Common Lisp Object System Specification

3. Metaobject Protocol

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

Terminology

For almost all the generic functions described below, there is only one method defined in CLOS. 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

The Common Lisp Object System can be described at three levels. These levels are a set of mutually supporting facilities for use at different levels of sophistication. The first level, the programmer's interface, provides a convenient syntax for defining classes and methods, which is all most users need to know to construct object oriented programs in Common Lisp. The second level consists of the description of the functions that underlie the basic facilities. At this level, there is a complete separation of objects and names, with all basic operations applicable to strictly anonymous objects. This level is useful for programmers who wish to provide an alternative interface to the facilities or who are building program development environments.

The third level, the Common Lisp Object System kernel (CLOS kernel), describes CLOS as an object oriented program. 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 kernel classes define the data structures used, and the generic functions and methods define the basic operations of the kernel, as specialized to those kernel classes. We liken this level of description to the description of Lisp or Scheme using a metacircular interpreter written in the language. To understand the behavior requires understanding the language. The metacircular interpreter thus provides a kind of intellectual bootstrap. 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. A protocol is a set of related generic functions for some purpose.

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 CLOS kernel makes 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 and extensions 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.

This chapter is organized as follows:

We start with a detailed description of the classes that are provided with the CLOS system. After describing the inheritance relationships of the classes, we describe the information contained in each class by describing how to create instances of these classes. We summarize from Chapter 1 how instances of standard-class are made using make-instance. We then specify several generic functions mentioned there but not described in chapter 2. The critical specification for creation is the set of arguments for the initialize-instance method for each class. We describe these arguments for each of the objects that support the standard facilities of CLOS. The specification consists primarily of the description of the arguments for methods of that generic function. We further specify a set of "reader" methods that read information from instances of the class stored by the initialize-instance method. This use of the term reader is a pun on the the use of that term as an automatically generated method for a class. The similarity is that each of these readers takes a class as its single argument and returns a value relevant for that class..

We then describe a general reinitialization protocol for CLOS. This reinitialization protocol uses initialize-instance to update instances, and also invokes a mechanism for updating other objects that are dependent on the state of the reinitialized instance. For each kernel class in CLOS, we specify when its instances can be reinitialized, and how objects dependent on state of these instances will be updated.

None of the operations above make any use of names for objects. The user interface macros (defclass, defmethod, defgeneric) do use names. We show how each of these macros expand, and describe the behavior of the functions they expand into.

We then describe the instance structure protocol. This consists of two levels. The slot level description involves the metaclass in the allocation of an instance, and the extraction of values for slots from the instance. The access-key level of description provides users a mechanism for specifying storage to be allocated in instances in addition to slots, and a mechanism to access to that information.

The protocol for building generic functions is then described. It includes the generic function for computing the effective method for a set of methods, and a generic function that can be specialized to compute the code used by the generic function to do its discrimination.

Method combination objects are used in the computation of effective methods. We include a description of the protocols used to create and manipulate these objects.

Finally there is a description of some of the generic functions related to make-instance but not previously described.

The Classes in the CLOS Kernel

The specification of the class structure of the CLOS kernel is based on a number of principles:

1. We specify only subclass relationships, and no direct subclasses. This allows implementations to insert implementation dependent classes into the lattice, for example, to organize code sharing.

2. We specify as disjoint the metaclasses that support built-in classes, structure classes and standard classes. These metaclasses are *built-in-class*, *structure-class*, and *standard-class* respectively. This allows use of a simple type test to determine the metaclass used by any class.

3. We provide classes in the kernel to support default behavior of instances of standard-class and structure-class. These are the classes *standard-object* and *structure-object* respectively.

4. In the kernel, we define classes *class*, *slot-description*, *method*, *generic-function*, and *method-combination*. These classes are intended to act as superclasses for any implementations of these types. They contain no structure or behavior themselves. These are distinguished from the classes that support the behavior of the user interface macros. The latter classes, subclasses of those above, are *standard-class*, *standard-slot-description*, *standard-method*, *standard-generic-function*, and *standard-method-combination*.

5. We do not specify slots in any class in the kernel, although implementations are free to use them. We define reader generic functions to obtain information from instances of a class. Other generic functions in the kernel are said to update this information, without specifying how the updating is done. This allows implementations to store information in compact forms not accessible by the user. It also allows an implementation to create and maintain caches.

In the specification, we specify some readers as *storage readers*. By this we mean they act as though they were reading slot values; that is, they return the same value each time, unless a specific updating action is done. We use the phrase *X is stored as a value for reader-Y* to talk about the only way of changing the value associated with a storage reader. For example, class-direct-slots is a reader for standard-class. We say *the list of slot-description objects is stored as the value for class-direct-slots.* This circumlocution is necessary because we want to limit places where updating is done, and we don't want to define setf functions that are accessible to users.

Lattice of Kernel Classes

The following should be a real diagram, but for now we use indentation to indicate subclass relationships. We use angle brackets to indicate those classes whose primary use is for type determination. All of the classes below are instances of standard-class except T, which is an instance of built-in-class, and standard-generic-function, which is an instance of funcallable-standard-class.

```
T
  <standard-structure>
  <standard-object>
    <generic-function>
        standard-generic-function
    <method>
        standard-method
        standard-accessor-method
            standard-reader-method
            standard-writer-method
    <method-combination>
        standard-method-combination
        simple-method-combination
    <slot-description>
        standard-slot-description
    <class>
        built-in-class
        structure-class
        standard-class
            funcallable-standard-class
```

One subclass relationship is not shown; *generic-function* is also a subclass of *function*. *function* is more specific than *standard-object* in the class precedence list of *generic-function*. *generic-function* is the one class in the kernel that has multiple superclasses. Implementation of this relationship awaits the change in Common Lisp that makes function be a real type.

Instances of class, built-in-class and structure-class

Instances of built-in-class are specified by the implementation, and have special constructors for their creation. They may include all the classes associated with Common Lisp types, though implementations are free to use standard-classes for any of them. If make-instance is called on an instance of built-in-class, an error is signalled.

CLOS does not specify the behavior of *structure-class* for make-instance.

Instances of standard-class

Instances of standard-class can be created using the make-instance protocol described in chapters 1 and 2. As shown in chapter 1, the standard method for make-instance acts as though it were defined as follows:

```
(defmethod make-instance ((class standard-class) &rest initargs)
  (setq initargs (default-initargs class initargs))
  (check-initargs class initargs)
  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
      instance))
```

At this point we will describe the generic functions allocate-instance, check-initargs and default-initargs. The generic function initialize-instance is described in chapter 2. We will then describe each of the classes in the kernel in terms of the arguments they expect for initialize-instance, and the readers that can extract information from instances of these classes.

allocate-instance

allocate-instance `(class standard-class) &rest initargs` [*Primary Method*]

Allocate-instance for standard-class creates a structure that has enough room to store information specified by the class. It calls allocate-standard-instance to achieve this. The underlying mechanism for specifying how much storage is allocated, and how it is accessed is described in the section on Instance Structure protocol.

Particular implementations are free to add keyword arguments for their own purposes (e.g. area allocation)

check-initargs

check-initargs `(class standard-class) initargs` [*Primary Method*]

The purpose of check-initargs is to determine if the keyword arguments provided to make-instance are acceptable to either initialize-instance or allocate-instance. It is equivalent to testing whether all the keyword arguments in initargs are members of the list returned by **(class-all-initargs class)**, but the latter need not be called by check-initargs each time.

default-initargs

default-initargs `(class standard-class) initarg-list` [*Primary Method*]

The system-supplied method for **default-initargs** implements the **:default-initargs** class option by appending initialization arguments that do not appear in the *initarg-list* argument to the *initarg-list* argument and returning the result.

The order of the initialization arguments appended to the list is determined by the rules for duplicate initialization arguments presented in the section "Rules for Duplicate Initialization Arguments."

The *initarg-list* argument is not modified.

The result is an initialization argument list. The results are undefined if the value returned by this function is modified.

Initializing standard-class

In building instances of standard-class, the only method that is specialized is initialize-instance. The purpose of this method is to check and normalize the input arguments, and to store appropriate information in the class.

initialize-instance *(class standard-class) &key :direct-superclasses :direct-slots :direct-options :documentation* [*:After Method*]

Arguments:

direct-superclasses

The direct-superclasses argument, if provided, must be a list of class objects. The value of direct-superclasses is defaulted by a call to

(class-default-direct-superclasses class direct-superclasses).

The value returned from this call should be a list of class objects to be used as a superclass for this class.

For each superclass in the returned-value, a call is made to

(valid-superclass-p class superclass)

to check if the metaclass of the provided superclass is compatible with the metaclass of class. If valid-superclass-p returns nil, an error is signalled. If there is no error for any class on the list, the value is stored as the value for class-direct-superclasses.

For each class object on that list, add-direct-subclass is called to maintain a backlink from super to subclasses. This is done before the value is stored.

If initialize-instance is not given a value for direct-superclasses, then no value is stored. If no value has ever been stored for class-direct-superclasses, then the class is considered not-initialized; that is, class-initialized-p will return nil for this class. Once a value has been stored for class-direct-superclasses, then class-initialized-p will return t.

direct-slots The direct-slots argument, if given, must be a list of slot-description objects.

For each element of this list,

(valid-slot-description-p class slot)

is used to check the compatibility of each slot-description-object.

If valid-slot-description-p returns nil, an error is signalled

This list is stored as the value of class-direct-slots. If initialize-instance is not given a value for direct-slots, and no value is currently stored for class-direct-slots, then nil is stored.

direct-class-options This is a list of class options. Each option must be a list. Each option in the list is checked using

(valid-class-option-p class option)

If valid-class-option-p returns nil, an error is signalled. If there are no errors, the argument direct-class-options is stored as the value of class-direct-class-options. If initialize-instance is not given a value for direct-options, and no value is currently stored for class-direct-options, then nil is stored.

If class-options contains a :metaclass option, it must pass the valid-class-option-p check. It is left in the list of options, but is ignored in further processing. Processing of :metaclass is handled by defclass expansion, or by using the specified class for make-instance.

:documentation This is a string used to document this standard-class. It will be accessible by the generic-function **documentation**. If no value is provided, and none has ever been provided, the empty string is stored.

class-default-direct-superclasses

This generic function is called to determine the direct-superclasses for a class. It is called by the method for initialize-instance for standard-class when direct-super-classes have been supplied. It receives as arguments the class for which the superclasses are intended, and the supplied superclasses. It returns a list of classes.

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

The purpose of this method is to ensure that the class named standard-object will be in the class precedence list of all standard-classes. If **supplied-superclasses** is **nil**, this method returns a list containing the class named standard-object. Otherwise it returns its argument **supplied-superclasses**.

class-default-direct-superclasses `(class funcallable-standard-class) supplied-superclasses` [*Primary Method*]

If **supplied-superclasses** is nil, then this method returns a list containing the class named function and the class named object. Otherwise it returns its argument **supplied-superclasses**.

Example of specializing class-default-direct-superclasses

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 a default super class.

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

;;; Implement the rule that where the null superclass list is provided,
;;; return a list containing the class named loops-object.

(defmethod class-default-direct-superclasses
           ((class loops-class) supplied-superclasses)
  (or supplied-superclasses
      (list (symbol-class 'loops-object))))
```

valid-superclass-p

The generic function valid-superclass-p tests whether the proposed superclass has a metaclass compatible with being a direct-superclass of the class being initialized. It returns t if the proposed superclass is compatible, and nil otherwise.

The default method signals an error unless the metaclasses are **EQ**.

valid-class-option-p

This generic function is used to check the validity of class options. It uses **or** method combination type, and returns true iff one of the applicable methods believes that the option is legal.

valid-class-option-p `(class standard-class) option` [*Primary Method*]

This method checks for the allowed options described in chapter 1.

valid-slot-description-p

This generic function is used to check the compatibility of a class and slot-objects supplied in the call to make-instance.

valid-slot-description-p `(class standard-class) (slot-object standard-slot-description` [*Primary Method*]

This method returns t.

valid-slot-description-p `(class standard-class) slot-object` [*Primary Method*]

This method returns nil. This ensures that random objects cannot be used as a slot-description, and a reasonable error message will be given.

valid-slot-option-p

The purpose of this generic function is to allow checking of slot description options without having to construct the slot descriptions themselves. The argument to this method is the class-prototype of the class of slot-description object that is to be constructed for this class. Since all

of the options are to be iniitialization arguments for initialize-instance, the effective code for the standard method can be:

```
(defmethod valid-slot-option-p ((slotd standard-slot-description) option-keyword)
  (member option-keyword
          (method-applicable-keywords
            #'initialize-instance
            (list  slotd))))
```

Readers for Standard-class

class-direct-superclasses

The value of this reader is a list of the classes that are the direct superclasses of the given class. This is the value stored by initialize-instance.

Remarks: This is a storage reader. The results are undefined if the value returned by this function is modified.

If a class has not had its direct-superclasses initialized, an error is signalled by **class-direct-superclasses**. The generic function class-initialized-p may be used to test if this error will be signalled.

class-direct-slots

The value of this reader is a list of slot-description objects defined directly for the given standard-class. This is the value stored by initialize-instance. The results are undefined if the value returned by this function is modified.

Remarks: This is a storage reader.

class-direct-options

The value of this reader is the list of class options as specified in Chapter 1. This is the value stored by initialize-instance.

Remarks: This is a storage reader. The results are undefined if the value returned by this function is modified.

class-precedence-list

Value: An ordered list of class objects. The first element of the list is the class itself. For each class C on that list, and for no others,

(subtypep class C)

is true. See chapter 1 for a full description of the class-precedence-list.

Remarks: If class-finalized-p is nil for a standard-class, an error is signalled. Otherwise the value returned is the one stored by update-dependent. The results are undefined if the value returned by this function is modified.

This generic function is applicable to all the subclasses of **class**, including **built-in-class** and **structure-class**.

class-finalized-p

The value of this reader is t if the class has been finalized by finalize-inheritance, and nil otherwise.

Remarks: class-finalized-p returns nil until its value is changed by finalize-inheritance. Once that value becomes t, it will always return that value.

class-initialized-p

The value of this reader is t if initialize-instance has ever been called with a value for:direct-superclasses, and is nil otherwise. This value is set to t by initialize-instance the first time a value has been stored for class-direct-superclasses.

Remarks: Class-direct-superclasses will signal an error iff class-initialized-p returns nil.

class-prototype

The value of this reader is an instance of the class. The results are undefined if the value returned by this function is modified.

Remarks: This instance may be a completely blank instance as created by allocate-instance. The same value may be returned on each call. The only thing guaranteed about this instance is that it will respond properly to class-of and will select a method specialized on this class or any subclass.

It is not specified when this instance is created; it can be as late as the first call to class-prototype for this class. The slots of this instance may all be unbound. People who write code for print-object should take this into account.

class-slots

The value of this reader is a list of slot-description objects. An error is signaled if class-finalized-p is nil. These slot-description objects are used to determine the structure of instances of this class. The results are undefined if the value returned by this function is modified. It is the value stored by update-dependent.

class-direct-slot-initargs

The value of this reader is a list of the form:

((initialization-arg-name slot-name slot-name)...)

This value is extracted from the value stored for class-direct-slots. The result is undefined if the value returned by this function is modified.

Remarks: The value of this reader specifies what initialize-instance argument names can be used to initialize slots defined in this class, and what slots they initialize.

class-slot-initargs

The value of this reader is a list of the form:

((initialization-arg-name slot-name slot-name)...).

Remarks: The value of this reader specifies what initialize-instance argument names can be used to initialize slots defined in this class or in any of its superclasses.

The value of this reader is dependent on the values of class-direct-slot-initargs for all classes in its class-precedence-list.

The result is undefined if the value returned by this function is modified.

class-direct-initargs

The generic function **class-direct-initargs** returns a list of the names of all initialization arguments corresponding to a given class. These initialization arguments include those that were specified using the **:initarg** slot option in the **defclass form** for the given class, as well as those that were defined in the lambda-lists for methods on **allocate-instance** and **initialize-instance** that are defined for the exact given class.

The results are undefined if the value returned by this function is modified.

It is permitted, but not required, for an implementation to return values that share with internal data structures.

class-initargs

This generic function returns a list of the names of all initargs that apply to a given class, including the names of inherited initargs. This result is the union of the following sets of the initialization argument names: the initialization argument names contained in the result of `(class-slot-initargs` *class*`)` and the initialization argument names that are defined by all the applicable methods for **allocate-instance** and **initialize-instance**.

Remarks: The results are undefined if the value returned by this function is modified.

class-direct-initarg-defaults

The generic function **class-direct-initarg-defaults** returns a list, each of whose elements is a list

consisting of the name of an initialization argument defined for a slot in that exact class followed by the default value function and the default initial value form for that initialization argument.

The default value function is the function whose body is the default initial value form; this function is created in the lexical environment of the **defclass** form that contains the default initial value form. The default initial value form is the form that was originally specified. It is retained only for documentation. The results are undefined if the default initial value form is evaluated. The default value function is what actually gets called; its effect is equivalent to enclosing the default initial value form in the appropriate lexical environment. The default value function takes no arguments.

This value is stored by initialize-instance whenever a value is provided for class-direct-options. The results are undefined if the value returned by this function is modified.

It is permitted, but not required, for an implementation to return values that share with internal data structures.

class-initarg-defaults

The generic function **class-initarg-defaults** returns a list, each of whose elements is a list consisting of the name of an initialization argument defined for a slot in that class or any of its superclasses followed by the default value function and the default initial value form for that initialization argument. The default value function is the function whose body is the default initial value form; this function is created in the lexical environment of the **defclass** form that contains the default initial value form. The purpose of this function is to capture the environment.

The default initial value form is the form that was originally specified. It is retained only for documentation. The results are undefined if the default initial value form is evaluated.

The default value function is what actually gets called; its effect is equivalent to enclosing the default initial value form in the appropriate lexical environment. The default value function takes no arguments.

The results are undefined if the value returned by this function is modified.

It is permitted, but not required, for an implementation to return values that share with internal data structures.

class-direct-subclasses

The value returned by class-direct-subclasses is a list of all classes that have this class as a direct superclass.

The value of this reader is maintained by initialize-instance and reinititialize-instance for standard-class through calls to add-direct-subclass and and remove-direct-subclass.

CLOS does not specify whether this generic function has an applicable method for built-in-class

or structure-class.

class-direct-methods

The value of class-direct-methods is a list of methods (with no duplicates) that use this class as a specializer.

The value of this reader is maintained by add-method and remove-method for standard-method and for standard-accessor-method through calls to add-method-on-specializer and remove-method-on-specializer.

class-direct-generic-functions

The value of class-direct-generic-functions is a list of generic-functions (with no duplicates) that contain a method that use this class as a specializer.

The value of this reader is maintained by add-method and remove-method for standard-method and for standard-accessor-method through calls to add-method-on-specializer and remove-method-on-specializer.

Initializing standard-slot-description

The keywords allowed as arguments for initialize-instance of standard-slot-description are those that support the description of slots provided in chapter 1.

initialize-instance *(class standard-slot-description) &key :name :initform :initfunction :initargs :readers :writers :accessors :type :allocation* [*:After Method*]

Arguments:

:name The argument :name must be provided; otherwise an error is signalled. It must be a symbol that can be used as a slot-name. This value is stored as the value for the reader **slot-description-name**.

:initform :initform can be any Lisp form. This value is stored as the value for the reader **slot-description-initform**. If no value is provided, nil is stored.

:initfunction :initfunction must be a function of no arguments. The purpose of this function is to capture the lexical context of the defining defclass form. At slot initialization time, this function is called to get an initial value for a n unbound slot. This value is stored as the value for the reader **slot-description-initfunction**. If no value is provided, nil is stored.

:initargs This is a list of symbols that can be used for slot-filling initargs for this slot. This value is stored as the value for the reader **slot-description-initfunction**. If no value provided, nil is stored.

:accessors This is a list of symbols that are to be used as names of generic functions for readers for this slot of this class, and for which names of the form (setf symbol) will be used as writers of this class. The value of :accessors is used to suplement :readers and :writers.

:readers This is a list of symbols that are to be used as names of generic functions for readers for this slot of this class. The value, augmented by information from :accesssors, is stored as the value for the reader **slot-description-readers**. If no value is provided, nil is stored.

:writers This is a list of function specifiers that are to be used as names of generic functions for writers for this slot of this class. The value, augmented by information from :accesssors, is stored as the value of the reader **slot-description-writers**. If no value is provided, nil is stored.

:type This is a Common Lisp type expression. If no value provided, t is stored.

:allocation This is one of the symbols :instance or :class. The value for this argument is stored as the value for the reader **slot-description-allocation**. This value determines how allocation for this slot will be done. If no value is provided, :instance is stored.

Readers for standard-slot-description

slot-description-name

This reader returns the name of the slot stored by initialize instance.

slot-description-initform

The value of this reader is the Lisp form stored by initialize-instance as the value of the :initform option.

slot-description-initfunction

The value of this reader is the Lisp function stored by initialize-instance as the value of the :initfunction option.

slot-description-type

The value of this reader is a Common Lisp type specifier. The value is the one passed in to the call to initialize-instance.

slot-description-allocation

The value of this reader for standard-class is one of :instance or :class. It is the value stored by initialize-instance.

slot-description-initargs

The value of this reader is a list of symbols that can be used as a slot-filling initarg for this slot.

slot-description-readers

The value of this reader is the list stored by initialize-instance constructed from the :readers and :accessors options.

slot-description-writers

The value of this reader is a list stored by initialize-instance constructed from the :writers and :accessors options.

Initialization for Methods

initialize-instance *(class standard-method) &key :lambda-list :specializers :qualifiers :function* [*:After Method*]

The :lambda-list, :specializers and :function must be provided. The :qualifiers will default to nil.

:lambda-list The lambda-list argument is a CLtL lambda-list. It must specify the *natural* arguments to the function provided as the :function argument to initialize-instance. The behavior is undefined if this contract is not met. When this method is added to a generic function, the congruence of this lambda list will be checked against the lambda-list of the generic function.

:specializers This must be a list whose length is equal to the the number of required arguments in the lambda-list. Its elements must be parameter specializers as defined in Chapter 1.

:qualifiers This must be a list of non-nil atoms. See Chapter 1.

:function This argument must be a function that satisfies method-function-p. See the description of this in the next section. This function must take as arguments at least those specified by lambda-list. Again, see the discussion of method-lambda in the section on Method Functions.

:documentation This is a string used to document this method. It will be accessible by the generic-fucntion **documentation**. If no value is provided, the empty string is stored.

Method Functions

In order to capture the contract that a generic function and method must support between them, an implementation may need to be pass additional information into a method at run time: for example, the list of next methods for call-next-method, or a permutation vector for optimized access to slots of some class, etc. We support this abstraction by introducing three new constructs: method-lambda, method-function-p and apply-method-lambda.

method-lambda

The special form method-lambda is used for constructing functions in exactly the same way

lambda is used to construct ordinary functions, except that the functions it produces are designed to be called with apply-method-lambda. The number of arguments for the function it constructs may be different than the number explicit in the lambda-list of the form. The results of calling a method-lambda with ordinary apply are undefined.

The predicate method-function-p can be used to test if a function was created with method-lambda.

method-function-p

The predicate method-function-p returns true if its argument is a function created with method-lambda, and return nil otherwise.

apply-method-lambda

The function apply-method-lambda is like apply. It takes the 'natural' arguments that want to be passed to the method function. It does anything else that needs to be done to call the method-function properly. Any call to call-next-method inside of the called method-function would signal an error indicating there were no next methods.

Here is some sample code.

```
(defun trace-gf (gf)
  (let ((nargs (generic-function-required-arguments gf))
        (lambda-list (generic-function-lambda-list gf))
        (specializers (make-list nargs :initial-element (symbol-class 't)))
        (qualifiers '(:around))
        (function (compile () `(method-lambda ,lambda-list
                                 (format *trace-output* "~&Hi there.")
                                 (call-next-method)))))
    (add-method gf
                (make-instance 'standard-method
                               :lambda-list lambda-list
                               :specializers specializers
                               :qualifiers qualifiers
                               :function function))))
```

Readers for standard-method

For all of these readers, the discussion in chapters 1 and 2 should be consulted.

method-generic-function

The value of this reader is the generic function object of which this method is a part. This reader returns nil if this method is not on a generic function. The value returned by this reader is

maintained by add-method and remove-method.

method-lambda-list

The value of this reader is the lambda-list stored by initialize-instance. The results are undefined if the value returned by this function is modified.

method-specializers

This is a list of parameter specializers for this method as specified in Chapter 1. The value of this reader is the list of specializers stored by initialize-instance. The results are undefined if the value returned by this function is modified.

method-qualifiers

This is a list of qualifiers for this method. It is a list of non-nil atoms. The value of this reader is the :qualifiers argument stored by initialize-instance. The results are undefined if the value returned by this function is modified.

method-function

The value of this reader is the method function stored as the :function argument by initialize-instance.

Initializing standard-accessor-methods

Standard-accessor-methods are the automatically generated methods that support the access to slots of classes, as specified by the :reader and :writer slot options for defclass.

These are generated by the update-dependent method on standard-class. It cannot be generated by initialize-instance since a class may not yet be fully defined when initialize-instance is called. Everything said here is true for both instances of standard-reader-method and standard-writer-method. Users can can create instances of these methods using make-instance.

initialize-instance *(class standard-accessor-method) &key :applicable-class :slot-name* [*:After Method*]

:applicable-class This argument is a class-object. The specializer list for this method is simply (list :applicable-class).

:slot-name is the name of slot in applicable-class. An error is signalled if this is not true when the method is created..

Method objects that are instances of standard-reader-method (standard-writer-method) supports reading (writing) the value of this slot in instances of applicable-class.

The value stored for the reader method-lambda-list for these accessor methods is:

(list (or (class-name applicable-class) 'object))

for readers, and

(list 'new-value (or (class-name applicable-class) 'object))

for writers.

Readers for standard-accessor-method

method-applicable-class

This method returns the applicable-class stored by initialize-instance. This method will be applicable to instance of this class, and all its subclasses.

method-slot-name

This method returns the slot-name stored by initialize-instance.

method-generic-function

The value of this reader is the generic function object of which this method is a part. This reader returns nil if this method is not on a generic function. The value returned by this reader is maintained by add-method and remove-method.

method-lambda-list

The value of this reader is described above in the section on initialize-instance. The results are undefined if the value returned by this function is modified.

method-specializers

The value of this reader is a list of one element, the applicable-class. This value is stored by initialize-instance for standard-sccessor-method. The results are undefined if the value returned by this function is modified.

method-qualifiers

This method always returns nil for accessor methods.

Initializing Standard-generic-function

Generic function objects are both like standard-objects in the sense that they maintain state that can be accessed by readers, and funcallable objects in the Common Lisp sense.

Generic function objects are intrinsically anonymous. They have names in the same sense that ordinary function objects do; that is, they may appear in the symbol-function cell of some function specifier. In the creation and initialization of these objects, there is no way to specify a name. But see the reader generic-function-name below for a way to associate a hint about a name with these objects.

The initialize-instance method is what is called by ensure-generic-function to initialize an instance of the generic function. We mention this here only because the constraints on the arguments to initialize-instance and reinitialize-instance described below are just those that support the capabilities described for ensure-generic-function.

initialize-instance *(class standard-generic-function) &key :lambda-list :argument-precedence-order :declare :documentation :method-class :method-combination :methods* [*:After Method*]

:lambda-list This is a lambda-list of the form described for defgeneric.

:argument-precedence-order This is a list which is a permutation of the names of the required arguments in the lambda-list. If no value is provided it defaults to a list of the required arguments of the lambda-list, in the same order as in the lambda list.

:declare This is a form that specifies declarations applicable to the generic function as a whole. See the description in defgeneric. If no value is provided, it defaults to ().

:documentation This is a string describing the generic function.

:method-class This must be a class object that will be used to create instances of methods for this generic function that are defined using the method defining forms. If it is not provided it defaults to (symbol-class 'standard-method).

:method-combination This is a method combination object. If it is not provided it defaults to an instance of (symbol-class 'standard-method-combination). See the description below of expansion of the user interface macro forms.

:initial-methods This is a list of method objects. These are methods that are stored as the value of generic-function-initial-methods. In addition, any of these methods which is not on the generic function is added to the generic function. If the generic function is reinitialized, each of these methods will be removed from the generic function. See method-reinitialization.

Readers for Standard-generic-function

generic-function-name

The value returned by this reader is a function specifier or nil. This value is a hint for the environment. In general, one cannot count on the following identity:

(eql (symbol-function (generic-function-name g-fn)) g-fn)

There is a setf generic function associated with generic-function-name. This may be used to change the value returned by this reader.

generic-function-lambda-list

This is the lambda-list stored by initialize-instance. It is also the lambda list of the generic function considered as a functional object.

The results are undefined if the value returned by this reader is modified.

generic-function-argument-precedence-order

This is the list stored by initialize-instance as the value for :argument-precedence-order. It is a permutation of the list of required arguments of the generic-function-lambda-list.

The results are undefined if the value returned by this reader is modified.

generic-function-method-class

This is the class object stored by initialize-instance as the value for :method-class. The value of this reader will be used in the creation of method objects for this generic function defined using defclass.

generic-function-declarations

This value is the form stored by initialize-instance from the :declare argument.

The results are undefined if the value returned by this reader is modified.

generic-function-method-combination

This is the method-combination object stored by initialize-instance as the value for :method-combination. The value of this reader will be used in the creation of the effective code for method combination.

The results are undefined if the value returned by this reader is modified.

generic-function-methods

The results of this reader is a list of methods defined on this generic function. This list is updated by add-method and remove-method.

The results are undefined if the value returned by this reader is modified.

generic-function-initial-methods

The results of this reader is a list of methods defined on this generic function at initialization

or reinititialization. These methods are also elements of the list returned by generic-function-methods. This list of methods is updated to the most recent value for the :initial-methods argument of initialize-instance.

The results are undefined if the value returned by this reader is modified.

method-applicable-keywords

The purpose of this generic function is to deliver to the user a list of the keyword arguments that are legal for a particular set of arguments for a generic function.

method-applicable-keywords `(g-fn standard-generic-function) list-of-required-arguments` [*Primary Method*]

The function returns a list of keywords (possibly nil). If &allow-other-keys is a member of the list, then any keyword argument will be legal in the call.

add-method, remove-method, get-method

The generic functions add-method and remove-method have been described in chapter 2. They are the only acceptable mechanism for updating the set of methods on a generic function. The standard method on add-method calls add-method-on-specializer. The standard method on remove-method calls remove-method-on-specializer.

The generic function get-method can be used to locate a method that has a particular set of specializers on a generic function.

add-method-on-specializer

add-method-on-specializer `(method method)(class standard-class)` [*Primary Method*]

Whenever a method is added to a generic-function, this method is called on every class specializer for the method. It updates the values of class-direct-methods and class-direct-generic-functions for the class.

remove-method-on-specializer

Whenever a method is removed from a generic-function, this method is called on every class specializer for the method. It updates the values of class-direct-methods and class-direct-generic-functions for the class.

Initializing Method Combination Objects

There are two classes of method combination objects defined in the standard: standard-method-combination and simple-method-combination.

initialize-instance *(method-combination standard-method-combination) &rest initargs* [*:After Method*]

This method has no initialize-instance arguments, since it has no state that is special. What is missing is a description of the expected arguments for a new class defined by define-method-combination.

initialize-instance *(class simple-method-combination) &key :name :order :operator :identity-with-one-argument :documentation* [*:After Method*]

This method supports the creation and initialization of objects for the short form of method-combination as specified in Chapter 2.

:order This argument is one of the two values **:most-specific-first** or **:most-specific-last**. If it is not provided, it defaults to **:most-specific-first**.

:operator This argument is the name of an n-ary Lisp function.

:identity-with-one-argument This argument is a boolean which is t if the optimization described in chapter 2 is possible, else nil.

:documentation This argument is a string used to document this simple-method-combination. It will be stored in a way that will make it accessible by the generic function **documentation**.

Readers for Method combination objects

method-combination-name

This generic function returns the name used by method-combination-object to identify this method combination type. For standard-method-combination the value is **standard-method-combination**

See the description later of method-combination-object in the section on expanding the user macros.

method-combination-options

This generic function returns the options used by method-combination-object to identify this method combination type. For standard-method-combination the value is nil. See the description later of method-combination-object in the section on expanding the user macros.

method-combination-order

method-combination-order `(method-combination simple-method-combination)` [*Primary Method*]

Returns the value of the argument :order given to initialize-instance.

method-combination-operator

method-combination-operator `(method-combination simple-method-combination)` [*Primary Method*]

Returns the value of the argument :operator given to initialize-instance.

method-combination-identity-with-one-argument

method-combination-identity-with-one-argument `(class simple-method-combination)` [*Primary Method*]

Returns the value of the argument :identity-with-one-argument given to initialize-instance.

Reinitialization and Updating Dependents

CLOS supports a general notion of reinitialization for standard-objects, and the recording and updating of dependent objects when an object is changed. The generic-function **reinitialize-instance** invokes **initialize-instance** to make changes to the state of an object, and then calls **update-dependents** to update any objects that have registered themselves as dependent on the state of the changed object. The generic function **add-dependent** is used to register the dependence of one object on the state of another, and **remove-dependent** is used unregister the dependence. The generic function **update-dependent** does the updating.

reinitialize-instance for standard-object

Following is some model code for the method on standard-object.

```
(defmethod reinitialize-instance :before
   ((object standard-object) &rest reinit-args)
  ;; signals an error on bad arguments
  (check-reinitargs object reinit-args))

(defmethod reinitialize-instance
   ((object standard-object) &rest reinit-args)
  ;; reuse initialization methods
  (apply #'initialize-instance
 object
 :allow-other-keys 't
 reinit-args))

(defmethod reinitialize-instance :after
   ((object standard-object) &rest reinit-args)
  ;; updates dependents
  (apply #'update-dependents object reinit-args))
```

Note that reinitialize-instance does not use default-initargs. Nor does it automatically make any of the slots of object be unbound. Hence in initialize-instance, only those arguments explicitly given will affect the object. Values for slot-filling initargs will be stored. Initforms will be evaluated and stored only for slots that are unbound. Programs can call slot-makunbound for particular slots, to cause those slots to be reinitalized from the intiforms.

check-reinitargs

This ensures that the arguments given are acceptable to either initialize-instance or to a method of reinitialize-instance itself.

```
(defmethod check-reinitargs
   ((object standard-object) reinit-args)
  (let* ((arglist (list object))
 (acceptable-args
           (union
             (method-keyword-arguments #'initialize-instance arglist)
             (method-keyword-arguments #'reinitialize-instance arglist)
             (class-all-slot-initargs (class-of object)))))
    (do* ((tail reinit-args (cddr tail))
  (key (car tail) (car tail)))
 ((null tail))
      (unless (member key acceptable-args :test #'eq)
(error ...)))))
```

update-dependents

There is a set of generic functions for managing the dependents of an object. This is part of the instance re-initialization protocol. How dependents are stored is implementation dependent. Some subclasses of standard-object which know that they are in fact going to have lots of dependents may want to replace the general methods with specialized mechanisms for storing the dependents (e.g. using a slot)

The set of generic functions include add-dependent, remove-dependent, map-dependents, for managing the data base, and update-dependents and update-dependent for doing the update.

The method on update-dependents specialized to standard-object maps through all the dependents of the object, calling update-dependent. Although the standard method does guarantee to do the mapping, for special cases update-dependents may have its own way of updating the dependents without mapping through a list of registered dependents.

The style recommended for use of this facility is not to make update-dependent be recursive, and have all dependnts, whether direct or indirect, be added to the set of dependents.

```
(defmethod update-dependents ((object standard-object) &rest update-args)
  (map-dependents
    object
    #'(lambda (changed-object dependent update-args)
(apply #'update-dependent changed-object dependent update-args))
    update-args))
```

map-dependents

map-dependents *object map-fn other-args* [*Generic Function*]

This generic function expects the object, a function and a list of other-args. It finds each regis-

tered dependent and does:

```
(apply #'map-fn object dependent other-args)
```

add-dependent

This generic function adds a dependent to the set of dependents. If the dependent is already a member, it is not added again.

In some cases, at the time of adding a dependent, the user may want to store additional information about the form of the dependence. Rather than providing an additional argument to add-dpendent, the user is expected to define a special object that captures the dependent, and the additional information. This is in the spirit of representing compound structures as objects rather than as lists. It also allows the type of the dependent object to be used to select the method.

remove-dependent

This generic function removes a dependent from the set of dependents. If the dependent is not a member, it is a noop.

update-dependent

update-dependent *object &rest update-args* [*Generic Function*]

This generic function is meant to do whatever is necessary to update dependent found by map-dependent. There is no method for update-dependent on standard-object so that calls to update-dependent that have no applicable method will signal an error.

Update-dependent methods are not supposed to recursively invoke update-dependent again. The intention is that all dependents should be recorded with add-dependent.

reinitialize-instance for standard-class

The reinitialize-instance method for standard-class is used to modify an existing class definition. One time that this happens is when add-named-class is called and there is already a class by that name. For example, this happens when a defclass form is reevaluated. All subclasses need to be informed when a class has changed must register themselves as dependents using add-dependent. A specialized method on update-dependent is used to propagate all necessary changes to the subclass.

The only specialized method for **reinitialize-instance** for standard-class is a primary method that helps in updating of the direct-subclasses. The keywords accepted by this method are just those of initialize-instance for standard-class.

```
(defmethod reinitialize-instance ((class standard-class) &rest reinit-args)
  (let ((old-direct-super-classes (class-direct-superclasses class)))
    (call-next-method)
```

```
    (let ((new-direct-super-classes (class-direct-superclasses class)))
      (dolist (c (set-difference old-direct-super-classes
 new-direct-super-classes))
;; remove obsolete direct subclass links
;; initialize-instance adds the links for new subclasses
(remove-direct-subclass c class)))))
```

This method maintains the links to subclasses.

add-direct-subclass

add-direct-subclass `(class standard-class) (subclass standard-class)` [*Primary Method*]

This generic function adds subclass to the value of class-direct-subclasses for superclass. This generic function is called by **initialize-instance** for standard-class. It is a noop if subclass is already a member of class-direct-subclasses.

remove-direct-subclass

remove-direct-subclass `(class standard-class) (subclass standard-class)` [*Primary Method*]

This generic function removes the subclass from the value of class-direct-subclasses for superclass. This generic function is called by **reinitialize-instance** for standard-class. It is a noop if subclass is not a member of class-direct-subclasses.

update-dependent for standard-class

The purpose of this method is to collect inherited information in the class. This includes the class-slots and class-precedence-list. When changes occur in the class lattice at or above any dependent class, **update-dependent** causes this collected information to be updated. **update-dependent** is also called by finalize-inheritance to ensure that the class contains the information necessary for instance creation.

The system-supplied method may conspire with methods for **make-instance**, **default-initargs**, **check-initargs**, **allocate-instance**, and **initialize-instance** to speed up object creation by precomputing and caching additional information. This optimization is implementation dependent, but the **update-dependent** mechanism that makes such optimizations possible is standardized.

Users with special optimization needs can write methods for **update-dependent** to precompute information based on inherited information and to update the precomputed information whenever changes occur.

The update-dependent code has an :around method to take care of class-access-keys and obsoleting instances if necessary.

```
(defmethod update-dependent :around ((reinitialized-class standard-class)
```

```
                                    (dependent standard-class)
                                    &rest other-args)
  ;; Implementation specific code to account for the fact that
  ;; the access keys for this class may change. This is required to
  ;; cause the optimization of standard-instance-access to
  ;; continue to work.
  ;; This may call make-instances-obsolete.
  )
```

The main method is used to collect information in the class from classes it inherits from. It computes the class-precedence-list by calling the generic function compute-class-precedence-list. This value is stored as the value for the reader class-precedence-list.

This method registers *dependent* as a dependent of every class on its class-precedence-list.

It computes the list of all slots by calling collect-slot-descriptions. This value is stored as the value for the reader class-slots.

When reinitialized-class = dependent, it goes through class-direct-slots and creates readers and writers as specified.

It collects values of default-initargs for dependent.

It may do other implemenation dependent actions.

```
(defmethod update-dependent ((reinitialized-class standard-class)
                             (dependent standard-class)
                             &key direct-superclasses
                                  direct-slots
                                  direct-options)
  ;; When we are called,there are two cases.  In one case
  ;; this method has been called from finalize-inheritance
  ;; In this case, the old cpl, slots, and generated
  ;; readers and writers are nil. We can check this case by
  ;; using the predicate class-finalized-p.
  ;; In the general case, the dependent is finalized.  We
  ;; have to do whatever updating is required to keep us finalized.
  ;; This includes:
  ;;  - recomputing our class-precedence-list
  ;;    go through the old and new cpls, calling remove-dependent
  ;;    and add-dependent to update our dependents.
  ;;  - recomputing our slots, this may change the value that
  ;;    class-access-keys will be returning.
  ;;  - go through the old and new slots adding and removing
  ;;    automatically generated reader and writer methods as
  ;;    required.
  ;;  - update the effect of class-options
```

)

Note that as indicated in the style note above, this update-dependent method for standard-class is not recursive. Indirect dependency of classes is not supported. All subclasses that depend on a class must register themselves directly as dependents.

finalize-inheritance

finalize-inheritance `(class standard-class)` [*Primary Method*]

The purpose of this generic function is to ensure that all inherited information is now collected in *class*. This generic-function is expected to be called only once. For standard-class, it is a noop to do it a second time, since update-dependent is expected to maintain the collected information once it has been stored. All the work of for standard-class is done by calling

(update-dependent class class)

reinitialize-instance for slot-description

reinitialize-instance `(class standard-slot-description)` [*Primary Method*]

Slot description objects are intended to be immutable after creation. An error is signalled if an attempt is made to reinitialize a slot-description object.

reinitialize-instance for methods

reinitialize-instance `(meth standard-method)` [*Primary Method*]

An error is signalled if an attempt is made to reinitialize a method while it is on a generic function. Otherwise, any of the arguments to initialize-instance will override the currrent value in the method. It is undefined what happens if the :function and the :lambda-list are changed in incompatible ways.

reinitialize-instance `(meth standard-accessor-method)` [*Primary Method*]

An error is signalled if an attempt is made to reinitialize a method while it is on a generic function. Otherwise, any of the arguments to initialize-instance will override the currrent value in the method.

reinitialize-instance for method-combination objects

reinitialize-instance `(meth method-combination)` [*Primary Method*]

Method combination objects are intended to be immutable after creation. An error is signalled if an attempt is made to reinitialize a method-combination object.

reinitialize-instance for standard-generic-function

The reinitialize-instance behavior for generic-functions supports the specified behavior of ensure-generic function.

reinitialize-instance `(g-fn standard-generic-function) &key :lambda-list :argument-precedence-order :declare :documentation :method-class :method-combination :initial-methods` [*Primary Method*]

This method then removes the set of initial-methods from the generic function. Then, if there are any methods on the generic function, it checks to see if the newly provided lambda-list is congruent with the previous lambda-list. If not, it signals an error. It then does (call-next-method).

:lambda-list If the lambda-list is not congruent to the current lambda-list, and the generic function has methods, then an error is signalled.

Other arguments are checked by initialize-instance.

Expansions of the User Macros

Expanding defclass forms

As with all objects in CLOS, there is a complete separation of the object space from any naming that is done. As explained in chapters 1 and 2, symbol-class is used to map from names to classes.

We do not specify what the function is that expands the defclass macro. It must do any system dependent and programming environment specific operations such as recording the locations of definitions on files etc. It is also allowed (encouraged) to use valid-slot-option-p and valid-class-option-p to check for errors to give early and better error messages than would occur at the time initialization takes place.

The defclass macro expands to a call to add-named-class. The first argument to add-named-class is a prototype-instance of the class specified by the :metaclass option in the defclass form. This allows class specific methods to be used for the actual definition of the class before having in hand the exact object that is the class. The other arguments for add-named class are just the obvious pieces of the defclass form.

add-named-class

This generic function is the programmatic interface for defining named classes. The first argument should be the class-prototype of the metaclass of the class being defined.

add-named-class `(class standard-class) &key :name :superclass-names :slot-specifications :direct-options :environment` [*Primary Method*]

The :super-names argument can be a list of either symbols (class names) or class objects. The slot-specifications argument should be a list of slot specifications as they would appear in a defclass form. The direct-options should be a list of the class options as they would appear in a defclass form. The environment is the macroexpansion environment of the defclass form. This is the &environment argument passed to the function that expands defclass.

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.

The standard method on add-named-class implements the behavior described for defclass in chapter 1.

If :super-names is provided, then it is normalized to a list of class objects as follows.

```
(mapcar #'(lambda(s)
    (cond ((typep s 'class) s)
```

```
  ((or (null s) (not (symbolp s)))
   (error ...))
  ((cboundp s)
   (symbol-class s))
  ;; create a standard-class named s with no
  ;; direct-superclasses this supports the
  ;; forward-referencing of standard-class this
  ;; uninitialized class can be reinitialized
  ;; but will cause an error if any of its
  ;; subclasses has its inheritance finalized
  (t (add-named-class
       (class-prototype (symbol-class 'standard-class))
       :name s))))
direct-superclasses)
```

This normalized list is used as the value for direct-superclasses.

The slot-specifications are parsed to turn them into slot description objects. The class-prototype is used to determine the class of the slot-description objects. For details, see the slot parsing protocol.

The next step performed by this method is to determine the class object which will be used for the new definition. If there is already a class with the given name, this method calls the generic function class-for-redefinition to get the class object to use.

If there was no existing class with the given name, make-instance is called using the metaclass specified, passing in the computed list of direct-superclass objects, the the computed set of slot-description objects, and the options as specified. It makes the resulting class have as a proper name the value of :name.

If class-for-redefinition was called, the resulting class will be reinititialized using reinitialize-instance using the same arguments specified above.

Some model code for this function is:

```
(defmethod add-named-class ((prototype standard-class)
                            &rest keys
                            &key name
                                 superclass-names
                                 slot-specifications
                                 direct-options
                                 environment)
  (let* ((direct-superclasses (mapcar ..))
         (slot-descriptions (mapcar ..))
         (new-class ()))
    (remf keys :name)
```

```
(remf keys :superclass-names)
(remf keys :slot-specifications)
(if (cboundp name)
    (progn
      (setq new-class
            (apply #'class-for-redefinition
                   (symbol-class name)
                   prototype
                   :direct-superclasses direct-supers
                   :slot-descriptions slot-specifications
                   keys))
      (apply #'reinitialize-instance
             new-class
             :direct-supers direct-supers
             :slot-specifications slot-specifications
              keys))
    (progn
      (setq new-class
          (apply #'make-instance
                 prototype
                 :direct-supers direct-supers
                 :slot-specifications slot-specifications
                 keys))
      (setf (class-name new-class name))
      (setf (symbol-class name environment) new-class)))
new-class))
```

class-for-redefinition

class-for-redefinition *prototype-instance old-class &key :direct-superclasses :direct-slots :direct-options :environment* [*Generic Function*]

class-for-redefinition is called by the standard method on add-named-class when there is already a class with the given name. The class-for-redefinition generic function is expected to return the class object which should be used for the new definition. For standard-class, the class object returned is the old class object since standard class supports the notion of updating old instances to reflect new definitions of the class. Other metaclasses might not support this notion; they might want new class definitions to use a new class object, or perhaps to signal an error if an attempt is made to redefine a class.

class-for-redefinition `(prototype-instance standard-class) (old-class standard-class)` [*Primary Method*]

This method returns the old-class argument.

Example 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)
            &rest other-args)
  (change-class existing-class
                (class-prototype (symbol-class 'optimized-class)))
  existing-class)

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

Expansion of defgeneric

The defgeneric form is expanded into a call to ensure-generic-function, and a defmethod method for each :method option that appears in the defgeneric form. The :methods argument for reinitialize-instance is a list of the method objects created by these defmethods.

The method-combination argument to ensure-generic-function is computed by the generic function **method-combination-object** applied to the method-combination option of defclass.

Following is one model of the expansion of defgenric. *We have to check that this really does what we want. It is probably still wrong*

```
(let ((g-fn
(ensure-generic-function <name>
```

```
 :method-combination (method-combination-object <meth-comb-arg>)
 ;; include other arguments directly from defgeneric
 ...)))
 (reinitialize-instance g-fn
   :initial-methods
   (list (make-instance (generic-function-method-class g-fn)
:lambda-list ...
:function #'(method-lambda ..)
..)
 ..)))
```

method-combination-object

The purpose of **method-combination-object** is to convert a method combination name and a list of options into an object.

method-combination-object *method-combination-name method-combination-options* [*Generic Function*]

It returns as a value a method-combination object.

method-combination-object signals an error if method-combination-name is unrecognized. Each method for **method-combination-object** signals an error if the method-combination-options are unrecognized or there are too many or too few of them.

method-combination-object might return the same object each time it is called with given arguments, or it might make a new object.

method-combination-object `(name (eql nil)) options` [*Primary Method*]

This method returns an instance of standard-method-combination. This method is used to compute the default method combination object when no argument is given to defgeneric.

method-combination-object `(name (eql 'standard-method-combination)) options` [*Primary Method*]

This method also returns an instance of standard-method-combination.

Expansion of defmethod

A defmethod form expands into

(add-named-method function-specifier method-qualifiers method-lambda-list method-specializers method-lambda-form)

For example,

```
(defmethod (setf foo) :before ((x c1) (y (eql *foo*)) z &optional (r 1))
  (do-foo ...))

would expand into

(add-named-method
  '(setf foo)
  '(:before)
  '(x y z &optional (r 1))
  (list (symbol-class 'c1 environment) (list 'eql *foo*) (symbol-class t))
  #'(method-lambda (x y &optional (r 1)) (do-foo ...)))
```

The method-lambda-list and the specializers must be computed byt the macro expansion function in order for the forms in the eql specializers to be evaluated in the correct lexical environment. The expansion uses two auxilliary functions, **extract-lambda-list** and **extract-specializers** to extract from the specialized lambda-list for the method the pieces needed for constructing a method. These auxiliary functions may be useful for users and are described below.

add-named-method

The purpose of **add-named-method** is to add a method to the named generic function.

Some model code for add-named-method function is:

```
(defun add-named-method
      (name qualifiers lambda-list specializers function)
 (let* ((gfn (ensure-generic-function name))
        (method (make-instance (generic-function-method-class g-fn)
                  :lambda-list lambda-list
                  :specializers specializers
                  :qualifiers qualifiers
                  :method-function function)))
   (add-method gfn method)
   method))
```

extract-lambda-list

extract-lambda-list *specialized-lambda-list &optional generic-function-lambda-list-p* [*Function*]

The purpose of this function is to construct a lambda list from a specialized lambda-list of the type specified for methods (See chapter 1). The optional argument *generic-function-lambda-list-p* specifies whether th resulting lambda-list is to be used in the definition of a generic function or a method. For a method, the resulting lambda-list is identical to the input lambda-list except that

specializers on the required arguments are stripped off. If *generic-function-lambda-list-p* is t, then the default values for optional arguments are also stripped off, and if there is an &key or &rest, the resulting lambda-list has an &rest argument.

Thus we have:

```
(extract-lambda-list
  '((x x-class)(y (eql 1)) &optional (z 2) &key w)
  nil)

==> (x y &optional (z 2) &key w)
-------------

(extract-lambda-list
  '((x x-class)(y (eql 1)) &optional (z 2) &key w)
  t)

==> (x y &optional z &rest rest-args)
```

These specifications are designed to conform to the minimal specifications of lambda-list congruence given in Chapter 1.

extract-specializers

extract-specializers *specialized-lambda-list* [*Function*]

The purpose of this function is to construct from a specialized lambda-list a form that will evaluate to the list of specializers for this method. This list has the same length as the number of required arguments for the specialized-lambda-list. *Question: How does the environment get passed in here?*

Expansion of define-method-combination

define-method-combination expands into a defmethod for method-combination-object. **remove-method** can be used to undefine a method combination type.

All invocations of the short form of define-method-combination define methods which return instances of the class simple-method-combination.

Each invocation of the long form of define-method-combination defines a new class which has a superclass in common with standard-method-combination. CLOS does not specify how many of these classes there are nor what their names are.

Example:

The short form of define-method-combination could have been defined
as follows:

```
(defclass short-form-method-combination (method-combination)
    ((name :initarg name :reader method-combination-name)
     (order :initarg order)
     (documentation :initarg documentation :reader documentation)
     (operator :initarg operator)
     (identity-with-one-argument :initarg identity-with-one-argument)))

(defmethod method-combination-options ((mc short-form-method-combination))
  (list (slot-value mc 'order)))

(defmethod compute-effective-method (generic-function
                                     (mc short-form-method-combination)
                                     methods)
  (let ((primary-methods (remove (list (slot-value mc 'name))
                                 methods :key #'method-qualifiers
                                 :test-not #'equal))
        (around-methods (remove '(:around)
                                methods :key #'method-qualifiers
                                :test-not #'equal)))
  (when (eq (slot-value mc 'order) ':most-specific-last)
    (setq primary-methods (reverse primary-methods)))
  (dolist (method (set-difference methods
                                  (union primary-methods around-methods)))
    (invalid-method-error method "The qualifiers of ~S, ~:S, are not ~S or ~S"
                          method (method-qualifiers method)
                          (list (slot-value mc 'name)) '(:around)))
  ;;--- Note: this example has not been updated to reflect the removal
  ;;--- of make-method-call.  If we do that, the update should be
  ;;--- straightforward.
  (make-method-call `(,@around-methods
                      ,(make-method-call primary-methods
                                         :operator (slot-value mc 'operator)
                                         :identity-with-one-argument
                                           (slot-value mc 'identity-with-one-
argument)))
                    :operator :call-next-method)))

(defmethod describe-method-concisely
           (generic-function
            method
            (method-combination short-form-method-combination))
  (declare (ignore generic-function))
```

```
  (write-string (string-downcase (string (first (method-qualifiers method))))))

(defmacro define-method-combination
          (name &key (documentation nil)
                     (operator name)
                     (identity-with-one-argument nil))
  `(defmethod method-combination-object
             ((name (eql ',name))
              options)
     (apply #'(lambda (&optional (order ':most-specific-first))
                (check-type order (member :most-specific-first
                                          :most-specific-last))
                (make-instance 'short-form-method-combination
                               'name ',name
                               'order order
                               'documentation ',documentation
                               'operator ',operator
                               'identity-with-one-argument
                                 ',identity-with-one-argument))
            options)))
```

Example of Defining a Method Combination Type via Inheritance

```
;This example defines a method combination type that is similar
;to standard method combination, except that it also allows :or
;methods.  The :or methods are executed after the :before methods,
;before the :after methods, inside the :around methods, and before
;the primary method.  The primary method is only called if all the
;:or methods return nil; if any :or method returns non-nil, its
;value becomes the value of the generic function (or the value
;returned by call-next-method in the least specific :around method)
;in place of the values of the most specific primary method.

;This assumes approach 2 or 3 to making effective method code
;analyzable, and assumes one particular code analysis tool, whose
;details I will not try to explain here.
;Those assumptions are not critical.

;I'm assuming we don't want to try to extend the define-method-combination
;macro so that it could exploit inheritance.  Instead I will
;define the example directly in terms of the next layer down.

(defclass standard-method-combination-with-or
          (standard-method-combination)
```

```
          ())

(defmethod method-combination-object
           (generic-function
            (name (eql 'standard-with-or))
            options)
  (declare (ignore generic-function))
  (unless (null options)
    (error "standard-with-or method combination does not accept options"))
  (make-instance 'standard-method-combination-with-or))

;This uses call-next-method to get the effective method in the absence
;of any :or methods, then it modifies the effective method form to
;incorporate the :or methods in an OR special form wrapped around the
;call to the most specific primary method.
(defmethod compute-effective-method (generic-function
                                     (mc standard-method-combination-with-or)
                                     methods)
  (let ((or-methods (remove '(:or) methods :key #'method-qualifiers
                            :test-not #'equal))
        (other-methods (remove '(:or) methods :key #'method-qualifiers
                               :test #'equal)))
    (lt:copyforms #'(lambda (subform kind usage)
                      (declare (ignore usage))
                      (if (and (listp kind) (listp subform)
                               (eq (first subform) 'method-call)
                               (null (method-qualifiers (second subform))))
                          ;; Put or methods before primary method
                          (values `(or ,@(mapcar #'(lambda (method)
                                                     `(method-call ,method))
                                                 or-methods)
                                       ,subform)
                                  t)
                          ;; Leave all other subforms of effective method alone
                          subform))
                  (call-next-method generic-function other-methods mc))))
```

The Slot Parsing Protocol

In converting a slot specification of the kind specified for defclass into a slot description object, the parsing goes through three steps. The first normalizes the specification into a keyword list of the kind acceptable to initialize-instance. It converts a slot specification that is just the symbol *x* to *(:name x)*, and pushes :name on the front of all other slot specifications.

In the second step, the class of the slot-description object is determined by a call to the generic function **slot-description-class**.

Finally, an instance of that class is created using make-instance, and the normalized slot-specification as initialization arguments.

```
(apply #'make-instance
       (slot-description-class
         (class-prototype specified-metaclass)
          normalized-slot-specification)
       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. However, the macro that parses the defclass form is free to use valid-slot-option-p to give earlier and better error messages.

slot-description-class

slot-description-class *containing-class normalized-slot-specification* [*Generic Function*]

The purpose of the generic function **slot-description-class** is to return the class to be used in the construction of a slot-description object for a *containing-class*. The reason normalized-slot-specification is passed to this generic function is to allow specializations that can use the specification itself to determine the class.

slot-description-class `(class standard-class) normalized-slot-specification` [*Primary Method*]

returns (symbol-class 'standard-slot-description)

slot-description-class

Computing Inherited Information

compute-class-precedence-list

This generic function computes the class precedence list of a class as described in Chapter 1. The value is a list of class objects in order.

compute-class-precedence-list `(class standard-class)` [*Primary Method*]

If in the computation of the class-precedence-list, the algorithm runs across a standard-class for which

(class-initialized-p class)=nil

an error is signalled.

The Slot Inheritance Protocol

The value of class-slots for any standard-class is computed by combining the class-direct-slots for the class and all of its superclasses. This combination is specified by the rules described in chapter 1.

The computation of the set of slots and their descriptions are controlled at two levels. The entire list is computed by **collect-slot-descriptions**. It uses the subfunction **compute-effective-slot-description** to combine sets of slots-descriptions that have the same slot-name.

collect-slot-descriptions

The generic function **collect-slot-descriptions** collects an ordered list of effective slot descriptions for this class. It uses the class-direct-slots from each of the classes on (class-precedence-list class).

collect-slot-descriptions `(class standard-class)` [*Primary Method*]

It sorts all the slots found into lists of slot-descriptions that have the same name. Each list is a mapping from the class-precedence-list, and has either a slot-description object or nil if none was defined at that class in the class-precedence-list. It calls compute-effective-slot-description with this list to create the new effective slot-description object.

For example, we mgiht have:

```
(defclass foo ()
    ((a :initform 1)
     (b :initform 2)))

(defclass bar (foo)
    ((c :initform 3)))

(defclass baz (bar)
    ((a :initform 4)))

There will be the following arguments to calls to
compute-effective-slot-description:

(#<slotd a :initform 4> NIL #<slotd a :initform 1>)
(#<slotd c :initform 3> NIL NIL)
(NIL #<slotd b :initform 2> NIL)

But they might not be in that order.
```

compute-effective-slot-description

compute-effective-slot-description `(class standard-class) slot-description-list` [*Primary Method*]

The standard method for this generic function supports the inheritance of slot options that is described in Chapter 1. It returns a new slot-description object with a description computed from all the slot-descriptions in slot-description-list.

Example of specializing compute-effective-slot-description

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 any of the superclasses. If none of the superclasses specified a type, the local class may either not specify a type at all or may 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 (and super-slot
  (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

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.

There are two level of description for access to instance structure. At the highest level, only slots are dealt with. We call this Slot Level Instance Access. At the lower level, the user has the ability to specify that additional named storage is to be allocated in an instance. This access-key level is also the focus of optimization for slot access, and hence for the additional storage specified by the user. All the methods at the Slot Level can be defined in terms of functions at the access-key level.

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

slot-value-using-class *class instance slot-name* [*Generic Function*]

The purpose of the generic function **slot-value-using-class** is to return the value of the slot with the name *slot-name*. The generic function **slot-value-using-class** is called by the function **slot-value**. **slot-value** computes the class of the instance as the first argument to **slot-value-using-class**. The behavior of **slot-value-using-class** is undefined if *class* is not the class of

instance

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** with condition-code *slot-value*. If the slot exists but is not bound calls **slot-unbound**.

slot-value-using-class

(setf slot-value-using-class) *new-value class instance slot-name* [*Generic Function*]

The purpose of the generic function **(setf slot-value-using-class)** is to set the value of the slot with the name *slot-name*. This generic function is called by the function **(setf slot-value)** which computes the class of the instance as the *class* argument to this function. The behavior of **(setf slot-value-using-class)** is undefined if *class* is not the class of *instance*

(setf slot-value-using-class) `new-value (class standard-class) instance slot-name` [*Primary Method*]

Sets the value of the slot named slot-name if such a slot exists. If the slot does not exist calls **slot-missing** with condition-code *setf*.

slot-boundp-using-class

slot-boundp-using-class *class instance slot-name* [*Generic Function*]

The purpose of the generic function **slot-boundp-using-class** is to test whether a slot *slot-name* in an instance of*class* is bound. The generic function **slot-boundp-using-class** is called by the function **slot-boundp**.

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 name *slot-name* exists, but that slot is not bound returns nil. If no slot with the given name exists, the function slot-missing is called with condition argument slot-boundp.

slot-makunbound-using-class

slot-makunbound-using-class *class instance slot-name* [*Generic Function*]

The purpose of the generic function **slot-makunbound-using-class** is to restore slot *slot-name* in instance to its unbound state. For instances of standard-class, attempting to read a slot immediately after it has been made unbound will result in a call to **slot-unbound**. **slot-makunbound-using-class** is called by the function **slot-makunbound**.

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

If a slot with the name *slot-name* exists in the class the slot is restored to its original unbound state. If there is no slot with the given name, a call is made to slot-missing with condition code slot-makunbound.

slot-exists-p-using-class

slot-exists-p-class *class instance slot-name* [*Generic Function*]

The purpose of the generic function **slot-exists-p-using-class** is to test whether a slot *slot-name* will be accessible from this instance. The generic function **slot-exists-p-using-class** is called by the function **slot-exists-p**.

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

If either an :instance or :class slot slot with the given name exists in the class, this method returns that allocation. Otherwise returns nil.

class-slot-value

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

The purpose of this generic function is to return the value of a slot whose allocation is in the class *class*.

class-slot-value `(class standard-class) slot-name` [*Primary Method*]

This returns the value of the slot *slot-name* with allocation :class in *class*. An error is signalled if there is no such slot.

Access-Key Instance Access

At the access-Key level, there is direct access to instances of metatype standard-class. However, even at this level, instances are accessed through a symbolic mapping. The set of keys used for accessing the objects is computed by a method on the class. Optimized access through those keys is supported.

Note that the user cannot portably change the mechanisms for laying out storage, nor for augmenting class-of.

compute-class-access-keys

compute-class-access-keys *class* [*Generic Function*]

The purpose of the generic function **compute-class-access-keys** is to compute the entire set of access keys that will be used with instances of a standard-class. This generic function is called by the update-dependent method on standard-class, and its values stored as the value of the reader

class-access-keys.

compute-class-access-keys `(class standard-class)` [*Primary Method*]

This method returns two values, each of which must be a list of access-keys. No access-key can appear more than once in a list, and the two lists may have no elements in common. The first value is the set of access keys for which storage will be allocated in the instance. The second is a list of access-keys for which storage will be allocated in the class. All these access-keys will be usable by standard-instance-access to fetch the contents of the storage.

Storage associated with keys in the second list will also be accesible through the function standard-class-access.

Specific implementations may extend the meaning of the access-key to provide a means to describe data packing and garbage collection control.

class-access-keys

class-access-keys *standard-class* [*Function*]

class-access-keys is a function that takes an instance of standard-class as an argument. It returns the latest values stored by update-dependent. It returns two values, a list of instance-keys and a list of class-keys.

The first set of keys will be accessible by standard-instance-access, and the second through standard-class-access.

allocate-standard-instance

allocate-standard-instance *standard-class* [*Function*]

Allocate-standard-instance is a function that takes an instance of standard-class as an argument. It uses information collected in the class by update-dependent to create a structure that has enough room for one entry for each access-key on the list of instance access-keys.

allocate-standard-instance *standard-class* [*Function*]

standard-instance-access

standard-instance-access *standard-instance access-key* [*Function*]

The function **standard-instance-access** is the low level access primitive for instances of standard-class in CLOS. It does its access based on the set of values stored for class-access-keys. If the access-key is missing it calls standard-instance-missing with the class the instance and the key. The standard method on standard-instance-missing calls slot-missing.

If the key is unbound it calls standard-instance-unbound with the class the instance and the key The standard-method on standard-instance-unbound calls slot-unbound.

If the key exists and is bound it just returns the value.

Unfortunately all we have for this is model code, no writeup yet.

Some model code follows:

```
(defun standard-instance-access (instance key)
  (let ((class (class-of instance)))
    (multiple-value-bind (instance-keys class-keys)
(class-access-keys class)
      (cond ((member key instance-keys)
     (if (internal-instance-boundp instance key)
 (internal-instance-value instance key)
 (standard-instance-unbound class instance key)))
    ((member key class-keys)
     (if (internal-class-boundp instance key)
 (internal-class-value instance key)
 (standard-instance-unbound class instance key)))
    (t
     (standard-instance-missing class instance key))))))
```

standard-class-access

```
(defun standard-class-access (instance key)
  (let ((class (class-of instance)))
    (multiple-value-bind (ignore class-keys)
(class-access-keys class)
      (if (member key class-keys)
  (internal-class-value instance key) ; class slots are always bound
  (standard-instance-missing class instance key)))))
```

standard-instance-missing

```
(defmethod standard-instance-missing ((class standard-class) instance key)
  (slot-missing class instance key))
```

standard-instance-unbound

```
(defmethod standard-instance-unbound ((class standard-class) instance key)
  (slot-unbound class instance key))
```

standard-instance-boundp

```
(defun standard-instance-boundp (instance key)
  (let ((class (class-of instance)))
    (multiple-value-bind (instance-key class-keys)
```

```
(class-access-keys class)
      (cond ((member key instance-keys)
     (internal-instance-boundp instance key))
    ((member key class-keys)
     (internal-instance-boundp instance key))
    (t
     (standard-instance-missing class instance key))))))
```

standard-instance-makunbound

The purpose of this function is make the specified by access-key be unbound.

The Instance Access Optimization Protocol

As described in chapters 1 and 2, most code access instances at the slot level. But, as described above, 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 go through this generic function call, to get to the call to standard-instance-access slot access would be unbearably slow.

To solve this problem, CLOS provides a mechanism for optimizing calls to slot-value and other instance access functions. At compile-time, this mechanism optimizes calls 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 the relevant arguments to the instance access function (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.

The fundamental hook for this mechanism is the define-instance-access-optimzation macro. This macro tells the compiler that a given function ends up calling standard-instance-access and so calls to it might be optimizable.

The general idea here is that this is a compiler optimizer interface. You declare to the compiler the function that needs to be optimized, its argument list, and for which of those arguments the class must be known. Then, in contexts where the class of that argument is known, the compiler calls the declared optimization function. The first argument will be the known class, the second argument will be the actual form. The optimization function should return the form which should be used. The returned form usually is a call to standard-instance-access.

```
;;;
;;; The CLOS system includes these calls to
;;; define-instance-access-optimization which arrange
;;; for calls to slot-value and (setf slot-value) to
;;; go through the appropriate optimization mechanism.
```

```
;;;
(define-instance-access-optimization slot-value
     (instance slot-name)
     instance
     optimize-slot-value)

(define-instance-access-optimization (setf slot-value)
     (new-value instance slot-name)
     instance
     optimize-setf-slot-value)
```

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.

optimizing slot-value

optimize-slot-value `(class standard-class) form` [*Primary Method*]

This method translates the call to slot-value into a call to standard-instance-access with the access key being the slot-name.

optimize-setf-slot-value `(class standard-class) form` [*Primary Method*]

This method translates the call to (setf slot-value) into a call to (setf standard-instance-access) with the access key being the slot-name.

Deoptimizing Instance Access

The standard-instance-access function provides the basic interface to the implementation-specific standard instance access optimization. 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 assume that the set of access-keys does not change unless update-dependent has been called, and it is the responsibility of update-dependent to inform each of the effected methods about how to update themselves.

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.

can-deoptimize-slot-accesses-p

```
(defmethod can-deoptimize-slot-accesses-p ((class standard-class))
  't)
```

deoptimize-slot-accesses

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

Deoptimizes its optimized slot accesses by calling deoptimize-standard-instance-access, with the class as the first argument, a second argument of #'slot-value-using-class and third argument of #'(setf slot-value-using-class). This causes the deoptimized slot accesses to go through the complete slot-value-using-class protocol.

deoptimize-standard-instance-access

deoptimize-standard-instance-access *class read-trap write-trap* [*Function*]

This function causes all the calls to standard-instance-access for a particular class to call the trap functions instead. The trap function received the class of the intance, the instance and the access-key as arguments.

```
(defun deoptimize-standard-instance-accesses (class read-trap write-trap)
  ;; Cause all optimized accesses to instances of class to go
  ;; through one of the trap functions instead.  Reads will go
  ;; through the read-trap function, writes will go through the
  ;; write trap function.  The reap trap function will be called
  ;; with three arguments, the class, the instance and the key.
  ;; The write trap function will be called with 4 arguments,
  ;; the new value, the class, the instance and the key.
  )
```

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.

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.

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 central component of the method lookup protocol is a piece of code called the discriminator code for the generic function. The discriminator code for a generic function is called when the generic function is called. The discriminator code must determine the effective method to be called and apply it to the arguments the generic function received.

The discriminator code is not computed at generic function invocation time, it is computed and stored whenever the generic function changes or whenever the class lattice changes in a way that affects the effective methods of the generic function. The entry to this protocol is the generic function **compute-discriminator-code**.

The standard-method on compute-discriminator-code uses the generic function **compute-effective-method** to construct an effective method from a set of applicable methods. **compute-discriminator-code** constructs code that selects from the set of effective methods to be called at run time.

In this protocol, there are several support functions supplied 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

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

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

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

This method supports the construction of an effective method of the kind described as the standard-method-combination in Chapter 1.

In addition to this method, there is one method here for each pre-defined method-combination-type. See the explanation for method-combination objects of how define-method-combination expands into a defmethod for compute-effective-method. We must also specify here which ones are specified for CLOS. The examples given in expansion of define-method-combination are illustrative.

compute-applicable-methods

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.

check-keyword-arguments

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

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

This is documented in chapter 2.

There is a problem with the method combination mechanism specified in chapter 2 which is that it doesn't allow the combined method to change the arguments that are passed to the individual methods. We should re-address the issues associated with the combined-method arguments abstraction.

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~                    func-
tions 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)))
```