Object-Oriented Programming with *Flavors*

David A. Moon
Symbolics, Inc.

Abstract

This paper describes Symbolics' newly redesigned object-oriented programming system, *Flavors*. *Flavors* encourages program modularity, eases the development of large, complex programs, and provides high efficiency at run time. *Flavors* is integrated into Lisp and the Symbolics program development environment. This paper describes the philosophy and some of the major characteristics of Symbolics' *Flavors* and shows how the above goals are addressed. Full details of *Flavors* are left to the programmers' manual, *Reference Guide to Symbolics Common Lisp. (5)*

History of *Flavors*

The original *Flavors* system was developed by the MIT Lisp Machine group in 1979. *(1, 2, 3)* It was used to build a window system and later applied to other system programming. In 1981, Symbolics designed a more efficient implementation of *Flavors*. Since that time, we have made increasingly heavy use of *Flavors* in nearly every aspect of the Symbolics 3600 software, such as I/O streams, network control programs, the debugger, the editor, and user interface facilities. Flavors permeate both the operating system and higher-level utilities. The same *Flavors* tools used in-house are fully documented and are used by most Symbolics customers to build their application programs.

Five years' experience with *Flavors* has pointed out the strengths and weaknesses of the original design. In 1985 we undertook a thorough redesign of *Flavors* to solve the problems that we had identified. The result is a new *Flavors* system that has been implemented at Symbolics and has been used in-house to develop several complex programs. This newly-designed *Flavors* is the version described in this paper. It will be released with the next Symbolics software release. *(5)*

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0-89791-204-7/86/0900-0001 75¢

What is object-oriented programming?

We view object-oriented programming as a technique for organizing very large programs. This technique makes it practical to deal with programs that would otherwise be impossibly complex.

An object-oriented program consists of a set of objects and a set of operations on those objects. These entities are not defined in a monolithic way. Instead, the definitions of the operations are distributed among the various objects that they can operate upon. At the same time, the definitions of the objects are distributed among the various facets of their behavior. An object-oriented programming system is an organizational framework for combining these distributed definitions and managing the interactions among them.

Object-oriented programming is also an abstraction mechanism. A program that manipulates an object uses certain defined operations to manipulate it. These operations serve as an interface, and the program does not need to know how the object implements the operations. The implementation of one operation can be different for different kinds of objects. At the same time, an object's behavior can be divided into several facets, which need not know each other's internal details.

Goals of *Flavors*

There are many possible and useful styles of object-oriented programming. *Flavors* adopts an approach aimed at these goals:

- Encourage program modularity. By this we mean that *Flavors* should make it easier to construct programs out of existing parts, to modify the behavior of existing programs without massively rewriting them, to understand programs one piece at a time, and to identify interfaces between modules.
- Ease development of large, complex programs. In addition to encouraging modularity, *Flavors* should

allow programs to be constructed incrementally, make it possible to change data representations and program organization while the program is running, and provide tools for analyzing programs.

- Provide high efficiency at run-time. The CPU time and page fault rate associated with object-oriented operations should not be very much larger than that associated with ordinary function calling and data manipulation. This must be accomplished without compromising other goals.

- Be compatible with the previous *Flavors* system. By this we mean tools and backward-compatibility features to ease conversion of programs written with the old *Flavors* to run with the new *Flavors*. Compatibility features are not discussed in this paper.

Later sections discuss how these goals are met. We first present the concepts of *Flavors*.

Basic *Flavors* Concepts

It is often convenient to organize programs around *objects*, which model real-world things. Each real-world object is modelled by a single Lisp object. Each object has some *state* and a set of operations that can be performed on it. A *Flavors* program is built around:

Flavors
: Each kind of object is implemented as a *flavor*. A flavor is an abstraction of the characteristics that all objects of this flavor have in common. It is a new aggregate data type. For example, this form defines a flavor that represents ships:

```
(defflavor ship
           (x-velocity y-velocity mass)
           ()
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)
```

Instances of a flavor
: Each object is implemented as an *instance* of a flavor. For example, here we create and initialize an instance of the **ship** flavor:

```
(setq my-ship
      (make-instance 'ship :mass 14
                           :x-velocity 24
                           :y-velocity 2))
```

Instance variables
: This is a set of named variables with separate values for each instance. The values of the instance variables represent the state of each object. The instance variables of the **ship** flavor are **x-velocity**, **y-velocity**, and **mass**. The **:readable-instance-variables** option generates accessor functions for reading the values of instance variables; for example:

```
(ship-mass my-ship)
--> 14
```

Similarly, the **:writable-instance-variables** option allows us to alter the value of an instance variable:

```
(setf (ship-mass my-ship) 100)
--> 100
```

Generic functions
: The operations that are performed on objects are known as *generic functions*.

Methods
: The Lisp function that performs a generic function on instances of a certain flavor is called a *method*. The instance variables are accessible by name inside the body of a method. Often, one generic function has several methods defined for it, attached to different flavors. An example method:

```
(defmethod (ship-speed ship) ()
  (sqrt (+ (expt x-velocity 2)
           (expt y-velocity 2))))
```

Generic Functions

A generic function operates on an object by selecting one or more methods that implement the generic operation in a specialized way for that object. One of the arguments (usually the first) to a generic function is the object: the available methods are those attached to its flavor. Generic functions are the interface between objects. They provide abstraction and isolation between modules.

Figure 1 shows the differences between ordinary Lisp functions and generic functions.

Ordinary Functions	*Generic Functions*
Have a single definition.	Have a distributed definition.
Interface is specified by defun.	Interface can be specified by defgeneric or defmethod.
Do not treat flavor instances specially.	First argument is usually an instance of a flavor.
Implementation is the same whenever the function is called.	Implementation varies from call to call, depending on the flavor of the first argument.

Figure 1

Differences between ordinary and generic Lisp functions.

Generic functions are smoothly integrated into the Lisp environment. Ordinary functions and generic functions are called with the same syntax. Making generic functions syntactically and semantically compatible with ordinary functions has the following advantages:

- The caller of a function need not know whether it is generic.
- The Common Lisp package system *(4)* can be used to isolate modules and to distinguish between public and private interfaces by `exporting` the names of public generic functions.
- Debugging tools such as `trace` can be used on generic functions.
- Program-development tools that get the argument list or documentation of a function work equally well on generic functions.

Mixing Flavors

A typical flavor is defined by combining several other flavors, called its components. The new flavor inherits instance variables, methods, and additional component flavors from its components. In a well-organized program, each component flavor is a module that defines a single facet of behavior. When two types of objects have some behavior in common, they each inherit it from the same flavor, rather than duplicating the code. When flavors are mixed together, *Flavors* organizes and manages the interactions between them. This multiple inheritance is a key aspect of the design of *Flavors*; the mechanism is described in the following sections.

Ordering Flavor Components

When a flavor is defined with one or more component flavors, *Flavors* chooses an *ordering* of its components, including both the direct components and the inherited components. Components at the beginning of the ordering are the most specific and those at the end are the most general. The ordering is important because it controls how a flavor's methods are inherited from components and how components earlier in the ordering specialize the behavior of those later in the ordering. The details of method inheritance are explained in a later section, but first the way the ordering is chosen must be understood.

Each flavor defines certain constraints on the ordering of itself and its direct components. Taken together, these constraints determine a partial ordering of all of the components of a flavor. *Flavors* computes a total ordering that is consistent with the partial ordering. Three rules control the ordering of flavor components:

- *A flavor always precedes its own components.*
- *The local ordering of components of a flavor is preserved.* This is the order of components given in the `defflavor` form.
- *Duplicate flavors are eliminated from the ordering.* If a flavor appears more than once, it is placed as close to the beginning of the ordering as possible, while still obeying the other rules.

The goal of ordering flavor components is to preserve the modularity of programs. A flavor should be treated as an intact unit, with well-defined characteristics and behavior; it is essential that mixing flavors together does not alter the internal details of any of the component flavors. This makes it easier to assemble a program from pieces by combining pre-existing flavors. The rules for ordering components support this by ensuring that a flavor's components will be in the same order when that flavor is part of another as they are when it stands alone.

Here is an example of ordering components. The third subform of `defflavor` is the list of direct components.

```
(defflavor pie () (apple cinnamon))
(defflavor apple () (fruit))
(defflavor cinnamon () (spice))
(defflavor fruit () (food))
(defflavor spice () (food))
(defflavor food () ())
```

The resulting ordering of flavor components for `pie` is:

```
(pie apple fruit cinnamon spice food vanilla)
```

`vanilla` is the flavor that is always included, to provide default behavior.

Programmers are not allowed to mix together flavors with incompatible constraints. When no ordering of components can satisfy all of the constraints, *Flavors* lists the conflicting constraints and requires the programmer to take corrective action. For example:

```
(defflavor spice-cake () (spice pie))
```

produces the error message:

```
Cannot order the components of SPICE-CAKE:
  No ordering of SPICE, PIE, CINNAMON works:
    SPICE-CAKE has SPICE and PIE as
      direct components in that order.
    PIE depends directly on CINNAMON.
    CINNAMON depends directly on SPICE.
    However SPICE, PIE, CINNAMON have
      no conflicting methods.
```

Instance Variable Inheritance

The instance variables of a flavor are the union of the instance variables of its components. If several components define instance variables with the same name, they are combined into one instance variable. Thus instance variables can be used for communication between component flavors if the programmer so chooses. A method can access inherited instance variables by name. The variable `self` is scoped like an instance variable but its value is the instance itself.

Programmers often employ a convention that only certain methods access a given instance variable. These methods are considered to be inside the module that owns that instance variable. All other usage of that instance variable is by applying a generic function to `self`, which

provides a more abstract interface. This convention is allowed but not enforced by *Flavors*. The particular conventions for each instance variable in a program depend on the design of that program. This is similar to the way Lisp allows but does not enforce a convention that private functions of a module should only be called from inside that module.

Initialization of an instance variable is controlled by the most specific flavor that specifies an initialization. Often an instance variable is initialized by the flavor that defines it, but sometimes initializing an inherited instance variable is a useful way to customize inherited behavior.

Method Inheritance

When a generic function is applied to an object of a particular flavor, methods for that generic function attached to that flavor or to one of its components are available. From this set of available methods, one or more are selected to be called. If more than one is selected, they must be called in some particular order and the values they return must be combined somehow.

The simplest form of method inheritance is to use the method of the most specific flavor that provides one, and to ignore methods of more general flavors. This is often useful, but is not sufficient for all cases. Sometimes good program modularity requires that different parts of the implementation be specified by multiple methods, which must then be combined.

Method Combination

The selection and combination of methods for a (generic function, flavor) pair is controlled by a method-combination type. The programmer can specify the name of a type and optional parameters when defining a flavor, allow it to be inherited from a component flavor, specify it when defining a generic function, or simply allow the default type to be used. Several method-combination types are built in and programmers are encouraged to define additional types of their own.

The method-combination type sorts the available methods according to the component ordering, thus identifying more specific and less specific methods. It then chooses a subset of the methods (possibly all of them). It controls how the methods are called and what is done with the values they return by constructing Lisp code that calls the methods. Any of the functions and special forms of the language may be used. The resulting function is called a combined method.

Some examples of built-in method-combination types are:

- Call only the most specific method.
- Call all the methods, most-specific first or in the reverse order.
- Try each method, starting with the most specific, until one is found that returns a value other than `nil`.
- Collect the values of the methods into a list.
- Compute the arithmetic sum of the values of the methods.
- Divide the methods into three categories: primary methods, before-daemons, and after-daemons. Call all the before-daemons, then call the most specific primary method, then call all the after-daemons.
- Use the second argument to the generic function to select one of the methods.

Declarative Control of Method Combination

Programmers control method combination separately from the definition of the methods themselves. They control it by declaring a method-combination type and constraints on the ordering of component flavors. The details of how this declarative specification is implemented as executable code in the combined method can be ignored most of the time.

When defining a method, the programmer only thinks about what that method must do itself, and not about details of its interaction with other methods that aren't part of a defined interface. When specifying a method-combination type, the programmer only thinks about how the methods will interact, and not about the internal details of each method, nor about how the method-combination type is implemented. Programming an individual method and programming the combination of methods are two different levels of abstraction. Keeping them separate promotes modularity.

Defining New Method-Combination Types

Programmers can easily define new method-combination types. *Flavors* provides macros that accept a largely declarative specification of method sorting, filtration, and combination, and automatically produce the detailed code to combine the methods. The full details are beyond the scope of this paper, but the following examples of built-in types should convey the philosophy.

This defines a method-combination type named `:sum` that calls all the methods and passes their values as arguments to the `+` function. The generic function returns the sum of the numbers returned by the methods:

```
(define-simple-method-combination :sum +)
```

This defines the method-combination type mentioned earlier that calls daemon methods before and after the primary method:

```
(define-method-combination :daemon ()
    ;; Select three subsets of methods
    ;; by pattern-matching
    ((before "before" :every
             :most-specific-first (:before))
     (primary "primary" :first
              :most-specific-first ())
```

```
    (after "after" :every
           :most-specific-last (:after)))
  ;; A Lisp form that calls the methods
  `(multiple-value-prog1
    (progn ,(call-component-methods before)
           ,(call-component-method primary))
    ,(call-component-methods after)))
```

The pattern-matching specifications bind the variables **before** and **after** to lists of all the daemon methods and bind the variable **primary** to the most specific primary method. The backquote expression then generates the Lisp code that calls all the methods in the desired order and returns the values of the primary method.

Encouraging Program Modularity

No programming system can guarantee program modularity or eliminate the need for careful design of a program's structure. However, a programming system can make it easier to build modular programs. *Flavors* provides organizational techniques for writing programs in a modular way and keeping them modular as they evolve.

Inheritance of methods encourages modularity by allowing objects that have similar behavior to share code. Objects that have somewhat different behavior can combine the generalized behavior with code that specializes it.

Multiple inheritance further encourages modularity by allowing object types to be built up from a toolkit of component parts.

Interfaces between modules are typically defined as generic functions. Using any kind of function as an interface lends some abstraction. Using a generic function has the additional advantage that there can be several modules conforming to the same interface but each implementing it in a different way.

A common technique is to mix flavors so that a single object is composed of several modules. The modules communicate through generic functions applied to **self**, combination of methods (such as before- and after-daemons) belonging to different modules, and shared instance variables.

Easing Development of Large, Complex Programs

The encouragement of modularity outlined above is a key feature when developing large programs. In addition, *Flavors* makes it easy to change design decisions at any time and provides tools to assist the programmer in understanding and modifying the structure of the program.

The flexibility to change parts of a program quickly and easily is useful in the development stage. *Flavors* enables you to redefine flavors, methods, and generic functions at any time, even while the program is running. To do so, you need only evaluate a new definition, which replaces the old. When a flavor is changed, the system propagates the changes to any flavors of which it is a direct or indirect component. It is also possible to erase a definition without replacing it with a new one.

Changing the data representation is just as easy. If you redefine a flavor, to add or remove instance variables, old instances of the flavor automatically convert themselves to the new format the next time they are accessed.

You can redefine a generic function to be an ordinary function, or an ordinary function to be a generic function, without having to recompile its callers.

Program Development Tools

The Symbolics programming environment offers a variety of tools for analyzing *Flavors*-based programs. These tools can be invoked through the command processor or by pointing the mouse at displayed instances, flavor names, or method names.

Show Flavor Components
: Answers: What is the order of flavor components, and why did *Flavors* pick that order?

Show Flavor Dependents
: Answers: What flavors inherit from this one?

Show Flavor Instance Variables
: Answers: What state is maintained by instances of this flavor?

Show Flavor Operations
: Answers: What generic functions are supported by this flavor?

Show Flavor Methods
: Answers: What methods are defined for this flavor or inherited from its component flavors?

Show Flavor Initializations
: Answers: How are new instances of this flavor initialized?

Show Flavor Differences
: Answers: What are the differences between two flavors?

Show Generic Function
: Answers: What flavors provide a method for this generic function?

List/Edit Methods
: View or edit the source code of all methods for this generic function.

Show Flavor Handler
: Answers: When a given generic function is applied to an instance of a given flavor, what methods implement the operation? What is the actual Lisp code produced to combine these methods?

List/Edit Combined Methods
: View or edit the source code of the methods that

implement a given generic function on an instance of a given flavor.

Show Effect of Definition
Answers: If I evaluate this definition (text in the editor), or erase it, what changes would take place? For example, changing the order of a flavor's components might change which inherited method it uses for a particular generic function.

By pointing the mouse at a displayed object, the programmer can view:

- Names of the arguments and return values of a generic function. If not explicitly declared, these are deduced from the methods.
- Documentation for a flavor or a generic function.
- Source code of a flavor, a generic function, or a method.

Efficiency Considerations

Efficiency can be defined in many ways. We are concerned with four dimensions of efficiency:

- Efficient use of the programmer's time.
- CPU time.
- Page fault rate.
- Virtual memory occupied.

We optimize the programmer's time through declarative mechanisms, powerful tools, the flexibility to redefine pieces of the program, and a complex implementation that allows the programmer's own programs to remain simple.

The efficiency philosophy of *Flavors* is to optimize run-time speed to the maximum extent that does not compromise other goals, such as the flexibility to redefine anything while the program is running. In addition to *Flavors*-related goals, general Symbolics system goals, such as full run-time error checking, avoiding widespread use of declarations, and providing the best functionality, must not be compromised for the sake of efficiency.

Run-time speed is important because applications, including the development tools themselves, are typically built on several layers of *Flavors*-based substrate. The *Flavors* approach is to use a complex, machine-dependent implementation of relatively simple-appearing features, such as generic functions and instance variables.

Programmers enjoy the efficiency benefit of this implementation without having to write their own programs in a complex machine-dependent way.

CPU time and page fault rate determine response time, so they are more important than virtual memory size, which only consumes inexpensive disk storage. Consequently *Flavors* maintains multiple copies of information when that improves virtual memory locality or execution speed. Keeping those multiple copies consistent slows down program development operations, especially when modifying flavors that have hundreds of dependents. This tradeoff is acceptable, since development operations need not be faster than human speeds (several seconds), while run-time operations must operate at computer speeds (microseconds). Speeding up the run-time operations also speeds up the development tools built on them.

The key areas that are important to optimize in an object-oriented programming system are:

- Selection of one or more methods when a generic function is called
- Instance variable access from a method
- Instance creation

The following sections discuss how *Flavors* implements these operations.

Implementation of Method Selection

The first time a flavor is instantiated, or during compilation of the program if so directed, a handler function is precomputed for each generic function that the flavor supports. The method-combination procedure selects a set of methods and produces Lisp code that combines them. If the code can be optimized into calling a single method, that method is the handler. Otherwise a combined method is generated, compiled to machine code, and used as the handler. The combined method calls the methods with ordinary Lisp function calls.

The results of this precomputation are saved in a *handler table* associated with the flavor, keyed by the generic function. Thus when a generic function is called, the method selection process consists of finding the instance's flavor, looking in the handler table, and calling the handler. (See figure 2.) The handler table is a hash table whose structure is optimized to exploit the pipelined characteristics of the 3600's memory bus.

Subsequent changes to the program such as adding methods, removing methods, declaring a different method combination type, or changing a flavor's components incrementally update all affected handler tables, compiling new combined methods when necessary. For this purpose each flavor is linked to the flavors that depend on it and each combined method records how it was generated.

Implementation of Instance Variable Access

An instance is a represented as a block of storage whose first word references the flavor and whose remaining words contain values of instance variables. Methods that access instance variables cannot contain constant offsets of instance variables within the instance. These offsets are variable at run time, depending on the flavor of the instance, which can be any flavor that has the method's flavor as a component. Multiple inheritance makes it impossible to allocate fixed offsets to instance variables, because two flavors using the same offset might later be

mixed together and one of them would have to change.

The solution is to use indirect addressing. Each entry in a handler table includes a *mapping table*, which contains instance variable offsets. (See figure 2.) A method receives a mapping table as an argument. Offsets into the mapping table are compiled into methods. These offsets don't change, because when two flavors are mixed together each has its own mapping table. Accessing an instance variable fetches the offset from the mapping table, adds it to the address of the instance, and references that memory location.

When a combined method calls another method, it supplies a mapping table fetched from its own mapping table. Thus mapping tables actually form a tree structure parallel to the tree structure of flavor components.

In principle every flavor needs a separate mapping table for each component, and the total number of mapping tables could be proportional to the square of the number of flavors. In practice the average number of components of a flavor is small and only components that have instance variables need mapping tables. Thus the average number of mapping tables per flavor is only 4.1 and the total memory occupied by mapping tables is negligible.

Flexible Representation of Instances

Redefining a flavor in a way that changes the representation of instances, such as adding or deleting an instance variable, arranges for existing instances to be updated automatically. It makes a new flavor (with the same name) and changes the old flavor's handler table so that all generic functions rearrange the instance, change its flavor reference to the new flavor, and retry the operation. If the new instance representation is larger, rearranging the instance allocates new storage, copies the instance variable values into it, and deposits forwarding addresses with a special tag into the old storage. The instance variable accessing instructions and the garbage collector recognize this special tag.

Implementation of Instance Creation

The first time a flavor is instantiated, the initialization information from all its components is combined and saved in a convenient form. Subsequent instantiations consist of allocating storage, copying a template instance, initializing any instance variables whose initial values are not constant, and invoking initialization methods if any have been defined.

Timing Measurements

These measurements were performed on a Symbolics 3640 using the beta-test version of Release 7.0. Absolute timing measurements vary depending on hardware configuration, software version, and measurement methodology, so the most meaningful information here is the ratios.

Calling a generic function with two arguments takes twice as long as calling an ordinary function with two arguments; 13 microseconds versus 7. Times include computing trivial arguments, the actual call, executing a trivial function body, and the return.

The first instance variable accessed by a method takes about 5 microseconds; succeeding instance variables take 2.2 microseconds. For comparison, accessing a lexical variable in a closure takes 1.1 microseconds and accessing a `special` variable takes 1.7 microseconds.

Creating an instance with two instance variables using `make-instance`, the most general mechanism, takes 353 microseconds, 6.1 times as long as creating a `defstruct` structure with two slots. Using a *Flavors* constructor function, analogous to a `defstruct` constructor, takes 165 microseconds if keyword arguments are used or 68 microseconds with positional arguments, reducing the ratio to 2.8 or 1.2.

These timing measurements suggest that programming with *Flavors* is not substantially less efficient than non-object-oriented programming. In fact, it can be more efficient:

- A *Flavors* instance is smaller than a `defstruct`

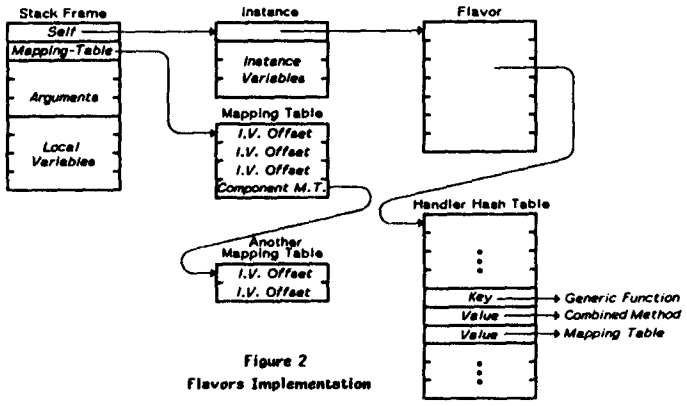

Figure 2
Flavors Implementation

structure with the same number of slots, by one word.

- Dispatching on object types with `typecase` takes about 15 times as long for a typical case as a generic function call.

Directions for Future Research

The following less well-understood aspects of object-oriented programming are good directions for future research:

- Protocols--Formalizing the notion of a generic interface, and further separating the contract of an object from the implementation of the object.
- Flavors for Non-Instances--Integrating the built-in data types of the Lisp language into the *Flavors* framework.
- Methods for Primitive Functions--Making `car` or `+` applied to an instance turn into a generic function and invoke a method, without slowing down the common non-object-oriented case.
- Higher-level Tools--Programmers of very large programs need all the help they can get. Programming tools that incorporate a model of the programming process, rather than just answering one question at a time, work better when the program is structured around a framework they understand. *Flavors* could be one such framework.
- Database--Integrating object-oriented programming with the persistent, reliable, data-independent structure of a database.
- Multiadic Operations--Generic functions that use more than one of their arguments to select methods are easy to implement, but a coherent and useful framework for organizing programs that work this way needs to be developed.

References

1. D. Weinreb, D. Moon, *Lisp Machine Manual*, MIT AI Lab, 1981, Chapter 20.

2. H. I. Cannon, "Flavors: A non-hierarchical approach to object-oriented programming", 1982.

3. R. D. Greenblatt, et al., "The LISP Machine", *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe, E. Sandewall, eds. McGraw-Hill, 1984.

4. G. L. Steele, *Common Lisp the Language*, Digital Press, 1984.

5. *Reference Guide to Symbolics Common Lisp: Language Concepts*, Symbolics Release 7 Document Set, 1986.