## Getting Ready for CLOS

### About This Document: Getting Ready for CLOS

The purpose of this document, "*Getting Ready for CLOS*", is to give Symbolics users information about the Symbolics implementation of CLOS, which is currently under development.

The main reason for making this information available in advance of the Symbolics CLOS implementation is to help users more quickly develop code to be run under CLOS, by using Flavors in place of CLOS during the initial stages of development and then translating to CLOS when it becomes available. Note that Flavors will continue to be supported, so there is no need to convert programs from Flavors to CLOS; this decision is up to the individual user.

This document describes the differences between Flavors and CLOS. It states, for each Flavors feature and operator, the name of the analogous CLOS feature or operator. In some cases, CLOS has no analogous capability, so users who plan on converting code from Flavors to CLOS can use this information to avoid developing code that depends on such Flavors features.

The section ""Questions and Answers About Symbolics CLOS"" gives some background information about CLOS itself, and tells where users can find more information about CLOS. This document does not give overview or tutorial information about CLOS, nor does it give reference documentation; both will be available with the Symbolics CLOS implementation.

### Questions and Answers About Symbolics CLOS

We introduce Symbolics CLOS with a series of questions and answers.

**Q:** What is CLOS?

**A:** CLOS stands for "Common Lisp Object System." It is an object-oriented programming language similar to Flavors. CLOS is part of the Common Lisp standard being produced by X3J13, the ANSI Common Lisp working group.

**Q:** What are the similarities and differences between Flavors and CLOS?

**A:** Both CLOS and Flavors are object-oriented languages built on Lisp. Symbolics participated in the design of CLOS, and thus many of the strengths and features of Flavors are also present in CLOS. CLOS includes extra functionality as well. CLOS does not support most Flavors functionality that is available solely for compatibility with Old Flavors. The rest of this document discusses the similarities and differences between Flavors and CLOS in some detail.

**Q:** How can I find out about CLOS?

**A:** CLOS is defined by its specification, which is:

"Common Lisp Object System Specification," X3J13 Document 88-002R, June 1988. Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon.

It is available in:

SIGPlan Notices (ISSN 0362-1340)
Volume 23
Special Issue -- September 1988

ISBN 0-89791-289-6

Price: Members $10.50, Nonmembers $14.00.
ACM Order Number: 548883

Additional copies may be ordered prepaid from:
ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

For a tutorial book on how to use CLOS, see "Object-Oriented Programming in COMMON LISP" by Sonya E. Keene, copublished by Symbolics Press and Addison-Wesley, 1989.

**Q:** What are Symbolics' intentions with respect to CLOS?

**A:** Symbolics will support CLOS for both Genera and CLOE. This will be a high-quality implementation that offers good performance and full coverage of the CLOS specification Chapters 1 and 2. Note that Chapters 1 and 2 define the CLOS Programmer's Interface, which is the part of CLOS that has been accepted by X3J13 as a standard part of Common Lisp.

**Q:** Will existing programs that use Flavors continue to work?

**A:** Yes. The introduction of CLOS will be an additive process and thus will not entail changes to Flavors. Existing Flavors programs will continue to work.

**Q:** What will Symbolics do to help Flavors users make the transition to CLOS?

**A:** Symbolics is working on a number of ways of helping you make the transition. Symbolics will continue to support the existing Flavors system. Whereas you may want to start writing new code in CLOS, there should be no necessity to convert existing code, because Flavors code will continue to run.

We have prepared this document, "Getting Ready for CLOS", which describes how CLOS differs from Flavors. This document will help you plan for CLOS, by helping you understand differences between CLOS and Flavors that will affect the way you develop code. We expect that some users will be developing Flavors code prior to CLOS availability with the intention of translating it to CLOS; this document will help those users plan for the conversion in advance.

Symbolics will provide some tools to aid in the translation of Flavors code to CLOS.

**Q:** What kind of documentation will you offer for CLOS?

**A:** CLOS will come with complete reference documentation and a copy of Sonya Keene's book, "Object-Oriented Programming in COMMON LISP."

**Q**: What kind of training will you offer for CLOS?

**A**: Symbolics Education Services is planning two courses about CLOS. The first will be a lecture-only seminar for Symbolics users who are experienced programmers in Flavors and would like to learn CLOS very quickly. The plan is to offer it as a day-long program in major cities throughout the country. The second course will be a three-to-four day lecture and laboratory course for experienced Common Lisp programmers who would like to learn object-oriented programming in CLOS.

**Q:** How will CLOS be integrated into the system?

**A:** The integration will occur in two phases.

The first release will add CLOS support without disturbing the existing Flavors support. Only very limited integration between flavors and classes will be provided.

The second release will focus on deeper integration with existing software, and will make upward-compatible changes to existing software to improve the integration.

**Q:** Will Symbolics support the Meta-Object protocol as defined by Chapter 3 of the specification?

**A:** Not in the first release, which will support only the Programmer's Interface, as defined by Chapters 1 and 2. Some portions of Chapters 1 and 2 that deal with meta-objects might not be fully supported in the first release. Note that Chapters 1 and 2 are already accepted parts of the standard, whereas Chapter 3 is not; the Meta-Object Protocol is continuing to evolve.

**Q:** Is CLOS a product in itself? Can I order it?

**A:** CLOS will be a standard part of Genera and CLOE. There will be no additional charge for CLOS. Customers with maintenance contracts will automatically get an update containing it.

**Q:** Will Symbolics CLOS be compatible with PCL?

**A:** Symbolics CLOS will conform to Chapters 1 and 2 of the CLOS specification. It will not aim to be compatible with other implementations, such as PCL. In some cases, these implementations deviate from the CLOS specification. Code that conforms to the CLOS specification and runs on PCL should run on Symbolics CLOS. Code developed to run on PCL without regard to the CLOS specification could depend on aspects of PCL that do not conform to the CLOS specification and may require some modifications in order to run on Symbolics CLOS.

**Q:** Will Symbolics be translating Genera to CLOS?

**A:** No. Flavors support will remain, and the portions of Genera that use Flavors will continue to be implemented in Flavors. Symbolics will not introduce gratuitous incompatibilities into Genera by recoding existing capabilities. Instead, Symbolics will provide good integration between CLOS and Flavors so that new code written in CLOS can coexist with Flavors. Symbolics expects that this support for existing code will aid customers who have a significant amount of existing Flavors code. In the future, Symbolics will decide on a case-by-case basis whether new software projects use Flavors or CLOS. We expect our users will do the same.

**Q:** Is Symbolics looking for Beta test sites?

**A:** Yes. We would like to sign up a small number of good Beta test sites to test the CLOS product. We are looking for Beta sites that would give the system a thorough workout, but which would not have CLOS functionality on their critical path.

### CLOS and Flavors Terminology

Here we translate between Flavors and CLOS terminology describing the basic elements of object-oriented programming:

| Flavors | CLOS |
|---|---|
| flavor | class |
| component flavor | superclass |
| dependent flavor | subclass |
| local component flavor | direct superclass |
| local dependent flavor | direct subclass |
| generic function | generic function |
| method | method |
| combined method | effective method |
| method option | method qualifier |
| instance | instance |
| instance variable | slot |
| ordering of flavor components | class precedence list |

### Comparison of CLOS and Flavors Features

This section discusses features that CLOS and Flavors have in common, Flavors features that CLOS lacks, and CLOS features that Flavors lacks. In this section, CLOS terminology is used. This is a high-level overview; more detailed information is given throughout this document.

### Features provided by both CLOS and Flavors

- Users can define classes, methods, and generic functions.
- Users can create and initialize instances.
- Users can access slots with accessor generic functions.
- A set of built-in method combination types is available.
- Users can define new method combination types.
- Multiple inheritance is supported; this is the ability to define a class built on more than one direct superclass.
- The precedence order of classes is computed based on constraints set locally in the definition of each class.
- Generic functions are called with the same syntax as ordinary Lisp functions.
- Users can redefine classes, methods, and generic functions.
- Users can change the class of an instance.

### Features provided by Flavors but not CLOS

- Wrappers.
- Automatic lexical access to slots by using variables within methods. (CLOS offers a special mechanism for achieving the same effect, but you must explicitly use the **with-slots** macro to get the desired lexical context.)
- Internal flavor functions, macros, and substs.
- Automatically generated constructors.
- Several **defflavor** options related to defining the intended use or protocol of a flavor, such as **:required-methods**, **:abstract-flavor**, **:mixture**, and others.
- The **send** function for sending messages.

### Features provided by CLOS but not Flavors

- The ability to define a method that specializes more than one of its required arguments. For example, one method is applicable if its first argument is a ship, and its second argument is a plane.

- The ability to define a method that specializes on an individual Lisp object, based on an **eql** test. For example, one method is applicable if its third argument is the symbol **danger**; another method is applicable if its first argument is the number **0**.

- The ability to define a method that specializes on Common Lisp types such as **symbol**, **integer**, **list**, and other useful types.

- The ability to define a method that specializes on **defstruct** types. In CLOS, **defstruct** defines a class, and methods can specialize on those classes.

- Class slots, which are slots whose values are shared by all instances of a class; these are used to store information that pertains to a class as a whole, or to all instances of the class.

These CLOS features extend the Flavors paradigm in significant ways, especially the ability to define a method that specializes more than one of its arguments. In Flavors, a method is linked with a single flavor, so it makes sense to say "a flavor inherits methods from its component flavors". In CLOS, a method is linked to a set of classes (because any of the required arguments can be specialized). One ramification of this extension is that in CLOS, there is no concept of "self" in a method body, because no single argument is distinguished. Also, since there is no "self", CLOS provides no automatic access to the instance variables of an object.

The CLOS concept of *applicable methods* is an extension of the Flavors concept of a flavor inheriting methods. A CLOS method has a lambda-list, which states the applicability requirements of the method; in other words, it states the set of arguments for which this method might be invoked. When a CLOS generic function is called, the set of applicable methods is identified, and these methods are executed according to the rules of the method-combination type. An applicable method is a method whose lambda-list requirements are satisfied by the arguments to the generic function.

CLOS classes are tightly integrated with the Common Lisp type system. As mentioned above, CLOS defines classes associated with several useful Common Lisp types (but not every type); you can define methods that specialize on these classes. Also, in CLOS every Lisp object is an instance of a class, so the term "instance" is generalized in CLOS. For example, the object **3** is an instance of the class **integer** and the object **"hello"** is an instance of the class **string**. Defstruct structures are instances of classes.

### CLOS Operators Analogous to Flavors Operators

The Flavors operators (functions, macros, special forms, and variables) are summarized in the Flavors documentation: See the section "Summary of Flavor Functions and Variables".

This section lists each Flavors operator, and shows the analogous CLOS operator, if there is one. In some cases, the concept of the Flavors operator is not applicable

in the CLOS paradigm. In other cases, we will offer a Symbolics CLOS (S-CLOS) extension that offers the same functionality as the Flavors operator.

Note that even where the names of operators are the same in Flavors and CLOS, the syntax and semantics usually differ to some extent.

### Basic Use of Flavors

| Flavors | CLOS |
|---|---|
| **defflavor** | **defclass** |
| **make-instance** | **make-instance** |
| **defgeneric** | **defgeneric** |
| **defmethod** | **defmethod** |
| **compile-flavor-methods** | S-CLOS extension |

There are syntactic and semantic differences between **defflavor** and **defclass**, and between the CLOS and Flavors versions of **make-instance**, **defmethod**, and **defgeneric**. For more information:

See the section "Differences in Defining Classes and Flavors".

See the section "Differences in Making CLOS and Flavors Instances".

See the section "Differences in Defining CLOS and Flavors Generic Functions".

See the section "Differences in Defining CLOS and Flavors Methods".

### Redefining Flavors, Instances, and Operations

| Flavors | CLOS |
|---|---|
| **change-instance-flavor** | **change-class** |
| **recompile-flavor** | *not applicable* |
| **flavor:remove-flavor** | *none* |
| **flavor:rename-instance-variable** | *none* |
| **flavor:transform-instance** | **update-instance-for-different-class** or **update-instance-for-redefined-class** |

In CLOS, there is no need to recompile a class, so **recompile-flavor** is not applicable in CLOS.

In CLOS, there is no operator for removing a class, but you can break the association between the class and its name.

The **flavor:transform-instance** generic function is called in two cases: when a flavor is redefined, and when the class of an instance is changed. You can write a method for **flavor:transform-instance** to control what happens in those cases. CLOS divides those two cases into separate generic functions: **update-instance-for-redefined-class** (called when a new **defclass** form is evaluated to change the definition of a class), and **update-instance-for-different-class** (called when **change-class** is used to change the class of an instance).

### Method Combination

| Flavors | CLOS |
|---|---|
| **define-simple-method-combination** | **define-method-combination** |
| **define-method-combination** | **define-method-combination** |
| **flavor:call-component-method** | **call-method** |
| **flavor:call-component-methods** | *none* |
| **flavor:multiple-value-prog2** | *none* |
| **flavor:method-options** | **method-qualifiers** |

The CLOS **define-method-combination** macro has a short form, which is similar to Flavors **define-simple-method-combination**, and it has a long form, which is similar to Flavors **define-method-combination**.

The syntax of defining a new method-combination type is different in CLOS and Flavors, but you should be able to use the CLOS operators to define any method-combination type that you defined using the Flavors tools.

There are semantic differences in the default method combination types, Flavors **:daemon**, and CLOS **standard**. See the section "Differences in CLOS and Flavors Default Method Combination".

There are also differences in the built-in method combination types: See the section "Differences in CLOS and Flavors Built-in Method Combination Types".

### Internal Functions of Flavors

| Flavors | CLOS |
|---|---|
| **defun-in-flavor** | *none* |
| **defmacro-in-flavor** | *none* |
| **defsubst-in-flavor** | *none* |

CLOS does not support internal flavor functions, macros, or substs.

### Wrappers and Whoppers

| Flavors | CLOS |
|---|---|
| **defwrapper** | *none* |
| **defwhopper** | around-methods |
| **continue-whopper** | **call-next-method** |
| **lexpr-continue-whopper** | **apply** of **call-next-method** |
| **defwhopper-subst** | *none* |

CLOS does not support wrappers. CLOS does support around-methods, which can be used in much the same way as whoppers.

### Variables

| Flavors | CLOS |
|---|---|
| **sys:*all-flavor-names*** | *none* |
| **flavor:*flavor-compile-trace-list*** | *none* |
| **self** | *not applicable* |

The first two variables are in the category of environmental tools, which CLOS does not support. They are not variables that would be used in application programs, so they should not affect Flavors users who want to convert code to CLOS.

The variable **self** is not part of CLOS because of a major difference in the design of CLOS and Flavors. In Flavors, each method is associated with a single flavor, and **self** can be used to refer to an instance of that flavor within the body of the method. In CLOS, a method can be associated with more than one class (or with an individual Lisp object instead of a class), so the concept of "self" is not appropriate to CLOS. In CLOS, you can refer to the instances for which the method is applicable by using the variable names given in the lambda-list of the method.

### Vanilla Flavor and Its Operations

| Flavors | CLOS |
|---|---|
| **flavor:vanilla** | **standard-object** |
| **:describe** | **describe-object** |
| **sys:print-self** | **print-object** |
| **:print-self** | **print-object** |
| **:send-if-handles** | *none* |
| **:which-operations** | *none* |
| **:operation-handled-p** | *none* |
| **operation-handled-p** | *none* |
| **get-handler-for** | *none* |

Both Flavors and CLOS define a class that is included in the definition of each user-defined class to provide default behavior. Flavors has **flavor:vanilla** and CLOS has a class called **standard-object**. (CLOS also specifies that the class **t** is included in the definition of all classes, whether user-defined or not.) The CLOS class **standard-object** provides methods for **describe-object** and **print-object**.

### Message-Passing

| Flavors | CLOS |
|---|---|
| **send** | *none* |
| **lexpr-send** | *none* |
| **send-if-handles** | *none* |
| **lexpr-send-if-handles** | *none* |

CLOS provides no support for message-passing. CLOS generic functions are called in the same way that ordinary functions are called.

The Flavors message-passing tools are available for compatibility with Old Flavors. Since some portions of Genera use message-passing (streams and some of the windows code, for example), it is important that **send** and related operations continue to be supported by Flavors.

### A Flavor's Handler for an Operation

| Flavors | CLOS |
|---|---|
| **flavor:compose-handler** | *none* |
| **flavor:compose-handler-source** | *none* |
| **get-handler-for** | *none* |

These functions are in the category of environmental tools, which the CLOS Programmer Interface does not support.

### A Flavor's default-init-plist

| Flavors | CLOS |
|---|---|
| **flavor:flavor-default-init-get** | *none* |
| **flavor:flavor-default-init-putprop** | *none* |
| **flavor:flavor-default-init-remprop** | *none* |

These functions are in the category of environmental tools, which the CLOS Programmer Interface does not support.

### A Flavor's Instance Variables

| Flavors | CLOS |
|---|---|
| **symbol-value-in-instance** | **slot-value** |
| **boundp-in-instance** | **slot-boundp** |

### Getting Other Information on Flavors

| Flavors | CLOS |
|---|---|
| **flavor:find-flavor** | **find-class** |
| **flavor:flavor-allowed-init-keywords** | *none* |
| **flavor:flavor-allows-init-keyword-p** | *none* |
| **flavor:get-all-flavor-components** | *none* |

The three operations that CLOS does not support fall into the category of environmental tools.

### Other Flavors Tools

| Flavors | CLOS |
|---|---|
| **sys:debug-instance** | *none* |
| **flavor:describe-instance** | S-CLOS extension |
| **sys:eval-in-instance** | *none* |
| **flavor:generic** | *not applicable* |
| **zl:get-flavor-handler-for** | *none* |
| **instancep** | **(typep** *x* **'standard-object)** |
| **zl:locate-in-instance** | S-CLOS extension |
| **flavor:print-flavor-compile-trace** | *none* |
| **sys:property-list-mixin** | *none* |
| **zl:set-in-instance** | **setf** of **slot-value** |
| **zl:symeval-in-instance** | **slot-value** |
| **:unclaimed-message** | **no-applicable-method** |
| **flavor:with-instance-environment** | **with-slots** |

The **flavor:generic** special form is used with the :**function** option to **defgeneric**; since CLOS does not support the **:function** option, it does not support **flavor:generic**.

In CLOS every Lisp object is an instance of some class, so an "**instancep**" function is not needed. However, in CLOS the form **(typep** *x* **'standard-object)** answers the same question that Flavors **instancep** does, namely: is this object an instance of a user-defined class?

CLOS does not define the **sys:property-list-mixin** class, but you can define such a class yourself.

### Differences in Defining Classes and Flavors

This section describes differences between Flavors **defflavor** and CLOS **defclass**. At the end of this section, we give an example of translating a Flavors **defflavor** form into a CLOS **defclass** form.

The syntax of **defclass** is:

```
defclass class-name superclasses slot-specifiers
        &rest class-options
```

A *slot-specifier* is one of:

*slot-name*
(*slot-name slot-options*...)

The CLOS **defclass** macro has two kinds of options: *slot options*, which pertain to a single slot; and *class options*, which pertain to the class as a whole. In Flavors this distinction is not as rigid, and many options (especially those that affect instance variables) can be made to refer to all instance variables, or a set of them. In CLOS, slot options must be given individually for each slot; there is no abbreviation for specifying that a slot option affects all slots. In the same vein, CLOS

does not have default names for functions (such as reader or writer generic functions) or initialization arguments.

In Flavors, you can give a default initial value form for an instance variable by giving a list containing the name and the initial value form. In CLOS, you use the **:initform** slot option to provide a default initial value form for a slot.

Here are the options to Flavors **defflavor**, and the analogous options to CLOS **defclass**. Note that CLOS enables implementations to extend **defclass** to support additional options, and Symbolics CLOS might choose to do so. The presentation of this information is based on a section of Flavors documentation. See the section "Summary of **defflavor** Options".

**Frequently Used Options to defflavor**

**:initable-instance-variables**
: The **defclass :initarg** slot option is similar. In CLOS, there is no shortcut way of indicating that all instance variables are initable, nor is there a default name for the initialization argument (in Flavors, the default is the keyword with the same name as the instance variable). In CLOS, you specify the **:initarg** slot option for each slot that should be initable, and you provide the symbol to be used as an argument to **make-instance**.

**:readable-instance-variables**
: The **defclass :reader** slot option is similar. In CLOS, there is no shortcut way of indicating that all instance variables are readable, nor is there a default name for the reader generic function (in Flavors, the default is *flavor-variable*). In CLOS, you specify the **:reader** slot option for each slot that should have a reader generic function, and you provide the name of the reader.

**:writable-instance-variables**
: The **defclass :accessor** slot option is similar, in that it defines both a reader generic function (you provide the symbol that names the reader) and a **setf** generic function that can be used to write the value of the slot. CLOS also offers another slot option, called **:writer**, which defines a writer generic function without also creating a reader. You provide the name of the writer generic function.

**:locatable-instance-variables**
: No analogous capability in CLOS, but an S-CLOS extension will be provided.

**:conc-name**
: No analogous option in CLOS **defclass**. The name of each reader and writer generic function must be provided explicitly.

**:constructor**
: No analogous option in CLOS **defclass**. You can define a constructor by hand, however, as an ordinary function that calls **make-instance**.

**:init-keywords**
: No analogous option in CLOS **defclass**. In CLOS, there is no need to de-

clare init-keywords that are used by initialization methods. A background note: in Flavors, you can control initialization by writing methods for **make-instance**, whereas in CLOS, the analogous way to control initialization is to write methods for **initialize-instance**, which is called by **make-instance**. In CLOS, any keyword argument accepted by an initialization method is automatically a valid argument to **make-instance**.

**:default-init-plist**
The **defclass :default-initargs** class option is similar.

**:required-instance-variables**
No analogous option in CLOS **defclass**.

**:required-init-keywords**
No analogous option in CLOS **defclass**.

**:required-methods**
No analogous option in CLOS **defclass**.

**:required-flavors**
No analogous option in CLOS **defclass**.

**:method-combination**
No analogous option in CLOS **defclass**. In CLOS (as in Flavors), there is a **:method-combination** option to **defgeneric**.

**Less-Frequently Used defflavor Options**

**:functions**
No analogous option in CLOS **defclass**. CLOS has no functions internal to a flavor.

**:mixture** No analogous option in CLOS **defclass**. To define a family of related classes, you must define each class with **defclass**, or define a macro to do so.

**:abstract-flavor**
No analogous option in CLOS **defclass**.

**:component-order**
No analogous option in CLOS **defclass**.

**:documentation**
The **defclass :documentation** class option is similar; it enables you to provide a documentation string associated with the class. In addition, CLOS has a **:documentation** slot option, which enables you to provide a documentation string associated with a slot.

**:area-keyword**
No analogous option in CLOS **defclass**, although there might be an S-CLOS extension.

**:no-vanilla-flavor**
No analogous option in CLOS **defclass**.

**:ordered-instance-variables**
No analogous option in CLOS **defclass**.

**:method-order**
No analogous option in CLOS **defclass**.

**Options Intended for System Internals**

**:special-instance-variables**
No analogous option in CLOS **defclass**.

**:special-instance-variables-binding-methods**
No analogous option in CLOS **defclass**.

**Options for Old Flavors Compatibility**

**:gettable-instance-variables**
The **defclass :reader** slot option is similar. See the discussion on the **:readable-instance-variables defflavor** option above.

**:settable-instance-variables**
The **defclass :accessor** slot option is similar. See the discussion on the **:writable-instance-variables defflavor** option above.

**:default-handler**
No analogous option in CLOS **defclass**, but you can achieve the same effect by defining a method on the CLOS generic function named **no-applicable-method**.

## Examples

This example comes from the Flavors documentation. See the section "Example of Programming with Flavors: Life".

```
;;; Flavors
(defflavor cell (x y status next-status neighbors) ()
  (:documentation "Functional unit of the Life game.")
  (:readable-instance-variables status)
  (:initable-instance-variables x y status))

;;; CLOS
(defclass cell ()
      ((x :initarg :x)
       (y :initarg :y)
       (status :initarg :status :reader cell-status)
       neighbors)
  (:documentation "Functional unit of the Life game."))
```

In **defclass**, the order of the list of direct superclasses and the list of slots is the reverse of **defflavor**. After the name of the class comes the list of direct superclasses, followed by the list of slots.

You can see how the syntax of **defclass** ensures that slot options (such as **:initarg** and **:reader**) are directly associated with a slot, whereas class options (such as **:documentation**) are associated with the class as a whole.

Also notice that **:initarg** and **:reader** have no defaults, so you must provide the symbol that initializes the slot and the symbol that names the reader generic function on a per-slot basis.

### Differences in Defining CLOS and Flavors Methods

At the end of this section, we give examples of translating Flavors **defmethod** forms into CLOS **defmethod** forms.

In Flavors, a generic function dispatches (selects and combines methods) based on the flavor of its first argument. In other words, the method specializes on the flavor of the first argument to the generic function.

The CLOS generic-function dispatching mechanism is more general:

- A CLOS method can specialize any number of its required parameters. The first argument to the generic function is not treated specially. This makes the generic dispatch mechanism somewhat more complicated, but it is a more flexible model than that of Flavors.

- A CLOS method can specialize on an individual Lisp object, as well as on a class. The method is applicable if the corresponding argument to the generic function is **eql** to the object.

- CLOS defines classes corresponding to many Common Lisp types, such as **symbol**, **list**, **integer**, and so on. The purpose is to allow CLOS methods to specialize on these classes.

- In CLOS, **defstruct** defines a class (unless the **:type** option is supplied). A CLOS method can specialize on a class defined by **defstruct**.

In general, the CLOS extended view of methods should not require effort when converting Flavors methods to CLOS. We mention the exceptions below, along with other differences that affect the translation of Flavors methods to CLOS.

- CLOS method bodies have no "self" or direct access to slots.

  Since each Flavors method is closely allied with a flavor, there are two convenient shortcuts that Flavors users are accustomed to having in the bodies of methods. First, the instance is accessible by the variable **self**. Second, the instance variables are accessible by name for reading and writing.

  In CLOS, there is no one-to-one correspondence between a method and a class. A method might be specialized on an individual Lisp object such as a number, or it might be specialized on more than one argument. Thus, the concept of

"self" is not relevant to CLOS. Instead, you can refer to each argument of the generic function in the usual Lisp way, by using a variable that names that parameter. Also, the slots are not automatically accessible by name. You can access them by calling accessor functions (readers or writers). Or, you can use **with-slots**; within the body of **with-slots** you can access slots by their names or by other variable names that you assign them. Similarly, **with-accessors** enables you to use a shortcut syntax to call the accessor functions by using variable names that you assign.

- CLOS requires a method qualifier when the simple non-**standard** method combination types are used.

  In CLOS, if a generic function uses a built-in method-combination type (other than **standard**) or one defined by the short form of **define-method-combination**, then all primary methods for that generic function must have a method qualifier (a symbol that is the name of the method-combination type). In Flavors, the primary methods can be unqualified, or can be qualified with the keyword with the same name as the method combination type.

  This requirement does not necessarily hold for method-combination types defined by the long form of the CLOS **define-method-combination** macro.

- CLOS defines different rules for congruence of lambda-lists for methods of a generic function.

  We discuss this topic separately: See the section "Differences in CLOS and Flavors Lambda-list Congruence Rules".

### Examples

```
;;; Flavors
(defmethod (aliveness cell) ()
  (if (eq status ':alive) 1 0))

;;; CLOS
(defmethod aliveness ((c cell))
  (with-slots (status) c
    (if (eq status ':alive) 1 0)))
```

In the Flavors method, the instance variable **status** is accessible by name. The CLOS method uses **with-slots** to make the slot accessible by the variable named **status**.

The CLOS **defmethod** syntax is more general than that of Flavors, as shown in the examples below. Briefly, we show some CLOS **defmethod** forms that specialize on more than one argument, and that specialize on individual Lisp objects.

```
;;; Applicable when first arg is a ship, second arg is a plane
(defmethod collide ((s ship) (p plane) location)
  body)

;;; Applicable when first arg is a plane, second arg is a plane
(defmethod collide ((p plane) (p plane) location)
  body)

;;; Applicable when second arg is a plane
(defmethod collide (vehicle (p plane) location)
  body)

;;; Applicable when first arg is eql to the value of *Enterprise*
(defmethod collide ((ent (eql *Enterprise*)) vehicle location)
  body)
```

The form ***Enterprise*** is evaluated once, when the method is defined; it is not evaluated when the generic function is called.

The required parameters that appear as variable names (not as lists) do not place any restrictions on the applicability of the method.

### Differences in Defining CLOS and Flavors Generic Functions

This section describes differences between Flavors **defgeneric** and CLOS **defgeneric**. At the end of this section, we give an example of translating a Flavors **defgeneric** form into a CLOS **defgeneric** form.

Here are the options to Flavors **defgeneric**, and the analogous options to CLOS **defgeneric**. Note that CLOS enables implementations to extend **defgeneric** to support additional options, and Symbolics CLOS might choose to do so. The presentation of this information is based on a section of Flavors documentation: See the section "Options for **defgeneric**".

**:compatible-message**
No analogous option in CLOS **defgeneric**.

**declare** The CLOS **declare** option is similar; it enables you to provide the **optimize** declaration, and to indicate that speed or space should be optimized. Note that in CLOS, the **declare** option must support **optimize**, but is not required to support additional declarations. Symbolics CLOS will support additional declarations such as those recognized by the Flavors **declare** option, **arglist**, **values**, **sys:downward-funarg**, and **sys:function-parent**.

:**dispatch** This Flavors option to **defgeneric** enables you to specify which argument to the generic function should be the basis of method selection. CLOS has no need for this option, because its design is based on the premise that methods can be specialized on any one or more of the required arguments to the generic function. In CLOS, the methods state which argument(s) should be the basis of method selection.

**:documentation**
The CLOS **:documentation** option is similar.

**:function** No analogous option in CLOS **defgeneric**.

**:inline-methods**
No analogous option in CLOS **defgeneric**.

**:method** The CLOS **:method** option is similar.

**:method-arglist**
No analogous option in CLOS **defgeneric**.

**:method-combination**
The CLOS **:method-combination** option is similar.

**:optimize** No analogous option in CLOS **defgeneric**. Note that this Flavors option is intended for use with the **:dispatch** option, which CLOS does not support.

### Example

The syntax of **defgeneric** is not much different in CLOS and Flavors.

```
;;; Flavors
(defgeneric aliveness (cell-unit)
  "Returns 1 if the cell-unit is currently alive, 0 otherwise.")

;;; CLOS
(defgeneric aliveness (cell-unit)
  (:documentation
"Returns 1 if the cell-unit is currently alive, 0 otherwise."))
```

### Differences in CLOS and Flavors Lambda-list Congruence Rules

When a generic function is called, any or all of the methods for that generic function might be called. In general, the arguments given to the generic function are passed to each method that is called (there are ways to modify this in both CLOS and Flavors). Thus, it is important that the methods that might be called accept the arguments that can be given to a generic function. In addition, if there is a **defgeneric** form, then its lambda-list must also be congruent with the methods. Both Flavors and CLOS have rules that describe the congruence of lambda-lists of a generic function and its methods; however, the rules are different.

In both Flavors and CLOS, the lambda-lists of the generic function and all its methods must specify the same number of required parameters. CLOS has an extra requirement that the lambda-lists must accept the same number of optional parameters. In CLOS, if one lambda-list uses &rest or &key, then all lambda-lists must use one or the other.

Another difference in the congruence rules lies in the treatment of keyword parameters. In Flavors, the programmer is responsible for ensuring that all the methods for a generic function accept the same set of keywords, either by naming them all explicitly in each lambda-list with &key, or by using &allow-other-keys.

Note that in CLOS, the set of applicable methods controls which keywords are accepted in a given generic function call. If a CLOS generic function is called with a keyword argument, and that keyword is not accepted by one of the applicable methods, then an error is signaled. In CLOS, no error is signaled if at least one applicable method accepts the keyword. In Flavors, an error is signaled unless every applicable method that is actually called accepts the keyword.

In Flavors, if &allow-other-keys appears in any applicable method, then the keyword argument checking is disabled for that method only. In CLOS, if &allow-other-keys appears in any applicable method, then the keyword argument checking is disabled for the generic function. Therefore, &allow-other-keys should be used less frequently in CLOS than in Flavors.

In CLOS, if a **defgeneric** form uses &key, then each of those keywords must be accepted by all methods for the generic function. The methods can accept those keywords by naming them explicitly in &key, or by using &allow-other-keys, or by using &rest but not &key. Each method may accept additional keyword arguments, and need not accept the keyword arguments accepted by other methods. If the **defgeneric** form or any applicable method uses &allow-other-keys, then any keyword argument can be supplied in the generic function call.

### Examples

In the following example, the Flavors primary method specifies two optional arguments, and the after-method uses &rest instead of specifying the optional arguments:

```
;;; Flavors
(defmethod (refresh basic-window) (arg1 &optional arg2 arg3)
  body)

(defmethod (refresh window-with-border :after) (arg1 &rest ignore)
  body)
```

To translate the Flavors methods into CLOS, it is necessary that the after-method specifies the two optional arguments:

```
;;; CLOS
(defmethod refresh ((w basic-window) arg1 &optional arg2 arg3)
  body)

(defmethod refresh :after ((w window-with-border) arg1 &optional arg2 arg3)
  body)
```

Suppose we have a generic function **f** which takes an optional argument *o* whose default value varies according to the class:

```
;;; CLOS
(defgeneric f (object &optional o))
```

```
(defmethod f ((object a) &optional (o 1))
  body)

(defmethod f ((object b) &optional (o 2))
  body)
```

Suppose we want to count the number of times **f** is applied to an object. One way of doing this is to make an encapsulation object:

```
;;; CLOS
(defclass encapsulation
          ()
          ((encapsulate :initarg encapsulate)
           (count :initform 0)))
```

You might try to define the following CLOS method, but the parameters to the method are not congruent to those of the generic function, so this cannot be done:

```
;;; Incorrect use of CLOS
(defmethod f ((object encapsulation) &rest rest)
  (with-slots (encapsulate count) object
    (incf count)
    (apply #'f encapsulate rest)))
```

Instead, you can define the following method:

```
;;; CLOS
(defmethod f ((object encapsulation) &optional (o nil o-supplied-p))
  (with-slots (encapsulate count) object
    (incf count)
    (if o-supplied-p
       (f encapsulate o)
       (f encapsulate))))
```

### Differences in CLOS and Flavors setf Generic Functions

The syntax of defining **setf** generic functions and methods is different in CLOS and Flavors.

```
;;; Flavors
(defgeneric (setf symbol) (instance args... new-value)
  options...)

(defmethod ((setf symbol) flavor) (args... new-value)
  body)

;;; CLOS
(defgeneric (setf symbol) (new-value instance args...)
  options...)
```

```
(defmethod (setf symbol) (new-value (instance class) args...)
  body)
```

The Flavors lambda-lists have the *new-value* parameter last, preceded by other arguments. The CLOS lambda-lists have the *new-value* parameter first, followed by other arguments.

In CLOS, any required parameter in the **setf** method's lambda-list may be specialized, including the *new-value* parameter.

### Differences in Making CLOS and Flavors Instances

CLOS has a powerful and flexible mechanism for creating and initializing instances. It offers most of the same features that Flavors does in this area, although the details of how you write code to use the features is different.

We discuss the differences between CLOS and Flavors in creating and initializing instances below:

- Flavors offers a :**constructor** option to **defflavor**, but CLOS has no similar option in **defclass**. You can define a constructor in CLOS as an ordinary function that calls **make-instance**.

- The CLOS **make-instance** generic function has a procedural definition. This means that **make-instance** is defined to call a set of generic functions, and users can customize various aspects of the procedure by specializing one or more of these generic functions. In CLOS, aspects of **make-instance** are available to be controlled both at the application level and at the meta-object level.

  In Flavors, users can customize the initialization of instances by specializing **make-instance**. In CLOS, users can customize the initialization of instances by specializing **initialize-instance**, a generic function that is called by **make-instance**. (The Old Flavors way of doing this was to write methods for the **:init** message; this is not supported in CLOS.)

  CLOS users can also control initialization by specializing **shared-initialize**, which controls initialization at object creation time, and in other contexts, such as redefining a class.

- In Flavors, if methods for **make-instance** accept arguments other than arguments that initialize instance variables, then the **defflavor** form must include the **:init-keywords** option to declare those arguments as valid. In CLOS, any arguments accepted by an applicable method for **initialize-instance** or **shared-initialize** are automatically declared as valid.

### Example

This example shows how a Flavors method for **make-instance** can be translated into a CLOS method for **initialize-instance**.

```
;;; Flavors
(defmethod (make-instance box-with-cell) (&rest ignore)
  (setq box-x-center (round (+ box-x (* .5 *side-length*))))
  (setq box-y-center (round (+ box-y (* .5 *side-length*)))))

;;; CLOS
(defmethod initialize-instance :after ((bwc box-with-cell) &key)
  (with-slots (box-x-center box-y-center box-x box-y) bwc
    (setq box-x-center (round (+ box-x (* .5 *side-length*))))
    (setq box-y-center (round (+ box-y (* .5 *side-length*))))))
```

### Differences in CLOS and Flavors Default Method Combination

The default method combination type is called **:daemon** in Flavors and **standard** in CLOS. They have the following similarities:

- Primary methods are recognized by the lack of a method qualifier.
- In addition to primary methods, before-methods and after-methods are supported. In CLOS, as in Flavors, the qualifiers are **:before** and **:after**.
- The order of calling before-methods, the primary method, and after-methods is the same in Flavors and CLOS.

A Flavors generic function that uses only primary methods, before-methods, and after-methods can be translated in a straightforward syntactic way to a CLOS generic function; the behavior of the generic functions would be the same.

The CLOS **standard** method combination differs from Flavors **:daemon** in the following ways:

- CLOS does not support wrappers or whoppers. However, CLOS supports around-methods, which are similar to whoppers. An around-method is identified by the method qualifier **:around**. In the body of the method, **call-next-method** is used to cause other methods to run. Around-methods are combined in a CLOS generic function in much the same way that whoppers are combined in Flavors.
- In a CLOS generic function, more than one primary method can be executed. CLOS supports the use of **call-next-method** within primary methods; this causes the next most specific method to be executed. Since this is an area where CLOS provides extra functionality, it does not require effort when converting Flavors programs to CLOS.

- CLOS requires a primary method. In CLOS, if a generic function is called and there is no applicable primary method, an error is signaled. In Flavors, in this situation a primary method is assumed that returns **nil**. This difference, though incompatible, should not affect much user code.

- Flavors has **:default** methods, and CLOS does not.

### Differences in CLOS and Flavors Built-in Method Combination Types

CLOS offers a set of built-in method-combination types that are analogous to the "simple" method-combination types offered by Flavors. CLOS does not provide any of the complex Flavors method-combination types, but it does allow them to be defined. We will provide most of these method-combination types as S-CLOS extensions.

| Flavors | CLOS |
|---|---|
| **:daemon** | **standard** |
| **:and** | **and** |
| **:append** | **append** |
| **:case** | **eql** parameter specializers |
| **:daemon-with-and** | S-CLOS extension |
| **:daemon-with-or** | S-CLOS extension |
| **:daemon-with-override** | S-CLOS extension |
| **:inverse-list** | S-CLOS extension |
| **:list** | **list** |
| **:max** | **max** |
| **:min** | **min** |
| **:nconc** | **nconc** |
| **:or** | **or** |
| **:pass-on** | S-CLOS extension |
| **:progn** | **progn** |
| **:sum** | + |
| **:two-pass** | S-CLOS extension |

There are semantic differences in the default method combination types: Flavors **:daemon**, and CLOS **standard**. See the section "Differences in CLOS and Flavors Default Method Combination".

Like Flavors, the CLOS "simple" built-in method-combination types have an *order* argument that enables you to specify **:most-specific-first** (the default) or **:most-specific-last** order of primary methods.

By convention, CLOS uses symbols (not keyword symbols) to name method-combination types.

The CLOS built-in method-combination types (other than **standard**) accept primary methods and around-methods. The CLOS feature of supporting around-methods in generic functions that use built-in method-combination types is similar to the Flavors feature of supporting whoppers.

In CLOS, a primary method for a built-in method-combination type (other than **standard**) must have a method qualifier that is the symbol that names the method-combination type. (This is a difference from Flavors, in which primary methods for built-in method-combination types can unqualified, or they can be qualified by the symbol that names the method-combination type).

### Differences in CLOS and Flavors Class Precedence Order

The CLOS algorithm for determining the class precedence list is similar to the New Flavors algorithm for determining the ordering of flavor components. The two algorithms, however, are not identical. The vast majority of Flavors programs will have the same ordering using the CLOS algorithm.

The algorithms have the following characteristics in common:

- A class must precede its superclasses in the class precedence list.
- Each class definition sets local constraints on the ordering of its direct superclasses. In CLOS (as in Flavors), the order of direct superclasses in the **defclass** form sets constraints that must be followed in the class precedence list. Each class has precedence over the classes that appear after it in the list of direct superclasses.

If a Flavors program depends only on those two rules, then the program will continue to work when converted to CLOS.

The following example illustrates how the two algorithms differ. In these class definitions, the order of **x** and **y** is unconstrained: Flavors resolves it by making two depth-first passes through the tree of Flavors, thus it selects **x** before **y**; CLOS resolves it by choosing the class that has the rightmost direct subclass, which is **y**.

```
(defclass a (b c d e) ())
(defclass b (e x) ())
(defclass c (e y) ())
(defclass d () ())
(defclass e () ())
(defclass x () ())
(defclass y () ())
```

Excluding the classes **flavor:vanilla**, **standard-object**, and **t**, the two class precedence lists for class **a** are:

```
Flavors:   (a b c d e x y)
CLOS:      (a b c d e y x)
```

CLOS does not support the **:component-order** option to **defflavor**, which enables Flavors users to state explicitly the ordering constraints. This Flavors option is used to relax constraints that come from the order of component flavors in the **defflavor** form, or to specify additional constraints.

### Developing CLOS Programs

The first release of CLOS will enable users to write some new programs that use CLOS. The CLOS operators will be in a separate package. Programming tools for CLOS will be available, including features analogous to the Flavor Examiner and the Flavors-related CP and m-X editor commands.

Note that complete integration between Flavors and CLOS will not be provided until the second release of CLOS. This means that some Flavors programs cannot be converted to CLOS until the second release of CLOS. For example, a Flavors program that makes use of flavors defined by Genera (such as window or stream flavors) cannot be converted to CLOS until the integration between CLOS and Flavors is completely supported.