

Custom Specializers in Object-Oriented Lisp

Jim Newton
Cadence Design Systems
Mozartstrasse 2
D-85622 Feldkirchen Germany
jimka@cadence.com

Christophe Rhodes
Department of Computing
Goldsmiths, University of London
New Cross, London, SE14 6NW
c.rhodes@gold.ac.uk

Abstract: We describe in this paper the implementation and use of custom specializers in two current dialects of Lisp: Skill and Common Lisp. We motivate the need for such specializers by appealing to clarity of expression, referring to experience in existing industrial applications. We discuss the implementation details of such user-defined specializers in both dialects of Lisp, detailing open problems with those implementations, and we sketch ideas for solving them.

1 Introduction

Lisp has a venerable history of object-oriented programming; at one point in time, early in the history of object-orientation, Flavors [Moo86] and New Flavors, Common Objects, Object Lisp and Common Loops [BKK⁺86] all coexisted. The Common Lisp Object System (CLOS) was incorporated into the language in June 1988 [Ste90, Chapter 26], and when the ANSI Common Lisp standard [PC94] was formalized in 1995, Common Lisp became the first ANSI-standardized programming language with support for object-oriented programming.

In the object systems in the Lisps under discussion in this paper, method specializers have the function of determining whether a particular method is applicable to a set of function arguments or not; method qualifiers determine the function of the method within the effective method (from method combination) if the method is applicable at all.

In standard Lisps, the repertoire of specializers is limited: in Skill, only classes are allowed as specializers by default, matching instances of that class; in Common Lisp, classes and `eql` specializers (matching a single object by identity) are allowed by default, though the CLOS Metaobject Protocol (MOP) allows for extensibility in principle, as it specifies a `mop:specializer` metaobject class.

1.1 Custom Specializers

It is sometimes the case that applications require dispatch on objects whose behaviour is not separated by class structure; the dispatch may be influenced by the global application state, or by the values of slots in the objects, or other such factors. In object systems where the specializer metaobject class is not extensible, there is then an impedance mismatch between the expression of the functionality and its implementation, and it is this impedance mismatch that we address by allowing the user to define subclasses of the specializer class. By giving the user this option, we aim to provide a means to improve locality and clarity of the implementation of a particular solution to a problem, by allowing direct expression rather than manual reimplementing of dispatch machinery to distinguish between things that happen to be instances of the same Lisp class (or where the class of the object is not relevant for dispatch).

This paper discusses the use and implementation of metaobject protocols to allow the user to take advantage of the ability to define subclasses of the specializer class; after introducing some background and discussing related work in the next section, we present a worked example in section 3 to attempt to motivate the definition and use of such specializer metaobject classes. We discuss implementation issues regarding both Skill and Common Lisp in section 4, and conclude in section 5.

2 Background

2.1 Skill

The users of Cadence Design Systems' custom Integrated Circuit (IC) tools use the Skill[®] programming language [Bar90, Pet93] extensively. Programmers write applications which customize the look and feel of the graphical system, automate the design process by reducing the amount of repetitive work the design engineer must do, and perform time-consuming, tedious verification checks. Other types of programs include automatic layout generation which quickly produce parameterizable layouts which are correct by design. The language has an optional C-style syntax with many engineer-friendly shortcuts, making it easy for non-programmers to write simple scripts to help in their daily work.

The same language is also a Lisp system having the basic features one would expect: a Read-Eval-Print Loop (REPL), a debugger, garbage collection, lexical and dynamic scoping, macros, and anonymous functions. As with most Lisp systems, the language can be extended through adding functions to the runtime environment.

The Skill language has a built-in object system called the Skill++ Object System or simply Skill++. Skill++ is based on CLOS, but provides only a subset

of the capabilities; missing are features such as: multiple dispatch, multiple inheritance, method combination, method qualifiers, equivalence specializers, and a Metaobject Protocol. Instead, it provides single dispatch, single inheritance, analogues to Common Lisp's `call-next-method` and `next-method-p`, class and method redefinition, explicit environment objects, and a per-method choice between lexical and dynamic scoping. Also important to note is that while the language is interpreted by a proprietary virtual machine, the method dispatch mechanism in particular is implemented in a high performance compiled language; consequently, generic function calls are as fast as normal function calls.

2.2 Common Lisp

CLOS was developed in conjunction with the design of a Metaobject Protocol (MOP), described in *The Art of the Metaobject Protocol* (AMOP) [KdRB91]. Common Lisp as standardized only includes a very small portion of this Metaobject Protocol (for instance, a recommendation to use `mop:slot-value-using-class` in `slot-value`; some introspective functionality such as `find-method`; and arguably a little ability for intercession in `compute-applicable-methods`, though in fact the standard does not require that `compute-applicable-methods` be called as part of generic function dispatch), and so to customize the behaviour of the object system in Common Lisp it is necessary to go beyond the standard language.

Many Common Lisp implementations support some of the MOP, to varying extents; a survey from a few years ago [BdL00] revealed many aspects of MOP support as being incomplete, even at the coarse level of specified classes and generic functions being unimplemented. More recently, the Closer¹ project has provided both a set of test cases for implementations of the Metaobject Protocol – which has encouraged some implementations to enhance their support for it² – and a compatibility layer to provide an environment as close as possible to that described in AMOP in major implementations of Common Lisp.

2.3 Related Work

The issue of dispatch customization in Common Lisp has arisen before; for example, predicate dispatching in Common Lisp has been discussed in [Uck01]. In that work, the predicate was not restricted at all, and the solution presented involved extending method qualifiers (arbitrary predicates not being associated with any particular argument, and methods being distinguished from each other only on the basis of qualifiers and specializers). The author of this predicate

¹ <http://common-lisp.net/project/closer/>

² At the time of writing, the MOP implementation of Steel Bank Common Lisp [N⁺00] fails none of tests in the Closer MOP suite.

```
(defgeneric walk (expr env call-stack)
  (:generic-function-class sop-cons-generic-function))
```

Figure 1: Code walker generic function definition.

dispatch system notes portability difficulties with this approach, which would likely still be present today; for example, some implementations will only accept non-standard qualifiers if the generic function has a non-standard method combination. Strictly, `define-method-combination` will signal errors if methods are placed in the same method group having the same specializers (even if the intent is to use qualifiers to influence method applicability): qualifiers in Common Lisp are meant to affect method combination rather than method selection.

Predicate dispatch in other languages has also been investigated; a system has been presented and implemented for Java [Mil04], wherein the predicates affecting dispatch are restricted to a set which can be reasoned over, and for which ambiguities are forbidden in the selection of the most specific method. We prefer to leave such policy decisions to the users of the system, at least while the capabilities and expressiveness are being explored: if it turns out that restricting specializers to express a limited set of predicates is acceptable, that can be enforced at a later stage.

At this time, we make no attempt to implement a specific predicate dispatch mechanism in either Skill or Common Lisp, but rather aim to provide a framework which is both sufficiently general to express predicate dispatch and straightforward to use, allowing issues of determinism, portability and performance to be explored and addressed by users.

3 Using Custom Specializers: a Worked Example

The following excerpts are from a code walker expressed using custom specializers. The code walker examines code written in a particular Lisp dialect and reports unbound and unused variables. For purposes of simplicity, the illustrated implementation uses a Common Lisp-like syntax, with Skill-like semantics in one or two respects noted below.

The goal of this illustration is to give an example of a solution that is more parsimonious when the language supports describing actions on wider ranges of data, rather than to convince that a particular type of specializer (such as the `cons` specializer used here) itself is a good idea. As with any pedagogical example, the same application could be written in many different ways without great loss of clarity.

```

(defmethod walk ((expr list) env call-stack)
  (let ((call-stack (cons expr call-stack)))
    (walk (car expr) env call-stack)
    (walk (cdr expr) env call-stack)))

(defmethod walk ((expr (eql nil)) env call-stack)
  nil)

(defmethod walk ((expr t) env call-stack)
  (format t "invalid expression ~A: ~A: ~A~%"
    (class-name (class-of expr)) expr call-stack))

```

Figure 2: Recursion engine and termination condition

The form in figure 1 defines the generic function `walk` as an instance of the generic function class named `sop-cons-generic-function`, which is assumed to already exist. We discuss the implementation issues of this metaclass in section 4.1.

The implementation of `walk` we present here contains four conceptual parts:

- a recursion engine which includes a termination condition and error handling;
- code to recognize variable references and mark bindings as *used*;
- code to ignore all irrelevant forms encountered during the recursion;
- code to handle special forms.

We begin by implementing the first three parts using standard CLOS functionality; the part to handle special forms is then implemented using a non-standard subclass of `mop:specializer`.

3.1 Code Walker Framework

The main engine of the code walker (figure 2) starts at a top level expression. If the expression is a list, it calls itself recursively on the elements of the list – with a few notable exceptions. Some of the necessary exceptions can be handled by equivalence specializers such as `(eql t)` and `(eql nil)`. Lisp special forms, such as `(quote ...)` and `(lambda ...)` forms, cannot be described by equivalence specializers but can be with `cons` specializers.

Next is the traversal engine based on the class specializer `list` and the termination condition based on an equivalence specializer `(eql nil)`. Thus the engine keeps traversing the lists until they are exhausted. There is also a method specializing on class `t` which will be called if something is encountered which the code

```
(defmethod walk ((var symbol) env call-stack)
  (if-let (binding (find-binding env var))
    (setf (used binding) t)
    (format t "unbound: ~A: ~A%" var call-stack)))
```

Figure 3: Checking the bindings of symbols.

```
(defmethod walk ((expr string) env call-stack)
  nil)
(defmethod walk ((expr number) env call-stack)
  nil)
(defmethod walk ((expr (eql t)) env call-stack)
  nil)
```

Figure 4: Ignoring certain atoms.

walker cannot otherwise handle. The job of the methods that follow will be to assure that everything that occurs in the traversal is handled by an appropriate method and that the "invalid expression" message never gets printed.

When a symbol is encountered the method in figure 3 is applicable. A check is made to see whether the variable is bound in the environment³. If so, the `used` slot of the binding object is set to true, to note that the binding is used. If the variable is unbound, then a diagnostic message is emitted, informing the user of where the reference to an unbound variable is made.

Figure 4 shows how certain types of self-evaluating atoms such as strings, numbers, and the symbol `t` are simply ignored when searching for variable references. A full implementation of this would ignore all atoms which cannot name variables; in this restricted Common Lisp-like language, we assume that those objects are instances of either `string` or `number`.

3.2 Special Forms

We now implement some of the special forms. Note that `quote` and `lambda` themselves are not special forms; they are simply symbols which evaluate as any other symbol – if one of these symbols is encountered in a context where it is used as a variable, the code walker must treat it as such. This means we cannot write

³ The implementation of the `find-binding` function is omitted. It returns a *binding* object by searching for a named variable in a given *environment* object. Such a binding object has an accessor named `used` to hold a boolean, indicating whether the binding is used or not.

```

(defmethod walk ((form (cons (eql quote))) env call-stack)
  nil)

(defmethod walk ((form (cons (eql lambda))) env call-stack)
  (destructuring-bind (lambda lambda-list &rest body) form
    (let ((bindings (derive-bindings-from-ll lambda-list)))
      (dolist (form body)
        (walk form (make-env bindings env) (cons form call-stack)))
      (dolist (bind bindings)
        (unless (used bind)
          (format t "unused: ~A: ~A~%" var call-stack)))))))

```

Figure 5: Handling the (`quote ...`) and (`lambda ...`) special forms.

a method for `walk` specializing on (`eql quote`)⁴. However, lists for evaluation whose first elements are `quote` or `lambda` are special and must be intercepted before the walker reaches the `quote` and `lambda` symbols themselves.

The `cons` specializer provides a mechanism for making a method applicable for such a list. Figure 5 implements methods for handling `quote` and `lambda` forms. The first method is applicable if its first argument is a list whose first element is the symbol `quote`. Since an evaluator would simply return the second element of this special form unevaluated, there can be no variable references inside it; so the code walker simply returns `nil`.

The second method handles `lambda` forms by creating new bindings as indicated by the lambda list and walking the body of the `lambda` with those bindings in place. After the code walker returns from walking the lambda body we can report if any of the new bindings were not referenced by the walked code.⁵

This implementation of `walk` is a simplified version of a walker for Skill that is used in production; we have elided many details of the full version. For example, rather than printing diagnostics, the walker communicates with the environment, allowing the offending forms to be highlighted in the editor; additionally, the walker supports a much broader range of the Skill language semantics, including ignorable and global variables, assignment, macro expansion and more special forms. The user-defined `cons` specializer presented here allows us to have a single generic function, `walk`, whose methods specialize on all of the different types of

⁴ Note that unlike in Common Lisp, here the argument of the `eql` specializer is unevaluated; (`eql quote`) is correct, rather than (`eql 'quote`). We discuss this further in section 4.2.

⁵ The implementations of the functions `derive-bindings-from-ll` and `make-env` are omitted for this illustration as they do not aid in understanding extensible specializers. The `derive-bindings-from-ll` function returns a list of *binding* objects from a lambda list. The `make-env` function allocates a new *environment* which references the given list of binding objects, and also references the given parent environment.

forms that must be handled differently.

As an example of perhaps a potentially generally useful specializer type, consider a specializer corresponding to a pattern, similar to those found in the ML family of languages. Using the mechanisms presented in this paper, it is possible to have the dispatch over patterns optimized as is expected in those languages, while still retaining the customary run-time extensibility of Lisp, by lazily compiling the dispatch (using algorithms such as those in [LFM01]) and invalidating the compiled code if methods are added or removed to the pattern-matching generic function.

An application which, we believe, would benefit from a protocol for defining specializers for which there is no corresponding hierarchy is an Emacs-like text editor, where ‘minor modes’ can affect the functionality of keystrokes and editor function calls. For instance, in the Climacs text editor [RSM05], minor modes are currently implemented by the creation of anonymous classes with a combination of superclasses corresponding to the currently-active modes, whereas it should be simpler to express this as a dispatch on aspects of the current editor state.

4 Implementation Details

4.1 Skill, Skill++ and VCLOS

To address the limitations of Skill++ (see section 2.1) a new object system for Skill was needed, to provide more of the features of CLOS. The new object system was required to be able to interface to programs written in the existing Skill++ system, and allow object-oriented techniques to be used on existing systems whose object models are not changeable, while also being extensible for the types of problems faced in application programming for IC development.

Neither VCAD (an organizational department within Cadence Design Systems) nor VCAD’s customers have write access to the Skill implementation, and so the language itself cannot be changed: the object-oriented extension must be provided as a loadable Skill application. From its Lisp heritage, Skill can be altered in this way so that the extension seems native to the Skill programmer and invisible to the end-user.

4.1.1 VCLOS and its Metaobject Protocol

The resulting system, VCAD CL-like Object System (VCLOS), was developed over several years; the major difference from CLOS and its Metaobject Protocol [KdRB91] is that more importance is given to the `mop:specializer` metaobject class, rather than having most of the dispatch functionality of generic functions be computed from the `class` of arguments.

The VCLOS Metaobject Protocol implemented is then similar to the CLOS MOP, with the following points to note:

- the `ClosClassSpecializer` and `ClosEqvSpecializer` classes are both subclasses of `ClosSpecializer`, while users are encouraged to define their own subclasses of `ClosSpecializer` by the provision of a protocol for using them in computation of the effective method (described further below);
- in VCLOS, `ClosComputeApplicableMethodsUsingSpecializers` takes the place of `mop:compute-applicable-methods-using-classes` in the standard AMOP generic function invocation protocol;
- a good CLOS implementation will memoize the results of `mop:compute-applicable-methods-using-classes` if possible, with a key based on the classes of the arguments (see [KR93] for some details). VCLOS supports memoization based on specializer names, computed using `ClosComputeSpecializerNames`.

In order to use a user-defined specializer class, the user must define a subclass of `ClosSpecGenericFunction`, the generic function subclass following the protocols for extensible specializers. The protocol defined on `ClosSpecGenericFunction` allows for the user to specify how to put specializers in precedence order through defining methods on Metaobject Protocol functions: `ClosAvailableSpecializers` and `ClosCmpLikeSpecializers`. The method on `ClosAvailableSpecializers` applicable to a particular generic function class must return a list of specializer class names, from most specific to least specific; methods on `ClosCmpLikeSpecializers` must decide which of two specializers of the same class (assumed both applicable to the same generic function argument) is more specific.

Among the other Metaobject Protocol functions which need to have methods defined for user-defined specializers to work are `ClosArgMatchesSpecializerP`, a function of a specializer and an arbitrary object, which returns true if a specializer corresponds to a type of which the given object is a member, and `ClosGetClassPrecedenceList` (which should perhaps have been called `ClosGetSpecializerPrecedenceList`), which for a given specializer computes a linearization of its less-specific specializers.

The treatment mapping specializer surface syntax such as `(cons quote)` to specializer metaobjects is performed by generic functions `ClosMatchesSpecializerSyntaxP` and `ClosSetSpecializerData`. Note that in this respect the Skill protocol and the extension to the Common Lisp Metaobject Protocol described in appendix A differ in the approach taken, as in Common Lisp the specializer syntax is sensitive to the lexical environment.

The Skill implementation of VCLOS provides memoization, keyed on the specializers of the arguments, to the effective method, allowing the elision of calls to `ClosComputeApplicableMethodsUsingSpecializers`, in a similar way to

the CLOS MOP protocol around `mop:compute-applicable-methods-using-classes`. There is still overhead involved in computing appropriate specializers corresponding to the arguments, relative to the baseline of computing an argument's class, but this memoization can significantly reduce the overhead of using a non-standard specializer.

4.2 Common Lisp and the Metaobject Protocol

Much of the work in implementing custom specializers in Skill was of course taken up by providing a suitably rich object system such that customizations can meaningfully be made: essentially, taking a single-dispatch, single-inheritance object system as found in Skill++ and implementing on top of it a multiple-dispatch, multiple-inheritance system with a Metaobject Protocol. By contrast, in Common Lisp (with the *de facto* standard MOP) we already have most of the framework for the implementation of custom specializers; for basic operation, we only require a few non-standard operators.

The Lisp-like language we have used for our examples, and the actual implementation of the specializer metaobject class in Skill, share one important difference in detail from Common Lisp. In Common Lisp's `defmethod` macro, the `eq1` specializer specifies not a specialization on a following literal, but instead a specialization on the value of a form in the lexical environment of the method definition.

This detail implies that there must be an operator, similar to `mop:make-method-lambda`, which is capable of converting surface syntax such as `(eq1 foo)` into code which constructs an `mop:eq1-specializer` metaobject at the time when the `defmethod` is executed. Of course, we could restrict the use of the lexical environment to the standardized `eq1` specializer, but since it is possible to support culturally-compatible use of the lexical environment through a relatively straightforward backward-compatible extension to the CLOS Metaobject Protocol (see appendix A), we choose to do so, defining our new operator as `make-method-specializers-form`. For convenience, we also suggest `parse-specializer-using-class` and `unparse-specializer-using-class` to consume and produce user-friendly representations of specializers, for use in `find-method` and printed representations of methods.

4.2.1 VCLOS implementation in SBCL

We have in addition implemented a version of the Skill and VCLOS Metaobject Protocol described in section 4.1.1 above, and used it to run the `walk` example from section 3. The implementation of the VCLOS protocol in SBCL's MOP is by no means complete and certainly not industrial-strength; however, even the simple implementation raises some issues.

Firstly, initial explorations revealed that current Common Lisp implementations have only partial support for subclassing `mop:specializer`; most implementations will allow defining the subclass, but very few recognize such a subclass as a valid specializer. In the implementation for SBCL, we had to alter a number of places in the CLOS implementation where the assumption had been made that a specializer was either a `class` or an `eql-specializer`.

Secondly, since the system needs to call, as part of the discriminating function, a new function `compute-applicable-methods-using-specializers` instead of the usual `compute-applicable-methods-using-classes` (and we need to be calling `specializer-of` rather than `class-of` on the generic function arguments), we must override `mop:compute-discriminating-function` for our generic function class. This in turn means that we need to interpret or compile the result of `mop:compute-effective-method` ourselves, which is not a straightforward procedure, as suitable definitions for `call-method` and `make-method` need to be provided; `mop:compute-effective-method` returns a form, not something which is directly executable.

Additionally, we need to provide an implementation of `compute-applicable-methods`, as well as the new protocol function `compute-applicable-methods-using-specializers`, because the new methods must call our protocol function `specializer-applicable-p` (for determining whether an argument matches a specializer). An implementation is not difficult in principle, but tedious and error-prone; because of limited resources we have instead provided a method which considers only the first required argument to a generic function, leaving the implementation of the multiple-dispatch aspect for further work.

While the presence of user-defined specializers makes it harder to reason about the cacheability of effective methods or lists of applicable methods, there are still points in the protocol discussed above which would allow a value to be computed once and reused for efficiency; our current implementation in Common Lisp does not take advantage of these.

5 Conclusions and Future Work

We have presented the implementation and use of custom specializers both in a Lisp dialect where that functionality is used, and also in Common Lisp, a language with a standardized core and *de facto* standard Metaobject Protocol.

The implementation in Skill is complete and used in production: the implementation is fully functional, has an extensive suite of unit tests, and is part of live design projects. Much time has been spent on optimization and refactoring for performance and readability of the code, but of course much more work in this area could be done.

The functionality for the user to define their own specializer classes has been available in SBCL since May 2007; in practice the design space seems to be too

general for easy exploration: having to reimplement the entirety of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` is excessive. Our ‘toy’ implementation of the VCLOS protocols should be refined and extended, so that users can experiment with their specializer classes without having to reimplement complicated protocol functions.

In particular, it is important to take advantage of the various points in the protocol where memoization can be used (in the calculation of the effective method, for instance), so that the run-time overhead from use of user-defined specializers is as low as possible. Doing this would allow us to compare the efficiency of the protocol implementation in Skill and Common Lisp, and to identify further points for optimization if necessary.

One thing missing from the Metaobject Protocol for Common Lisp (including our extension) is a general case for something that SBCL in particular takes advantage of: in SBCL, a method definition with a standard specializer will inform the method body (by inserting a declaration) that the corresponding element in the method function arguments is of a relevant type. There is at present no way of communicating this information for an arbitrary user-defined specializer.

Acknowledgments

Skill® is a registered trademark of Cadence Design Systems, Inc.

References

- [Bar90] Timothy J. Barnes. SKILL: A CAD system extension language. In *DAC '90*, pages 266–271. ACM, 1990.
- [BdL00] Tim Bradshaw and Raymond de Lacaze. A Survey of Current CLOS MOP Implementations. In *Japan Lisp Users Group Meeting*, 2000.
- [BKK⁺86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Common Loops: Merging Lisp and Object-Oriented Programming. In *OOPSLA '86 Proceedings*, pages 17–29, 1986.
- [KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [KR93] Gregor Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In Andreas Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, pages 335–348. MIT Press, Cambridge, Mass., 1993.
- [LFM01] Fabrice Le Fessant and Luc Maranget. Optimizing Pattern Matching. In *ICFP'01 Proceedings*, pages 26–37, 2001.
- [Mil04] Todd Milstein. Practical Predicate Dispatch. In *OOPSLA '04*, pages 345–364. ACM, 2004.
- [Moo86] David Moon. Object Oriented Programming with *Flavors*. In *OOPSLA '86 Proceedings*, pages 1–8, 1986.
- [N⁺00] William Harold Newman et al. SBCL User Manual. <http://www.sbcl.org/manual/>, 2000.

- [PC94] Kent Pitman and Kathy Chapman, editors. *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI, 1994.
- [Pet93] Edwin S. Petrus. SKILL: a Lisp based extension language. *Lisp Pointers*, VI(3):71–79, 1993.
- [RSM05] Christophe Rhodes, Robert Strandh, and Brian Mastenbrook. Syntax Analysis in the Climacs Text Editor. In *International Lisp Conference Proceedings*, 2005.
- [Ste90] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, second edition, 1990.
- [Uck01] Aaron Mark Ucko. Predicate dispatching in the Common Lisp Object System. Technical Report AITR-2001-006, MIT AI Lab, Cambridge, MA, 2001. MEng thesis.

A Common Lisp extension to the MOP

The CLOS Metaobject Protocol requires little extension to support everything discussed in this paper. On a fundamental level, in fact, no new operators are required, though for convenient use of the standardized operators `defmethod` and `find-method` we propose an analogue to `mop:make-method-lambda` and operators to parse and unparse parameter specializer names.

In order to emulate the specific specializer handling present in VCLOS, an overriding implementation of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` would be necessary. However, for any particular strategy for dealing with the method applicability and ordering computation, such an implementation need only be written once; once written, the CLOS user would be free to implement specializers using the defined protocol.

A.1 Dictionary

Generic Function `parse-specializer-using-class`

Syntax:

`parse-specializer-using-class` *generic-function* *specializer-name*

This generic function returns an instance of `mop:specializer`, representing the specializer named by *specializer-name* in the context of *generic-function*.

Primary Method `parse-specializer-using-class` (*gf* `standard-generic-function`) (*name* *t*)

This method applies the standard parsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `unparse-specializer-using-class`

Syntax:

`unparse-specializer-using-class` *generic-function* *specializer*

This generic function returns the name of *specializer* for generic functions with class the same as *generic-function*

Primary Method `unparse-specializer-using-class` (*gf* `standard-generic-function`) (*specializer* *specializer*)

This method applies the standard unparsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `make-method-specializers-form`

Syntax:

`make-method-specializers-form` *generic-function* *method* *specializer-names* *env*

This function is called with (maybe uninitialized, as with the analogous arguments to `mop:make-method-lambda`) *generic-function* and *method*, and a list of specializer names (being the parameter specializer names from a `defmethod` form, or the symbol `t` if unsupplied), and returns a form which evaluates to a list of specializer objects in the lexical environment of the `defmethod` form.

Primary Method `make-method-specializers-form` (*gf* `standard-generic-function`) (*method* `standard-method`) *names env*

This method implements the standard behaviour for parameter specializer names.