The SOS Reference Manual

Edition 2.8
8 December 2000

by Chris Hanson

Introduction

SOS is a Scheme *object system* derived from Tiny CLOS[1], which in turn was loosely derived from CLOS, the Common Lisp Object System. Its basic design and philosophy is closely related to Tiny CLOS, but there are differences in naming and interface.

This document is a reference manual, and as such does not attempt to teach the reader about object-oriented programming. It is assumed that you already have a passing familiarity with CLOS and with Scheme.

In the procedure descriptions that follow, certain argument names imply restrictions on the corresponding argument. Here is a table of those names. The parenthesised name in each entry is the name of the predicate procedure that the argument must satisfy.

class The argument must be a *class* (`class?`).

instance The argument must be an *instance* (`instance?`).

name The argument must be a symbol (`symbol?`); sometimes this is also allowed to be `#f` (`false?`).

generic-procedure
The argument must be a *generic procedure* (`generic-procedure?`).

method The argument must be a *method* (`method?`).

specializer The argument must be a *method specializer* (`specializer?`).

procedure The argument must be a procedure (`procedure?`).

slot The argument must be a *slot descriptor* (`slot-descriptor?`).

[1] Tiny CLOS was written by Gregor Kiczales of Xerox PARC; SOS is derived from version 1.2 of Tiny CLOS.

1 Classes

A *class* is an object that determines the structure and behavior of a set of other objects, which are called its *instances*. However, in this document, the word *instance* usually means an instance of the class `<instance>`.

A class can inherit structure and behavior from other classes. A class whose definition refers to other classes for the purpose of inheriting from them is said to be a *subclass* of each of those classes. The classes that are designated for purposes of inheritance are said to be *superclasses* of the inheriting class.

A class can have a *name*. The procedure `class-name` takes a class object and returns its name. The name of an anonymous class is `#f`.

A class *C_1* is a *direct superclass* of a class *C_2* if *C_2* explicitly designates *C_1* as a superclass in its definition. In this case, *C_2* is a *direct subclass* of *C_1*. A class *C_n* is a *superclass* of a class *C_1* if there exists a series of classes *C_2*, ..., *C_n-1* such that *C_i+1* is a direct superclass of *C_i* for all *i* between *1* and *n*. In this case, *C_1* is a *subclass* of *C_n*. A class is considered neither a superclass nor a subclass of itself. That is, if *C_1* is a superclass of *C_2*, then *C_1* is different from *C_2*. The set of classes consisting of some given class *C* along with all of its superclasses is called "*C* and its superclasses."

Each class has a *class precedence list*, which is a total ordering on the set of the given class and its superclasses. The total ordering is expressed as a list ordered from the most specific to the least specific. The class precedence list is used in several ways. In general, more specific classes can *shadow*, or override, features that would otherwise be inherited from less specific classes. The method selection and combination process uses the class precedence list to order methods from most specific to least specific.

When a class is defined, the order in which its direct superclasses are mentioned in the defining form is important. Each class has a *local precedence order*, which is a list consisting of the class followed by its direct superclasses in the order mentioned in the defining form.

A class precedence list is always consistent with the local precedence order of each class in the list. The classes in each local precedence order appear within the class precedence list in the same order. If the local precedence orders are inconsistent with each other, no class precedence list can be constructed, and an error is signalled.

Classes are organized into a *directed acyclic graph*. There are two distinguished classes, named `<object>` and `<instance>`. The class named `<object>` has no superclasses. It is a superclass of every class except itself. The class named `<instance>` is a direct subclass of `<object>` and is the base class for *instance* objects. Instances are special because SOS has efficient mechanisms for dispatching on them and for accessing their slots.

1.1 Class Datatype

The procedures in this section may be used to construct and inspect classes.

make-class *name direct-superclasses direct-slots* Procedure

Creates and returns a new class object.

Name is used for debugging: it is a symbol that appears in the printed representation of the class and has no role in the semantics of the class. Alternatively, *name* may be `#f` to indicate that the class is anonymous.

Direct-superclasses must be a list of class objects. The new class inherits both methods and slots from the classes in this list. Specifying the empty list for *direct-superclasses* is equivalent to specifying `(list <instance>)`.

Direct-slots describes additional slots that instances of this class will have. It is a list, each element of which must have one of the following forms:

name
(*name* . *plist*)

where *name* is a symbol, and *plist* is a property list. The first of these two forms is equivalent to the second with an empty *plist*.

Each of the elements of *direct-slots* defines one slot named *name*. *Plist* is used to describe additional properties of that slot. The following properties are recognized:

`initial-value`
: This property specifies the default initial value for the slot, i.e. the value stored in the slot when an instance is created and no value is explicitly specified by the instance constructor. If neither the `initial-value` nor the `initializer` property is specified, the slot has no default initial value.

`initializer`
: This property specifies a procedure of no arguments that is called by an instance constructor whenever an instance containing this slot is created. The value returned by the `initializer` procedure is the initial value of the slot.

`accessor`
: This property specifies a generic procedure; `make-class` will add an accessor method for this slot to the procedure. See Chapter 3 [Slots], page 15.

`modifier`
: This property specifies a generic procedure; `make-class` will add a modifier method for this slot to the procedure. See Chapter 3 [Slots], page 15.

`initpred`
: This property specifies a generic procedure; `make-class` will add an "initialized?" predicate method for this slot to the procedure. See Chapter 3 [Slots], page 15.

Slot properties are combined in slightly complicated ways.

- It is not allowed to specify both `initial-value` and `initializer` for a slot in a given call to `make-class`; at most one of these properties may be given.
- If a slot is specified for a given class, and a slot of the same name is inherited from a superclass, then the slot properties for the two slots are combined. Slot properties from the subclass shadow those of the superclass. However, if a superclass has a slot property, and the subclass does not, the property is inherited. The resulting class never has more than one slot of a given name.
- When combining superclass and subclass slots, `initial-value` and `initializer` shadow one another. In other words, regardless of the inherited slot properties, the resulting slot has at most one of these two properties.

Examples of `make-class`:

```
(define <cell>
  (make-class '<cell> '() '()))

(define <contact>
  (make-class '<contact>
              (list <cell>)
              `((name accessor ,cell-name))))

(define <compound-cell>
  (make-class '<compound-cell>
              (list <cell>)
              `((width accessor ,cell-width)
                (height accessor ,cell-height)
                (components accessor ,cell-components
                            modifier ,set-cell-components!
                            initial-value ()))))
```

define-class *name direct-superclasses direct-slot ...* Syntax

Define *name* to be a class. In its basic form, `define-class` might have been defined by

```
(define-syntax define-class
  (syntax-rules ()
    ((define-class name (class ...) slot ...)
     (define name
       (make-class (quote name)
                   (list class ...)
                   (quote (slot ...)))))))
```

Note that slot properties are handled specially by `define-class`. If a *direct-slot* specifies a slot properties property list, the keys of the property list (i.e. the even-numbered elements) are not evaluated, while the datums of the property list *are* evaluated. The expansion above does not show the proper treatment of slot properties.

In addition to the slot properties recognized by `make-class`, `define-class` recognizes a special slot property, called `define`. The `define` property specifies that some or all of the slot accessors should be defined here; that is, generic procedures should be constructed and bound to variables, and then the accessor methods added to them.

The argument to the `define` property is a list containing any combination of the symbols `accessor`, `modifier`, and `initpred`. As an abbreviation, the argument may be one of these symbols by itself, which is equivalent to the list containing that symbol. Also, the argument may be the symbol `standard`, which is equivalent to `(accessor modifier)`.

The argument to `define` specifies the accessors that will be defined by this form. The accessors are defined using default names, unless the names are overridden by the corresponding slot property. The default names for a class `<foo>` and a slot `bar` are `foo-bar`, `set-foo-bar!`, and `foo-bar-initialized?`, respectively for the accessor, modifier, and initpred. For example,

```
(define-class foo
  (bar define accessor))
```

defines an accessor called `foo-bar`, but

```
(define-class foo
  (bar define accessor accessor foo/bar))
```

instead defines an accessor called `foo/bar`. Finally,

```
(define-class foo
  (bar accessor foo/bar))
```

doesn't define any accessor, but assumes that `foo/bar` is a previously-defined generic procedure and adds an accessor method to it.

`define-class` permits the specification of *class options*, which are options that pertain to the class as a whole. Class options are specified by overloading *name*: instead of a symbol, specify a pair whose CAR is a symbol and whose CDR is an alist. The following class options are recognized:

`(predicate [`*name*`])`
: Specifies that a predicate procedure should be defined for this class. *Name* must be either a symbol or `#f`: a symbol specifies the name that will be bound to the predicate procedure, and `#f` specifies that no predicate procedure should be defined. If *name* is omitted, or if no `predicate` option is specified, a predicate procedure is defined by appending `?` to the name of the class. If the class name is surrounded by angle brackets, they are stripped off first. For example, the default predicate name for the class `<foo>` is `foo?`.

`(constructor [`*name*`]` *slot-names* `[`*n-init-args*`])`
: Specifies that a constructor procedure should be defined for this class. *Name* must be a symbol, which is the name that will be bound to the constructor procedure; if omitted, a default name is formed by prepending `make-` to the name of the class. If the class name is surrounded by angle brackets, they are stripped off first. For example, the default constructor name for the class `<foo>` is `make-foo`.

  Slot-names and *n-init-args* correspond to the arguments of the respective names accepted by `instance-constructor`, and can take any of the allowed forms for those arguments.

`(separator` *string*`)`
: Specifies how names for slot accessors are constructed. If this option isn't given, the name of a slot accessor is formed by concatenating the name of the class with the name of the slot, with a hyphen between them. When this option is given, *string* is used instead of the hyphen. For example, normally a slot accessor for the slot `bar` in the class `foo` is called `foo-bar`. A class option `(separator ".")` will cause the slot accessor to be called `foo.bar`, the modifier to be called `set-foo.bar!`, and the initialization predicate to be called `foo.bar?`.

Examples of `define-class` (compare these to the similar examples for `make-class`):

```
(define-class <cell> ())
```

```
(define-class (<contact> (constructor (name) no-init)) (<cell>)
  (name accessor cell-name))

(define-class (<compound-cell> (constructor ())) (<cell>)
  (width accessor cell-width)
  (height accessor cell-height)
  (components accessor cell-components
              modifier set-cell-components!
              initial-value '()))
```

make-trivial-subclass *superclass1 superclass2 . . .* Procedure

This convenience procedure makes a subclass that defines no new slots, and that inherits from the given superclasses. It is equivalent to the following

```
(make-class (class-name superclass1)
            (list superclass1 superclass2 ...)
            '())
```

class? *object* Procedure

Returns `#t` if *object* is a class, otherwise returns `#f`.

subclass? *class specializer* Procedure

Returns `#t` if *class* is a subclass of *specializer*, otherwise returns `#f`. If *specializer* is a class, the result follows from the above definition of subclass, except that a class is a subclass of itself. If *specializer* is a record type, it is equivalent to having used the `record-type-class` of the record type. Finally, if *specializer* is a union specializer, `subclass?` is true if *class* is a subclass of one or more of the component classes of *specializer*.

object-class *object* Procedure

Returns the class of *object*. *Object* may be any Scheme object; if *object* is known to be an instance, `instance-class` is faster than `object-class`.

class-name *class* Procedure

Returns the name of *class*. This is the *name* argument passed to `make-class` when *class* was created.

class-direct-superclasses *class* Procedure

Returns a list of the direct superclasses of *class*. If a non-empty *direct-superclasses* argument was passed to `make-class` when *class* was created, this list is `equal?` to that argument. The returned value must not be modified.

class-direct-slot-names *class* Procedure

Returns a list of symbols that are the names of the direct slots of *class*. This list contains only those slots that were defined in the call to `make-class` that created *class*; it does not contain slots that were inherited. The returned value must not be modified.

class-precedence-list *class* Procedure
Returns a list of the superclasses of *class*. The order of this list is significant: it is the method resolution order. This list will always have *class* as its first element, and `<object>` as its last element. The returned value must not be modified.

1.2 Predefined Classes

SOS provides a rich set of predefined classes that can be used to specialize methods to any of Scheme's built-in datatypes.

<object> Class
This is the class of all Scheme objects. It has no direct superclasses, and all other classes are subclasses of this class.

<instance> Class
This is the class of instances. It is a direct subclass of `<object>`. The members of this class are the objects that satisfy the predicate `instance?`.

<boolean> Class
<char> Class
<entity> Class
<pair> Class
<procedure> Class
<record> Class
<string> Class
<symbol> Class
<vector> Class
These are the classes of their respective Scheme objects. They are all direct subclasses of `<object>`. The members of each class are the objects that satisfy the corresponding predicate; for example, the members of `<procedure>` are the objects that satisfy `procedure?`.

<generic-procedure> Class
This is the class of generic procedure instances. It is a direct subclass of `<procedure>`.

<method> Class
This is the class of method objects. It is a direct subclass of `<instance>`.

<chained-method> Class
<computed-method> Class
<computed-emp> Class
These classes specify additional method objects with special properties. Each class is a subclass of `<method>`.

The following are the classes of Scheme numbers. Note that `object-class` will never return one of these classes; instead it returns an implementation-specific class that is associated with a particular numeric representation. The implementation-specific class is a subclass of one or more of these implementation-independent classes, so you should use these classes for specialization.

<number> Class
<complex> Class
<real> Class
<rational> Class
<integer> Class

These are the classes of the Scheme numeric tower. `<number>` is a direct subclass of `<math-object>`, `<complex>` is a direct subclass of `<number>`, `<real>` is a direct subclass of `<complex>`, etc.

<exact> Class
<exact-complex> Class
<exact-real> Class
<exact-rational> Class
<exact-integer> Class

These are the classes of exact numbers. `<exact>` is a direct subclass of `<number>`, `<exact-complex>` is a direct subclass of `<exact>` and `<complex>`, and in general, each is a direct subclass of preceding class and of the class without the `exact-` prefix.

<inexact> Class
<inexact-complex> Class
<inexact-real> Class
<inexact-rational> Class
<inexact-integer> Class

These are the classes of inexact numbers. `<inexact>` is a direct subclass of `<number>`, `<inexact-complex>` is a direct subclass of `<inexact>` and `<complex>`, and in general, each is a direct subclass of preceding class and of the class without the `inexact-` prefix.

1.3 Record Classes

SOS allows generic procedures to discriminate on record types. This means that a record structure defined by means of `make-record-type` or `define-structure` can be passed as an argument to a generic procedure, and the generic procedure can use the record's type to determine which method to be invoked.[1]

In order to support this, SOS accepts record type descriptors in all contexts that accept classes. Additionally, every record type descriptor has an associated SOS class; either the class or the record type can be used with equivalent results.

[1] If the `type` option of `define-structure` is used, the resulting data structure is *not* a record and thus cannot be used in this manner.

record-type-class *record-type* Procedure

Record-type must be a record type descriptor (in other words, it must satisfy the predicate `record-type?`). Returns the class associated with *record-type*.

record-class *record* Procedure

Record must be a record (in other words, it must satisfy the predicate `record?`). Returns the class associated with *record*. This is equivalent to

```
(record-type-class (record-type-descriptor record))
```

1.4 Specializers

A *specializer* is a generalization of a class. A specializer is any one of the following:

- A class.
- A record type, which is equivalent to its associated class.
- A union specializer, which is a set of classes.

A specializer may be used in many contexts where a class is required, specifically, as a method specializer (hence the name), as the second argument to `subclass?`, and elsewhere.

specializer? *object* Procedure

Returns `#t` if *object* is a specializer, otherwise returns `#f`.

specializer-classes *specializer* Procedure

Returns a list of the classes in *specializer*. If *specializer* is a class, the result is a list of that class. If *specializer* is a record type, the result is a list of the record type's class. If *specializer* is a union specializer, the result is a list of the component classes of the specializer.

specializer=? *specializer1 specializer2* Procedure

Returns `#t` if *specializer1* and *specializer2* are equivalent, otherwise returns `#f`. Two specializers are equivalent if the lists returned by `specializer-classes` contain the same elements.

union-specializer *specializer . . .* Procedure

Returns a union specializer consisting of the union of the classes of the arguments. This is equivalent to converting all of the specializer arguments to sets of classes, then taking the union of those sets.

union-specializer? *object* Procedure

Returns `#t` if *object* is a union specializer, otherwise returns `#f`.

specializers? *object* Procedure

Returns `#t` if *object* is a list of specializers, otherwise returns `#f`.

specializers=? *specializers1 specializers2* Procedure

Specializers1 and *specializers2* must be lists of specializers. Returns `#t` if *specializers1* and *specializers2* are equivalent, otherwise returns `#f`. Two specializers lists are equivalent if each of their corresponding elements is equivalent.

2 Instances

An *instance* is a compound data structure much like a record, except that it is defined by a class rather than a record type descriptor. Instances are more powerful than records, because their representation is designed to support inheritance, while the representation of records is not.

instance-constructor *class slot-names* [*n-init-args*] Procedure

Creates and returns a procedure that, when called, will create and return a newly allocated instance of *class*.

Class must be a subclass of `<instance>`. *Slot-names* must be a list of symbols, each of which must be the name of a slot in *class*. *N-init-args* will be described below.

In its basic operation, `instance-constructor` works much like `record-constructor`: the *slot-names* argument specifies how many arguments the returned constructor accepts, and each of those arguments is stored in the corresponding slot of the returned instance. Any slots that are not specified in *slot-names* are given their initial values, as specified by the `initial-value` or `initializer` slot properties; otherwise they are left uninitialized.

After the new instance is created and its slots filled in, but before it is returned, it is passed to the generic procedure `initialize-instance`. Normally, `initialize-instance` does nothing, but because it is always called, the programmer can add methods to it to specify an initialization that is to be performed on every instance of the class.

By default, `initialize-instance` is called with one argument, the newly created instance. However, the optional argument *n-init-args* can be used to specify additional arguments that will be passed to `initialize-instance`.

The way this works is that the returned constructor procedure accepts additional arguments after the specified number of slot values, and passes these extra arguments to `initialize-instance`. When *n-init-args* is not supplied or is `#t`, any number of extra arguments are accepted and passed along. When *n-init-args* is an exact non-negative integer, exactly that number of extra arguments must be supplied when the constructor is called. Finally, if *n-init-args* is the symbol `no-initialize-instance`, then the constructor accepts no extra arguments and does not call `initialize-instance` at all; this is desirable when `initialize-instance` is not needed, because it makes the constructor significantly faster.

For notational convenience, *n-init-args* may take two other forms. First, it may be a list of symbols, which is equivalent to the integer that is the length of the list. Second, it may be the symbol `no-init`, which is an abbreviation for `no-initialize-instance`.

Note that the default method on `initialize-instance` accepts no extra arguments and does nothing.

Examples of `instance-constructor`:

```
(define-class <simple-reference> (<reference>)
  (from accessor reference-from)
  (to accessor reference-to)
  (cx accessor reference-cx)
  (cy accessor reference-cy))

(define make-simple-reference
  (instance-constructor <simple-reference>
                        '(from to cx cy)
                        'no-init))

(define-class <simple-wirenet> (<wirenet>)
  (cell accessor wirenet-cell)
  (wires accessor wirenet-wires
         modifier set-wirenet-wires!
         initial-value '()))

(define make-simple-wirenet
  (instance-constructor <simple-wirenet> '(cell)))
```

instance? *object* Procedure

Returns `#t` if *object* is an instance, otherwise returns `#f`.

instance-class *instance* Procedure

Returns the class of *instance*. This is faster than `object-class`, but it works only for instances, and not for other objects.

instance-of? *object specializer* Procedure

Returns `#t` if *object* is a general instance of *specializer*, otherwise returns `#f`. This is equivalent to

```
(subclass? (object-class object) specializer)
```

instance-predicate *specializer* Procedure

Returns a predicate procedure for *specializer*. The returned procedure accepts one argument and returns `#t` if the argument is an instance of *specializer* and `#f` otherwise.

3 Slots

An instance has zero or more named slots; the name of a slot is a symbol. The slots of an instance are determined by its class.

Each slot can hold one value. When a slot does not have a value, the slot is said to be *uninitialized*. The default initial value for a slot is defined by the `initial-value` and `initializer` slot properties.

A slot is said to be *accessible* in an instance of a class if the slot is defined by the class of the instance or is inherited from a superclass of that class. At most one slot of a given name can be accessible in an instance. Slots are accessed by means of slot-access methods (usually generated by `make-class`).

3.1 Slot Descriptors

Slots are represented by *slot descriptors*, which are data structures providing information about the slots, such as their name. Slot descriptors are stored inside of classes, and may be retrieved from there and subsequently inspected.

class-slots *class* Procedure

Returns a list of the slot descriptors for *class*. This contains all slots for *class*, both direct slots and inherited slots. The returned value must not be modified.

class-slot *class name error?* Procedure

Returns the slot descriptor for the slot named *name* in *class*. If there is no such slot: if *error?* is `#f`, returns `#f`, otherwise signals an error of type `condition-type:no-such-slot`.

slot-descriptor? *object* Procedure

Returns `#t` if *object* is a slot descriptor, otherwise returns `#f`.

slot-name *slot* Procedure

Returns the name of *slot*.

slot-class *slot* Procedure

Returns the class of *slot*. This is the class with which *slot* is associated. This is not necessarily the class that defines *slot*; it could also be a subclass of that class. If the slot was returned from `class-slots` or `class-slot`, then this class is the argument passed to that procedure.

slot-properties *slot* Procedure

Returns an alist of the properties of *slot*. This list must not be modified.

slot-property *slot name default* Procedure

If *slot* has a property named *name*, it is returned; otherwise *default* is returned.

slot-initial-value? *slot* Procedure

Returns `#t` if *slot* has an initial value, and `#f` otherwise. The initial value is specified by the `initial-value` slot property when a class is made.

slot-initial-value *slot* Procedure

Returns the initial value for *slot*, if it has one; otherwise it returns an unspecified value. The initial value is specified by the `initial-value` slot property when a class is made.

slot-initializer *slot* Procedure

Returns the initializer for *slot*; the initializer is specified by the `initializer` slot property when a class is made. This is a procedure of no arguments that is called to produce an initial value for *slot*. The result may also be `#f` meaning that the slot has no initializer.

3.2 Slot Access Methods

The procedure `make-class` provides slot properties that generate methods to read and write slots. If an *accessor* is requested, a method is automatically generated for reading the value of the slot. If a *modifier* is requested, a method is automatically generated for storing a value into the slot. When an accessor or modifier is specified for a slot, the generic procedure to which the generated method belongs is directly specified. The procedure specified for the accessor takes one argument, the instance. The procedure specified for the modifier takes two arguments, the instance and the new value, in that order.

All of the procedures described here signal an error of type `condition-type:no-such-slot` if the given class or object does not have a slot of the given name.

Slot-access methods can be generated by the procedures `slot-accessor-method`, `slot-modifier-method`, and `slot-initpred-method`. These methods may be added to a generic procedure by passing them as arguments to `add-method`. The methods generated by these procedures are equivalent to those generated by the slot properties in `make-class`.

slot-accessor-method *class name* Procedure

Returns an accessor method for the slot *name* in *class*. The returned method has one required argument, an instance of *class*, and the specializer for that argument is *class*. When invoked, the method returns the contents of the slot specified by *name* in the instance; if the slot is uninitialized, an error of type `condition-type:uninitialized-slot` is signalled.

```
(define-generic get-bar (object))

(add-method get-bar
  (slot-accessor-method <foo> 'bar))
```

slot-modifier-method *class name* Procedure

Returns a modifier method for the slot *name* in *class*. The returned method has two required arguments, an instance of *class* and an object. The specializer for the first

argument is *class* and the second argument is not specialized. When invoked, the method stores the second argument in the slot specified by *name* in the instance.

```
(define-generic set-bar! (object bar))

(add-method set-bar!
  (slot-modifier-method <foo> 'bar))
```

slot-initpred-method *class name* Procedure

Returns an "initialized?" predicate method for the slot *name* in *class*. The returned method has one required argument, an instance of *class*, and the specializer for that argument is *class*. When invoked, the method returns `#t` if the slot specified by *name* is initialized in the instance; otherwise it returns `#f`.

```
(define-generic has-bar? (object))

(add-method has-bar?
  (slot-initpred-method <foo> 'bar))
```

3.3 Slot Access Constructors

For convenience, and for consistency with the record-accessor procedures `record-accessor` and `record-modifier`, each of the above method-generating procedures has a corresponding accessor-generator. Each of these procedures creates a generic procedure, adds an appropriate method to it by calling the corresponding method-generating procedure, and returns the generic procedure. Thus, for example, the following are equivalent:

```
(slot-accessor <foo> 'bar)

(let ((g (make-generic-procedure 1)))
  (add-method g (slot-accessor-method <foo> 'bar))
  g)
```

slot-accessor *class name* Procedure

Returns a generic procedure of one argument that is an accessor for the slot *name* in *class*. The argument to the returned procedure must be an instance of *class*. When the procedure is called, it returns the contents of the slot *name* in that instance; if the slot is uninitialized, an error of type `condition-type:uninitialized-slot` is signalled.

slot-modifier *class name* Procedure

Returns a generic procedure of two arguments that is a modifier for the slot *name* in *class*. The first argument to the returned procedure must be an instance of *class*, and the second argument may be any object. When the procedure is called, it modifies the slot *name* in the instance to contain the second argument.

slot-initpred *class name* Procedure

Returns a generic procedure of one argument that is an "initialized?" predicate for the slot *name* in *class*. The argument to the returned procedure must be an instance

of *class*. When the procedure is called, it returns `#t` if the slot *name* in that instance is initialized, otherwise it returns `#f`.

3.4 Slot Access Procedures

Finally, there is another set of three procedures, which access the contents of a slot directly, given an instance and a slot name. These procedures are very slow by comparison with the above.

However, note the following. You can use these procedures in the body of a `define-method` special form in an efficient way. If the `define-method` specifies the correct number of arguments, the body of the form contains a call to one of these procedures and nothing else, and the specified slot name is quoted, the form is rewritten during macro-expansion time as a call to the corresponding method-generating procedure. For example, the following are equivalent:

```
(define-method p ((v <foo>))
  (slot-value v 'bar))

(add-method p
            (slot-accessor-method <foo> 'bar))
```

slot-value *instance name* Procedure

Returns the contents of the slot *name* in *instance*; if the slot is uninitialized, an error of type `condition-type:uninitialized-slot` is signalled.

set-slot-value! *instance name object* Procedure

Modifies the slot *name* in *instance* to contain *object*.

slot-initialized? *instance name* Procedure

Returns `#t` if the slot *name* in *instance* is initialized, otherwise returns `#f`.

4 Generic Procedures

Like an ordinary Scheme procedure, a generic procedure takes arguments, performs a series of operations, and perhaps returns useful values. An ordinary procedure has a single body of code that is always executed when the procedure is called. A generic procedure has a set of multiple bodies of code, called methods, from which a subset is selected for execution. The selected bodies of code and the manner of their combination are determined by the classes of one or more of the arguments to the generic procedure.

Ordinary procedures and generic procedures are called with identical procedure-call syntax.

Generic procedures are true procedures that can be passed as arguments, returned as values, and otherwise used in all the ways an ordinary procedure may be used. In particular, generic procedures satisfy the predicate `procedure?`.

4.1 Generic Procedure Datatype

The following definitions are used to construct and inspect generic procedures.

make-generic-procedure *arity* [*name*] Procedure

Creates and returns a new generic procedure. The generic procedure requires *arity* arguments.

Arity may take one of the following forms. An exact positive integer specifies that the procedure will accept exactly that number of arguments. A pair of two exact positive integers specifies inclusive lower and upper bounds, respectively, on the number of arguments accepted; the CDR may be `#f` indicating no upper bound.

Name is used for debugging: it is a symbol that has no role in the semantics of the generic procedure. *Name* may be `#f` to indicate that the generic procedure is anonymous. If *name* is not specified, it defaults to `#f`.

Examples:

```
(define foo-bar (make-generic-procedure 2))

(define foo-baz (make-generic-procedure '(1 . 2) 'foo-baz))

(define foo-mum (make-generic-procedure '(1 . #f)))
```

define-generic *name lambda-list* Syntax

Defines *name* to be a generic procedure. *Lambda-list* is an ordinary parameter list, which is exactly like the parameter list in a `lambda` special form. This expands into

```
(define name
  (make-generic-procedure arity
                          (quote name)))
```

where *arity* is determined from *lambda-list*.

Examples (compare to examples of `make-generic-procedure`):

```
(define-generic foo-bar (x y))

(define-generic foo-baz (x #!optional y))

(define-generic foo-mum (x . y))
```

generic-procedure? *object* Procedure
Returns `#t` if *object* is a generic procedure, otherwise returns `#f`. Note that every generic procedure satisfies the predicate `procedure?`.

generic-procedure-arity *generic-procedure* Procedure
Returns the arity of *generic-procedure*, as specified in the call to `make-generic-procedure`. The returned arity must not be modified.

generic-procedure-name *generic-procedure* Procedure
Returns the name of *generic-procedure*, as specified in the call to `make-generic-procedure`.

4.2 Method Storage

Methods are stored in generic procedures. When a generic procedure is called, it selects a subset of its stored methods (using `method-applicable?`), and arranges to invoke one or more of the methods as necessary. The following definitions provide the means for adding methods to and removing them from a generic procedure.

add-method *generic-procedure method* Procedure
Adds *method* to *generic-procedure*. If *generic-procedure* already has a method with the same specializers as *method*, then the old method is discarded and *method* is used in its place.

delete-method *generic-procedure method* Procedure
Removes *method* from *generic-procedure*. Does nothing if *generic-procedure* does not contain *method*.

add-methods *generic-procedure methods* Procedure
Adds *methods*, which must be a list of methods, to *generic-procedure*. Equivalent to calling `add-method` on each method in *methods*.

generic-procedure-methods *generic-procedure* Procedure
Returns a list of the methods contained in *generic-procedure*. The returned list must not be modified.

4.3 Effective Method Procedure

When a generic procedure is called, it arranges to invoke a subset of its methods. This is done by *combining* the selected methods into an *effective method procedure*, or EMP, then tail-recursively invoking the EMP. `compute-effective-method-procedure` is the procedure that is called to select the applicable methods and combine them into an EMP.

compute-effective-method-procedure *generic-procedure classes* Procedure

Collects the applicable methods of *generic-procedure* by calling `method-applicable?` on each method and on *classes*. Combines the resulting methods together into an effective method procedure, and returns that EMP.

compute-method *generic-procedure classes* Procedure

This procedure is like `compute-effective-method-procedure`, except that it returns the result as a method whose specializers are *classes*.

`compute-method` is equivalent to

```
(make-method classes
             (compute-effective-method-procedure generic-procedure
                                                 classes))
```

5 Methods

A method contains a method procedure and a sequence of *parameter specializers* that specify when the given method is applicable.

A method is not a procedure and cannot be invoked as a procedure. Methods are invoked by the effective method procedure when a generic procedure is called.

5.1 Method Datatype

The following procedures are used to construct and inspect methods.

make-method *specializers procedure* Procedure

Creates and returns a new method. Note that *specializers* may have fewer elements than the number of required parameters in *procedure*; the trailing parameters are considered to be specialized by `<object>`.

After the returned method is stored in a generic procedure, *Procedure* is called by the effective method procedure of the generic procedure when the generic procedure is called with arguments satisfying *specializers*. In simple cases, when no method combination occurs, *procedure is* the effective method procedure.

method? *object* Procedure

Returns `#t` iff *object* is a method, otherwise returns `#f`.

method-specializers *method* Generic Procedure

Returns the specializers of *method*. This list must not be modified.

method-procedure *method* Generic Procedure

Returns the procedure of *method*.

method-applicable? *method classes* Procedure

This predicate is used to determine the applicability of *method*. When a method is contained in a generic procedure, and the procedure is applied to some arguments, the method is *applicable* if each argument is an instance of the corresponding method specializer, or equivalently, if each argument's class is a subclass of the corresponding method specializer.

`method-applicable?` determines whether *method* would be applicable if the given arguments had the classes specified by *classes*. It returns `#t` if each element of *classes* is a subclass of the corresponding specializer of *method*, and `#f` otherwise.

5.2 Method Syntax

The following syntactic forms greatly simplify the definition of methods, and of adding them to generic procedures.

define-method *generic-procedure lambda-list body ...* Syntax

Defines a method of *generic-procedure*. *Lambda-list* is like the parameter list of a `lambda` special form, except that the required parameters may have associated specializers. A parameter with an associated specializer is written as a list of two elements: the first element is the parameter's name, and the second element is an expression that evaluates to a class.

Lambda-list must contain at least one required parameter, and at least one required parameter must be specialized.

A `define-method` special form expands into the following:

```
(add-method generic-procedure
  (make-method (list specializer ...)
    (lambda (call-next-method . stripped-lambda-list)
      body ...)))
```

where *stripped-lambda-list* is *lambda-list* with the specialized parameters replaced by their names, and the *specializer*s are the corresponding expressions from the specialized parameters. If necessary, the *specializer*s are interspersed with references to `<object>` in order to make them occur in the correct position in the sequence.

For example,

```
(define-method add ((x <integer>) (y <rational>)) ...)
```

expands into

```
(add-method add
  (make-method (list <integer> <rational>)
    (lambda (call-next-method x y) ...)))
```

Note that the list of specializers passed to `make-method` will correspond to the required parameters of the method; the specializer corresponding to a non-specialized required parameter is `<object>`.

Further note that, within the body of a `define-method` special form, the free variable `call-next-method` is bound to a "call-next-method" procedure (see `make-chained-method` for details). If the `define-method` body refers to this variable, the defined method is a chained method, otherwise it is an ordinary method.

method *lambda-list body ...* Syntax

This special form evaluates to an anonymous method in the same way that `lambda` evaluates to an anonymous procedure. *Lambda-list* and *body* have exactly the same syntax and semantics as the corresponding parts of `define-method`. Note that the following are completely equivalent:

```
(define-method g ((a <foo>) b)
  (cons b a))

(add-method g
  (method ((a <foo>) b)
    (cons b a)))

(add-method g
  (make-method (list <foo>)
    (lambda (a b)
      (cons b a))))
```

5.3 Chained Methods

Sometimes it is useful to have a method that adds functionality to existing methods. *Chained methods* provide a mechanism to accomplish this. A chained method, when invoked, can call the method that would have been called had this method not been defined: it is passed a procedure that will call the inherited method. The chained method can run arbitrary code both before and after calling the inherited method.

make-chained-method *specializers procedure* Procedure

Create and return a chained method. *Procedure* must be a procedure of one argument that returns a procedure. When the chained method is combined, its procedure will be called with one argument, a "call-next-method" procedure; it must then return another procedure that will be called when the method is invoked. The "call-next-method" procedure may called by the method procedure at any time, which will invoke the next less-specific method. The "call-next-method" procedure must be called with the same number of arguments as the method procedure; normally these are the same arguments, but that is not required.

chained-method? *object* Procedure

Returns `#t` if *object* is a chained method, otherwise returns `#f`. Note that every chained method satisfies `method?`.

5.4 Computed Methods

A *computed method* is a powerful mechanism that provides the ability to generate methods "on the fly". A computed method is like an ordinary method, except that its procedure is called during method combination, and is passed the classes of the arguments in place of the arguments themselves. Based on these classes, the computed method returns an ordinary method, which is combined in the usual way.

Note that computed methods and computed EMPs both satisfy the predicate `method?`. They are not really methods in that they cannot be combined with other methods to form an effective method procedure; however, they are treated as methods by procedures such as `add-method` and `method-specializers`.

make-computed-method *specializers procedure* Procedure

Create and return a computed method. *Procedure* will be called during method combination with the classes of the generic-procedure arguments as its arguments. It must return one of the following:

- An ordinary method (as returned by `make-method` or `make-chained-method`). The returned method's specializers must be restrictions of *specializers*, i.e. each specializer in the returned method must be a subclass of the corresponding specializer in *specializers*. In the usual case, the returned method's specializers are the same as *specializers*.
- A procedure, which is converted into an ordinary method by calling `make-method` on *specializers* and the returned procedure.
- `#f`, which means that the computed method declines to generate a method.

computed-method? *object* Procedure

Returns `#t` if *object* is a computed method, otherwise returns `#f`.

A *computed* EMP takes the computed-method mechanism one step further. A computed EMP is like a computed method, except that it returns an effective method procedure rather than a method. `compute-effective-method-procedure` tries each of the applicable computed EMPs, and if exactly one of them returns an EMP, that is the resulting effective method procedure.

make-computed-emp *key specializers procedure* Procedure

Create and return a computed EMP. *Procedure* will be called during method combination with the classes of the generic-procedure arguments as its arguments. It must return either an EMP or `#f`.

Key is an arbitrary object that is used to identify the computed EMP. The *key* is used by `add-method` and `delete-method` to decide whether two computed EMPs are the same; they are the same if their *key*s are `equal?`. This is necessary because a generic procedure may have more than one computed EMP with the same specializers.

computed-emp? *object* Procedure

Returns `#t` if *object* is a computed EMP, otherwise returns `#f`.

computed-emp-key *computed-emp* Generic Procedure

Returns the key for *computed-emp*.

6 Printing

The following procedures can be used to define a custom printed representation for an instance. It is highly recommended that instances be printed by `write-instance-helper`, as this ensures a uniform appearance for all objects.

write-instance *instance port* Generic Procedure

This is called by the runtime system to generate the printed representation of *instance*. The methods of this procedure should write the representation to *port*.

write-instance-helper *name instance port thunk* Procedure

This writes a standardized "frame" for a printed representation method. It generates the following output on *port*:

`#[`*name hash-number*`...]`

where *hash-number* is the result of calling `hash` on *instance*, and ... is the output generated by *thunk*.

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long

as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may

include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> `Copyright (C)` *year* *your name*.
> `Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being` *list their titles*`, with the Front-Cover Texts being` *list*`, and with the Back-Cover Texts being` *list*`. A copy of the license is included in the section entitled ``GNU Free Documentation License''.`

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Binding Index

<

A

C

D

G

I

M

O

P

R

S

U

W

Concept Index

Table of Contents