensures that the class and all its subclasses are notified of a change higher in the class lattice, and updates slots, class precedence and/or options that are effected by the newly input data. To make the local updates, `update-class-locally` uses the generic functions `compute-class-precedence-list`, `collect-slots`, `add-reader-method`, `add-writer-method`, `remove-reader-method`, and `remove-writer-method`. Change propagation treats classes that are instances of `forward-referenced-class` as ordinary classes, with no superclass but T, and no slots.

We have to make sure that the rest of the document is consistent with this use of forward-referenced classes

The cache finalization part of the class upate protocol is implemented by the generic function `finalize-inheritance`, and it allows implementations to update any caches used for method and slot position lookup. It should be called after `update-class` has returned, and should be called before an instance is made of any updated class, or any method run that is specialized on an updated-class. The finalization should warn about any classes in the superclasses of a class that are instances of `forward-referenced class`.

update-class

The generic function `update-class` is used to update existing classes. It ignores any information about class names, and deals with the classes as anonymous objects. This generic function also installs newly defined classes. This is the only interface to updating a class. Programs that try to change the structure of the class object by directly changing values of slots will get what they

deserve.

The following defines the argument list of the generic function:

```
update-class (class
              &rest key-arguments
              &key (direct-superclasses () new-supers-p)
                  (direct-slots () new-slots-p)
                  (options () new-options-p)
              &allow-other-keys)
```

update-class (class built-in-class) &rest key-arguments [Primary Method]

update-class (class structure-class) &rest key-arguments [Primary Method]

These two methods signal an error since neither built-in-class nor structure-class are allowed to be updated.

update-class (class standard-class) &rest key-arguments &key (direct-superclasses () new-supers-p) (direct-slots () new-slots-p) (options () new-options-p) [Primary Method]

In this method on standard-class, if direct-superclasses is given, the list is normalized by a call to *default-class-supers*; that is, the list of superclasses actually used to update the class is the value of: (default-class-supers class supplied-supers).

This generic function implements the feature that standard-classes have the class named object as their default superclass if () is provided as the superclasses list (say by the defclass form).

For each direct superclass, the generic function check-super-metaclass-compatibility is used to check if the given superclass has a metaclass compatible with the class being defined. The standard method on check-super-metaclass-compatibility signals an error if there is a compatibility problem.

For each of the options provided, the generic function legal-class-option-p is called as follows: (legal-class-option-p class option-keyword).

The standard method for legal-class-option-p signals an error if the option-keyword is not one of the options described in chapter 2.

If there has been no error, for each class in the current and previous list of direct superclasses, the standard method on update-class updates their direct-subclass list to maintain the correct back pointers. The newly provided direct superclasses are stored in the slot *direct-superclasses*.

To make the changes take effect in the class hierarchy, this method on update-class calls the generic function propagate-class-update, which starts at the changed class, and walks the class

tree below (it may visit some subclasses more than once). On the walk the standard method on `propagate-class-update` calls the generic function `update-class-locally` to cause the local effects to be combined with information from the class lattice.

The generic function returns the modified class.

`update-class`

default-class-supers

This generic function returns a list of classes to be used as the direct-superclasses of a class. It is called by the `update-class` method on `standard-class`.

default-class-supers (`class standard-class`) `supplied-supers`) [*Primary Method*]

If `supplied-supers` is `NIL`, or the list just containing the class named `T`, then this method returns a list containing the class named `Object`. Otherwise it returns its argument *supplied-supers*.

default-class-supers (`class structure-class`) `supplied-supers`) [*Primary Method*]

If `supplied-supers` is `NIL` then this method returns a list containing the class named `T`. Otherwise it returns its argument *supplied-supers*.

Example:

Suppose we have `loops-class` as a subclass of `standard-class`, and we want all instances of `loops-class` to have the class named `loops-object` as their default super.

```
(defclass loops-class (standard-class) ())

(defmethod default-class-supers ((class loops-class) supplied-supers)
  ;; Implement the rule that where standard-class would
  ;; have made the superclasses be a list of the class
  ;; object, we make them be a list of the class loops-object.
  (let ((default (call-next-method)))
    (if (and (null (cdr default))
              (eq (car default) (class-named 'object)))
        (list (class-named 'loops-object)
              default))
        default)))
```

check-super-metaclass-compatibility

For each direct superclass, the generic function `check-super-metaclass-compatibility` tests whether the superclass provided has a metaclass compatible with the class being defined.

the default method signals an error unless the metaclasses are EQ

check-super-metaclass-compatibility (class superclass)

Question: can check-super-metaclass-compatibility have a side effect on any class – that is make things compatible by changing the metaclass of one or more classes.

propagate-class-update

The generic function `propagate-class-update` modifies a class and all its subclasses in the lattice below the changed class to take into account new structure of the class. The generic function `propagate-class-update` takes arguments that indicate where the changes actually happened and what it was. There is only one method on `propagate-class-update`, the standard method specialized with `(class standard-class)`, and we describe that method here. It is highly unlikely that there would be a good reason to specialize this method that walks the subclass lattice.

This method is called by the standard method for `update-class`.

```
propagate-class-update
  (class changed-class
   &rest key-arguments
   &key (direct-superclasses () new-supers-p)
        (direct-slots () new-slots-p)
        (options () new-options-p))
```

The class argument for the standard method is the class in the lattice to be updated. The `changed-class` argument is the class that was the entry point, and hence started the propagation. Thus `(eql changed-class class)` is T only for the initial call to `propagate-class-update` from `update-class`. The other arguments are the same as those passed to `update-class`. Although for the standard methods, these arguments are not used, it is the appropriate information to pass down.

At each class, `propagate-class-update` calls the generic function `update-class-locally` to do the local updating. It passes all of its given arguments to that generic function.

update-class-locally

This generic function is responsible for updating a class in the lattice that has been changed either directly (by `update-class`) or indirectly, by being a subclass of a directly changed class.

```
update-class-locally
  (class changed-class
   &rest key-arguments)
```

```
&key (direct-superclasses () new-supers-p)
      (direct-slots () new-slots-p)
      (options () new-options-p))
```

There is only one method on `propagate-class-update`, the standard method specialized with `(class standard-class)`, and we describe that method here.

If `new-supers-p` is true (the supers of some class has changed), the standard method on `update-class-locally` sets the value of the slot `class-precedence-list` to the result obtained by calling the generic function `class-precedence-list`.

If `(eql class changed-class)` then the standard method on `update-class-locally` processes the class-options, and stores the new options in the slot `options`.

If `new-slots-p` is true and `(eql class changed-class)`, the standard method on `update-class-locally` removes no longer required reader and writer method, and adds newly required methods using the generic functions `remove-reader-method`, `remove-writer-method`, `add-reader-method`, `add-writer-method`. It then stores the new direct-slots in the slot in the class called `direct-slots`.

If either `new-supers-p` or `new-slots-p` is true, the method on `update-class-locally` sets the value of the slot `slots` in the class to the result obtained by calling the generic function `collect-slots`.

If the result of calling `collect-slots` is a list that specifies a different list of instance slots, and the class has any instances, then the standard method calls `make-instances-obsolete` on the class. The generic function `class-has-instances-p (class)`

tests if a class has (ever had) instances. This predicate returns true if the class, with this set of instance slots, has ever had an instance created.

The standard method sets to NIL the slot `inheritance-finalized-p`. This slot is used as a flag for the method on `standard-class` for `make-instance` to determine if that method should call `finalize-inheritance`.

finalize-inheritance

The generic function `finalize-inheritance` is called by the method on `make-instance` that is specialized to `standard-class` if the slot `inheritance-finalized-p` is NIL. Users with special optimization requirements can write methods to precompute information based on inherited information, and be assured they will be called when ever changes occur.

when else is it called. When update-obsolete-instance is called? Whenever a method is defined? Does it walk the subclasses?

finalize-inheritance (class standard-class))

[Primary Method]

The standard method warns if any of the superclasses are forward referenced classes. It can be called by users to precompute information that may make instance creation faster. The standard method on `finalize-inheritance` sets the flag in the slot *inheritance-finalized-p* to T.

Adding and Removing Accessor Methods

As part of the processing of the class update, readers and accessors for particular slots may have to be added. If there was a previous definition of the class being defined, some readers and writers may need to be removed. The following generic functions are used to implement this facility.

```
add-reader-method (class slotd generic-function)
```

```
add-writer-method (class slotd generic-function)
```

```
remove-reader-method (class slotd generic-function)
```

```
remove-writer-method (class slotd generic-function)
```

The Slot Inheritance Protocol

The total set of slots for any given class is computed by combining the locally defined slots for the class and all of its superclasses. For standard classes, this combination proceeds according to the rules described in chapter 1. This combination is implemented by the slot inheritance protocol.

The slot-inheritance protocol is a two level protocol.

collect slotds collects up all the slotds and then calls *compute-effective-slotd* to condense them into one slotd. Need to make some statement about the ordering constraints on what *collect slotds* will do. Perhaps there aren't any.

The computation of the set of slots and their descriptions are controlled at two levels. For each slot, the set of slots with that name, ordered by class precedence list (most specific first), is used to compute an effective slot description for the slot locally, using

```
compute-effective-slotd (class slotds)
```

The standard method for this generic function supports the inheritance of slot options that is described in Chapter 1. It returns a slot-description object that can be used locally.

The generic function *collect-slotds* (*class local-slots cpl*) collects an ordered list of effective slot descriptions for this class. It takes the local-slots as an argument, and recursively builds up the list of all slots that need to be in this class. It calls *compute-effective-slotd* to combine multiple definitions of a single slot found in classes on the class precedence list.

Example of Specialization:

Suppose a user wanted to define a new metaclass which implemented a different rule for the inheritance of the `:type` slot option. This new rule might want to say that a subclass must specify a type which is at least as specific as the type specified by the any of the superclasses. If none of the superclasses specified a type, the local class can either not specify a type or specify any type it likes.

```
(defmethod compute-effective-slot-description ((class my-class)
                                             slot-descriptions)
  (when (car slot-descriptions)
    (when (slot-boundp (car slot-descriptions) 'type)
      ;; The class has a local specification for this slot and
      ;; the :type option is specified in that specification.
      ;; Make sure the specified type is at least as specific
      ;; as all the other types specified.
      (let ((local-type (slot-value (car slot-descriptions) 'type)))
        (dolist (super-slot (cdr slot-descriptions))
          (when (slot-boundp super-slot 'type)
```

```
(unless (subtypep slotd-type (slot-value super-slot 'type))
  (error "~S is not a subtype of ~S"
    local-type
    (slot-value super-slot 'type))))))
(call-next-method))
```

The Instance Structure Protocol

This section does not yet include descriptions of any setf functions. While reading it, you should assume that the obvious functions have setf functions with the obvious meanings.

It sure would be real nice if there were some abbreviated way to discuss setf functions. Putting them in line all the time is real painful and interrupts the flow of the text. I don't think its possible though.

Metaclasses determine the structure of their meta-instances. This includes allocating the memory for and managing the layout of the instance. This is handled by the instance structure protocol.

The instance structure protocol has several levels. At the lowest level, it permits the allocation and access to two kinds of instances: standard-class and structure-class. At this level, instances appear to be vector-like blocks of memory with the additional property that the type system (including class-of) can determine their class. At this level, positive integers called indexes can be used to access the elements of the instance. For this reason this is called the Indexed Level of instance structure. At this level there is also support for implementation specific mechanisms for controlling the packing and garbage collection parameters of this access.

At the next level, there is a mapping from symbolic descriptions of the elements of an instance to the index which specifies the location of that element at the lowest level. This is called the Symbolic Level of index structure. For standard class and structure class this mapping is from the symbol which names a slot to the actual index in the instance where the slot is stored.

At the highest level, the instance appears to contain a set of slots as described in chapter 1. This is called the Slot Level of index structure. Since only metaclass programmers make use of the levels below the slot level, it is often useful to think of this as the user level.

User defined metaclasses can define new elements of instances at any of the three levels.

The rest of this section describes these three levels and describes how the standard-class and structure-class metaclasses use them.

There is a bit of design rationale I would like to put in here, but I can't figure out how. Specifically, the reason we specify just the two meta-instance types and don't specify a general way to allocate new kinds of instance is that we don't want to have to specify a more powerful portable way to extend an implementation's type system. Also, it turns out that this provides as much power as a seemingly more general portable mechanism would provide. This is because all the more general schemes I have been able to come up with turn out to be essentially equivalent to this.

Instance Allocation

At the indexed level of the protocol, an instance of a certain type and size is allocated by calling the appropriate allocation function. At the symbolic level, information about what will be stored in the instance – the slots – is used to determine the appropriate size for the instance. At the slot or user level, the metaclass determines the kind of instance which is allocated. the instance.

Index Level Instance Allocation

At the lowest level, there are two functions used for allocating instances. These are `allocate-standard-instance` and `allocate-structure-instance`. Each of these takes as arguments an instance size. The size indicates the size of the elements in index numbers (see the low level instance access section). The optional argument `storage-information` is an implementation-specific value which can be used to specify packing and garbage collection information for the instance.

allocate-standard-instance *class size &optional storage-information* [Function]

allocate-structure-instance *class size &optional storage-information* [Function]

The value returned is the newly allocated instance.

standard-instance-p *thing* [Function]

Returns true if *thing* is a standard instance (was created by a call to `allocate-standard-instance`).

structure-instance-p *thing* [Function]

Returns true if *thing* is a structure instance (was created by a call to `allocate-structure-instance`).

Implementations are free to make standard instances and structure instances be the same but they must do so consistently. In other words if any value returned by `allocate-structure-instance` is `standard-instance-p` they must all be and vice versa.

Symbolic Level Instance Allocation

At the symbolic level, information about what is to be stored in the instances of the class is used to determine the appropriate size for the instance. This level acts as a translation between the information stored at the slot level (about what slots the instance has) and the indexed level.

Methods on `compute-instance-size` take care of this conversion. The kernel methods on `compute-instance-size` return a size greater than or equal to the number of slots that must be stored in the instance.

compute-instance-size (`class standard-class`) [Primary Method]

This method returns a number greater than or equal to the number of `:instance` slots of the class.

compute-instance-size (class structure-class)

[*Primary Method*]

This method returns a number greater than or equal to the number of slots of the class.

Slot Level Instance Allocation

At the slot level, instances are allocated using the generic function `allocate-instance`. Methods on `allocate-instance` take care of calling the appropriate index level instance-allocation function. These methods determine the appropriate size for the instance by calling the `compute-instance-size` generic function.

allocate-instance (class standard-class) &key &allow-other-keys

[*Primary Method*]

This method allocates instances using the `allocate-standard-instance` function. The size argument to the `allocate-standard-instance` function is determined by calling the `compute-instance-size` generic-function with the class as its only argument. Whether the `packing-information` argument to `allocate-standard-instance` is supplied is implementation dependent.

allocate-instance (class structure-class) &key &allow-other-keys

[*Primary Method*]

This method allocates instances using the `allocate-standard-instance` function. The size argument to the `allocate-standard-instance` function is determined by calling the `compute-instance-size` generic-function with the class as its only argument. Whether the `packing-information` argument to `allocate-standard-instance` is supplied is implementation dependent.

Instance Access

The instance access part of the instance structure protocol operates at the same three levels as the instance allocation part does.

Index Level Instance Access

At the index level, instances are accessed as if they were vector-like blocks of memory. They are accessed with functions specific to the kind of instance being accessed.

standard-instance-ref *instance index* *Optional storage-info*

[*Function*]

Takes a `standard-instance` and returns the element stored at index number `index`. If the instance argument is not a `standard-instance` the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the `storage-info` argument to provide mechanisms for data packing and garbage collection control.

structure-instance-ref *instance index* *Optional storage-info* [Function]

Takes a structure-instance and returns the element stored at index number index. If the instance argument is not a structure-instance the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the storage-info argument to provide mechanisms for data packing and garbage collection control.

standard-instance-boundp *instance index* *Optional storage-info* [Function]

If there is a value stored in index number index of the standard-instance instance this returns true. If there is no value stored there returns false. If the instance argument is not a standard-instance the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the storage-info argument to provide mechanisms for data packing and garbage collection control.

standard-instance-makunbound *instance index* *Optional storage-info* [Function]

Causes there to be no value stored in element number index of the standard instance standard-instance. If the instance argument is not a standard-instance the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the storage-info argument to provide mechanisms for data packing and garbage collection control.

Symbolic Level Instance Access

The symbolic storage layer provides indirection from symbolic descriptions of an element of an instance to the index number at which that element is stored. Standard-class and structure-class use this layer to map from slot names to the index number at which the slot is stored.

The symbolic storage layer is designed to provide an interface to this symbolic mapping which can be used by metaclass programmers to take advantage of implementation specific optimization mechanisms.

index-in-instance

The generic function index-in-instance is used to convert a symbolic description of an element of an instance to the index number at which that element is stored. The kernel methods for index in instance support symbolic descriptions which are symbols, specifically slot-names. User defined methods can extend this mechanism to use other kinds of symbolic descriptions.

index-in-instance (class standard-class) instance description [*Primary Method*]

If description is a symbol which is the name of a :instance slot in the class, returns the index number at which that slot is stored. Otherwise this returns nil. For more information about exactly how the index number is computed see the section computing slot inheritance.

index-in-instance (class structure-class) instance description [*Primary Method*]

If description is a symbol which is the name of a slot in the class, returns the index number at which that slot is stored. Otherwise this returns nil. For more information about exactly how the index number is computed see the section computing slot inheritance.

Optimized Symbolic Level Instance Access

The standard-instance-access function provides the basic interface to the implementation-specific standard instance access optimization. This function is just a simple combination of more primitive instance access mechanisms, but it is designed to be the place where the implementation provides its optimization. In most implementations calls to this function will be replaced by just a few instructions. Specific implementations of CLOS are expected to implement their instance access optimization by optimizing these functions and then using the Instance Access Optimization Protocol to convert instance accesses to calls to this function.

The effective definition standard-instance-access is:

```
(defun standard-instance-access
  (instance description trap missing-function not-bound-function)
  (let* ((class (class-of instance))
        (index (index-in-instance class description)))
    (cond ((null index)
          (funcall missing-function instance description))
          ((null (standard-instance-boundp instance index))
           (funcall not-bound-function instance description))
          (t
           (standard-instance-ref instance index))))))
```

In order to support the optimization there is a contract between standard-instance-access and the kernel methods which provide the class updating protocol. Specifically, standard-instance-access is allowed to call index-in-instance at finalize-inheritance time. This means that any user defined methods on index-in-instance which might affect uses of standard-instance-access must guarantee that their value only changes when a class update happens.

In order to allow flexible use of the optimization standard-instance-access provides, there is a mechanism for deoptimizing calls to standard-instance-access for a particular class. This mechanism causes all the calls to standard-instance-access for a particular class to call the trap

function instead. The trap function received the instance and the description as its arguments.

deoptimize-standard-instance-access *class* [Function]

...

There is a similar set of functions for structure instance. But, structure-instance-access doesn't support deoptimization, and structure-instance-access is free to call index-in-instance at load time. This reflects the different performance optimization structure-class provides.

Slot Level Instance Access

At the highest level, instance access is in terms of slots. The basic functions for accessing the slots of an instance are described in chapters 1 and 2. In this section we describe the generic functions underlying those functions. The functions rely entirely on these generic functions to implement their behavior. Each of the corresponding functions calls the generic functions directly, the only difference is that the class of the object is included as the first argument to the generic function. In the case of setf functions, the class is the second argument. For example the slot-value and (setf slot-value) functions are implemented in terms of slot-value-using-class and (setf slot-value-using-class) as follows:

```
(defun slot-value (instance slot-name)
  (slot-value-using-class (class-of instance) instance slot-name))

(defun (setf slot-value) (new-value instance slot-name)
  (setf (slot-value-using-class (class-of instance) instance slot-name)
        new-value))
```

slot-value-using-class

The generic function **slot-value-using-class** is called by the function **slot-value**.

slot-value-using-class returns the value of the slot with the given name. All methods on slot-value-using-class call slot-missing if the slot with the given name does not exist. Some methods on slot-value-using-class may do additional checks, for example to see if the slot is bound.

slot-value-using-class (class standard-class) instance slot-name [Primary Method]

Returns the value of the slot named slot-name if such a slot exists and is bound. If the slot does not exist calls slot-missing. If the slot exists but is not bound calls slot-unbound.

slot-value-using-class (class structure-class) instance slot-name [Primary Method]

Returns the value of the slot named slot-name if such a slot exists. If the slot does not exist calls

slot-missing.

slot-boundp-using-class

The generic function **slot-boundp-using-class** is called by the function **slot-boundp**.

The generic function **slot-boundp-using-class** tests whether a specific slot in an instance of a given class is bound. Not all metaclasses support this operation.

slot-boundp-using-class (class standard-class) instance slot-name [*Primary Method*]

If a slot with the given name exists, and that slot is bound, returns true. If a slot with the given name exists and that slot is not bound returns false. If no slot with the given name exists the function slot-missing is called.

slot-boundp-using-class (class structure-class) instance slot-name [*Primary Method*]

If a slot with the given name exists, returns true. If no slot with the given name exists the function slot-missing is called.

slot-makunbound-using-class

The generic function **slot-makunbound-using-class** is called by the function **slot-makunbound**.

For metaclass which support unbound slots, the generic function **slot-makunbound-using-class** restores a slot to its unbound state. Attempting to read a slot after it has been made unbound will result in a call to **slot-unbound**.

slot-makunbound-using-class (class standard-class) instance slot-name [*Primary Method*]

If a slot with the given name exists in the class the slot is restored to its original unbound state. If there is no slot with the given name in the class calls slot-missing.

slot-makunbound-using-class (class structure-class) instance slot-name [*Primary Method*]

Since this operation is not supported by structure-class, this method signals an error.

slot-exists-p-using-class

The generic function **slot-exists-p-using-class** is called by the function **slot-exists-p**.

The generic function **slot-exists-p-using-class** tests whether a slot by the given exists in the instance.

`slot-exists-p-using-class` (class standard-class) instance slot-name [*Primary Method*]

If either a :instance or :class slot slot with the given name exists in the class returns true. Otherwise returns false.

`slot-exists-p-using-class` (class structure-class) instance slot-name [*Primary Method*]

If a slot with the given name exists in the class returns true. Otherwise returns false.

Example of Using the Instance Structure Protocol

This example also makes use of the Optimizing Instance Access Protocol, to fully understand it see that section also.

```
;;;
;;; Define a new metaclass faceted-slot-class which provides one facet
;;; for each :instance slot in the class. In the instances, the facets
;;; are stored between the slots, each facet comes immediately after its
;;; corresponding slot.
;;;

(defclass faceted-slot-class (standard-class) ())

(defmethod compute-instance-size ((class faceted-slot-class))
  (* 2 (call-next-method)))

(defmethod index-in-instance ((class faceted-slot-class) description)
  (cond ((symbolp description)
        (* 2 (call-next-method)))
        ((and (listp description)
              (eq (car description) 'facet))
         (1+ (index-in-instance (cadr description))))
        (t
         (error "Don't understand the description ~S." description))))

(defun slot-facet (instance slot-name)
  (standard-instance-access instance (list 'facet slot-name)))

(defun (setf slot-facet) (new-value instance slot-name)
  (setf (standard-instance-access instance (list 'facet slot-name))
        new-value))

(defmethod optimize-instance-access
  ((class faceted-class) function args instance-arg context)
  (cond ((or (equal function 'slot-facet)
```

```
      (eq function #'slot-facet))
    '(standard-instance-access ,(car args)
      '(facet ,(cadr args))
      #'slot-facet
      #'facet-unbound
      #'facet-missing))
  ((or (equal function '(setf slot-facet))
        (eq function #'(setf slot-facet))))
    '(setf (standard-instance-access ,(cadr args)
      '(facet ,(cadr args))
      #'slot-facet
      #'facet-unbound
      #'facet-missing))
      ,(car args)))
  (t
    (call-next-method)))
```

The Instance Access Optimization Protocol

As described in chapters 1 and 2, most code access instances at the slot level. But, as described in the Instance Structure Protocol section, a call to `slot-value` results in a call to the `slot-value-using-class` generic function which then calls `standard-instance-access`. If every call to `slot-value` had to do this generic function call, slot access would be too slow.

To solve this problem, CLOS provides a mechanism for optimizing calls to `slot-value`. At compile-time, this mechanism optimizes those to `slot-value` where it is possible to convert the call to a use of `standard-instance-access`. This requires that the compiler be able to ascertain the class of an instance (it can be a subclass of that class at run-time).

This mechanism is general enough that it can be used to optimize any access to instances whose class is known at compile time.

There is a notion which needs to be defined here. It is the concept of a context in which a call to `standard-instance-access` can be optimized. I don't understand quite how to define this. It needs to be phrased in a portable way.

The fundamental hook for this mechanism is the `optimize-instance-access` generic function. This generic function is called on any instance accessing form which, if it were converted to a call to `standard-instance-access` could be further optimized. This gives the metaclass programmer an opportunity to optimize any instance accessing form into a call to `standard-instance-access` whenever it would do any good.

`optimize-instance-access` receive as arguments the class of the instance being accessed (or a superclass) the function being called on the instance, all the arguments to the function, the particular one of those arguments which will be the instance at run-time and information about the context the access is in. The context will be the symbol `:effect` if the compiler is guaranteeing that this access is for effect only.

`optimize-instance-access` (class standard-class) function arguments instance-argument context [Primary Method]

```
(defmethod optimize-instance-access
  ((class standard-class) function args instance-arg context)
  (cond ((or (equal function 'slot-value)
            (eq function #'slot-value))
        '(standard-instance-access ,(car args)
                                   '(facet ,(cadr args))
                                   #'slot-value
                                   #'slot-unbound
                                   #'slot-missing)))
```

```
((or (equal function '(setf slot-value))
      (eq function #'(setf slot-value))))
  '(setf (standard-instance-access ,(cadr args)
                                     '(facet ,(cadr args))
                                     #'slot-value
                                     #'slot-unbound
                                     #'slot-missing)
         ,(car args)))
(t nil)))
```

When a metaclass optimizes slot accesses, it may do so in a way that makes them deoptimizable. A deoptimized slot access is one that goes through the full access protocol rather than the optimized access. If a metaclass can deoptimize its slot accesses, it should return true from `can-deoptimize-slot-accesses-p`, if not it should return false.

can-deoptimize-slot-accesses-p (class standard-class) *[Primary Method]*

Returns true.

can-deoptimize-slot-accesses-p (class structure-class) *[Primary Method]*

Returns false.

deoptimize-slot-accesses (class standard-class) *[Primary Method]*

Deoptimizes its optimized slot accesses by calling `deoptimize-standard-instance-access`.

can-deoptimize-slot-accesses-p (class structure-class) *[Primary Method]*

Signals an error.

The Generic Function Update Protocol

The generic functions `get-method`, `add-method` and `remove-method` previously described provide and interface for directly accessing and manipulating the methods of a generic function.

In order to update links between classes and generic functions that have used the classes as specializers, the standard method `on-generic-function-changed` calls

`add-method-on-specializer(method standard-method specializer)`

`remove-method-on-specializer(method standard-method specializer)`

For other changes to a generic function, the `update-generic-function` generic function is used. It is called with the generic function as a first argument and keywords describing the change that should be made to the generic function.

update-generic-function *generic-function* &key `class` `method-combination-type` `method-combination-arguments` `argument-precedence-order` [*Generic Function*]

The standard method makes the change as specified and then calls `compute-discriminator-code` to compute new discriminator code for the generic function.

The Method Lookup Protocol

When a generic function is called with particular arguments, it must determine the code to execute. This code is called the *effective method* for those arguments. The effective method is a *combination* of the applicable methods in the generic function. A combination of methods is a Lisp expression that contains calls to some or all of the methods. If a generic function is called and no methods apply, the generic function **no-applicable-method** is invoked.

When the effective method has been determined, it is converted to an actual function and the actual function is applied to the same arguments as were passed to the generic function. Whatever values it returns are returned as the values of the generic function.

The specification for the precise way the kernel computes the effective method appears in chapter 1. This section describes the protocol used to compute and invoke the effective method.

The key component in this protocol is the discriminator code for the generic function. The discriminator code for a generic function is called whenever the generic function is called; it computes the effective method and invokes it. The discriminator code for a generic function is computed each time the generic function changes. When the generic function itself is called, the pre-computed discriminator code is called. The protocol described here is used to compute the discriminator code, thus the protocol described here is not invoked when the generic function is called, it is invoked whenever the generic function is updated. This allows method lookup to be a fast operation.

The standard-method on update-generic-function calls compute-discriminator-code whenever the generic function changes. compute-discriminator-code is expected to return the discriminator code for the generic function. This discriminator code must be valid until the next time update-generic-function is called.

The standard-method on compute-discriminator-code provides the documented behavior of computing the effective method and calling it. It does this by providing a second layer of protocol, specifically the compute-effective-method generic function. compute-effective-method is called by the standard-method on compute-discriminator-code to compute the effective method for a set of applicable methods.

There are also several support functions supplied by the kernel to assist users in extending the method lookup protocol. These support functions implement certain key parts of the kernel method lookup behavior.

compute-discriminator-code *generic-function* [*Generic Function*]

compute-discriminator-code (generic-function standard-generic-function) [*Primary Method*]

```

(defmethod compute-discriminator-code
  ((generic-function standard-generic-function))
  #'(lambda (&rest args)
      (let* ((lambda-list (slot-value generic-function 'lambda-list))
             (methods (compute-applicable-methods generic-function args))
             (function
              (make-effective-method-function
               generic-function
               (compute-effective-method
                generic-function
                methods
                (slot-value generic-function
                            'method-combination-type)
                (slot-value generic-function
                            'method-combination-arguments))))))
          (check-keyword-arguments lambda-list methods args)
          (apply function args))))

```

compute-effective-method *generic-function applicable-methods method-combination-type method-combination-arguments* [*Generic Function*]

compute-effective-method (generic-function standard-generic-function) applicable-methods method-combination-type method-combination-arguments [*Primary Method*]

Actually, there is one method here for each pre-defined method-combination-type. This needs to be explained in terms how define-method-combination expands into a defmethod for compute-effective-method.

Support Functions for Method Lookup

In order to help the user use the method lookup protocol the CLOS kernel provides some helpful support functions.

compute-applicable-methods *generic-function arguments* [*Function*]

Given a generic function and a set of arguments, this uses the standard rules to determine the ordered set of applicable methods.

compute-combination-points *generic-function* [*Function*]

Computes all the combination points for this generic functions. That is all the points at which (if you are using combined-methods) methods must be combined. For each point it also provides the

ordered set of methods applicable at the point.

This may sound like it is too implementation specific to be useful in the metaobject protocol, but I think that is because of the way I am describing it. I believe a lot of method lookup hackers are going to want to compute this, and given that it is hard to compute accurately and quickly I think we should provide it.

check-keyword-arguments *generic-function lambda-list methods args* [Function]

This implements the keyword congruence rules specified in chapter 1. If the keyword arguments in args are OK, this returns t. Otherwise it signals an error.

make-method-call *method-list &key operator identity-with-one-argument* [Function]

This is documented in chapter 2.

make-effective-method-function *generic-function effective-method-body* [Function]

This takes the effective method body as computed by compute-effective-method-body and converts it to a function which implements the effective method. This function accepts the same arguments the generic function accepts. This function does the standard keyword congruence checking. This function arranges to call all the methods “as if with :allow-other-keys t”.

Basically, make-effective-method-function and make-method-call are the ones that have the contract that makes calling methods work. They communicate the information about what parameters the arguments will be bound to, how to hack :allow-other-keys t etc.

I am concerned that actually we have to get rid of make-method-call and have a call-method special form. Otherwise, I don't know how someone is going to build a portable stepper for standard generic functions.

Example of using the Method Lookup Protocol

This example defines a special class of tracing generic function. This class of generic function provides two kinds of tracing facilities. The first kind allows the user to specify that calls to particular generic functions should cause breakpoints. The second allows the user to specify that calls to the effective method for particular sets of methods should cause breakpoints.

```
(defvar *trace-generic-functions* ())
(defvar *trace-effective-methods* ())

(defclass tracing-generic-function (standard-generic-function) ())

(defmethod compute-discriminator-code ((gf tracing-generic-function))
  (let ((real-discriminator-code (call-next-method)))
```

```
#'(lambda (&rest args)
  (when (member gf *trace-generic-functions*)
    (break "The generic function ~S is one of the generic~%~
           functions on *trace-generic-functions*"
           gf))
  (apply real-discriminator-code args))))

(defmethod compute-effective-method ((gf tracing-generic-function)
                                     methods
                                     method-combination-type
                                     method-combination-arguments)
  '(progn
    (when (member ',methods *trace-effective-methods* :test #'equal)
      (break "The set of methods ~S is one of the sets of~%~
             methods on *trace-effective-methods*."
             ',methods))
    ,(call-next-method)))
```