Increasing Readability and Efficiency in Common Lisp

António Menezes Leitão

Instituto Superior Técnico
Dep. de Engenharia Informática
Grupo de Inteligência Artificial
Av. Rovisco Pais
1049-001 Lisboa
Portugal
Tel: 351-1-8417641
Fax: 351-1-8417472
E-mail: aml@gia.ist.utl.pt

Abstract

Common Lisp allows programmer intervention in the compilation process by means of macros and, more appropriately, compiler macros. A compiler macro describes code transformations to be done during the compilation process. Unfortunately, compiler macros are difficult to write, read, and extend. This article presents a portable extension to Common Lisp's compiler macros. The extension allows easy definition and overloading of compiler macros and is based in two known techniques, namely pattern matching and data driven programming. Three different uses are presented: (1) Code optimization, (2) Code deprecation, and (3) Code quality assessment. The extension is being used extensively in the reengineering of a large AI system, with good results.

1 Introduction

There is a conflict between *readability* and *efficiency*. It's a programmer's dream to have a Sufficiently Smart Compiler that translates a clearly written program into efficient machine-code. Unfortunately, such compiler is only a dream. In order to achieve acceptable performance, at least some parts of the program must be hand-optimized. Usually, these optimizations transform readable and inefficient algorithms into efficient and unreadable algorithms.

In this article, we analyze this problem in the context of the Common Lisp language, as defined in [AI96]. This language is much more complex than most functional languages and the presence of several impure features makes it very hard to prove properties about programs. As a consequence, it is difficult to generate efficient code without help from the user.

To understand the problem, let us write an algorithm to compute the sum of the squares of the odd numbers in a given list *list*, using a functional style. Using Common Lisp, we could write:

```
(reduce #'+
        (mapcar #'square
                (remove-if-not #'oddp
                               list)))
```

Common Lisp is a strict language. This implies that the previous computation produces two intermediate lists that are used just to "transport" values from one function to the other. These intermediate lists consume both space and time.[1]

If the code above turns out to be a bottleneck in a larger system, the programmer will be tempted to rewrite it so that no intermediate structures are used. Here's one possibility:

```
(let ((sum 0))
  (dolist (i list sum)
    (when (oddp i)
      (setq sum (+ sum (square i))))))
```

Unfortunately, the code is much less readable. The outermost computation, to compute the sum, is buried inside the iteration. Also, the filtering is not explicit, as it is replaced by a combination of an iteration and a conditional execution. Everything is done at the same time. The code is harder to understand and modify.

2 Related Work

For the particular example given above, there's an efficient solution that combines both readability and efficiency: Series [Wat89].

A series is a data-structure similar to a sequence but for which functional composition doesn't necessarily entail generation of intermediate structures and, in most cases, can be executed iteratively.

Series allows the programmer to express its algorithms in a functional style while maintaining the efficiency of an imperative style.

The previous example can be rewritten in Series as follows:

```
(collect-sum (map-fn t #'square
                     (choose-if #'oddp
                                (scan list))))
```

This code is then converted into a loop equivalent to the hand optimization presented above, thus providing both a clear style and an efficient execution. Series is the Common Lisp equivalent to the *deforestation* techniques ([Wad90] [GJ94]) available in other functional languages.

Unfortunately, a series is a new data type, disjoint from all other Common Lisp data types. To take advantage of the series idea, we should use series from the beginning. On the other hand, systems developed by incremental refinement start with simpler data types that are progressively adapted, and where converting the entire system to use series may be a very difficult task.

[1] In a lazy language, the space is not a problem because each list element is produced as it is needed, but the operation is still time-consuming.

Our goal is a bit different. We want to write and modify large software systems without early commitment to one data type or paradigm. After debugging the code, we want to selectively give advice to the compiler to speed up our system, keeping the readability of our source code but generating efficient machine code. Last, but not least, we want these methods to be applicable to any piece of code, and not only to some data type as in the Series package. The important lesson of the Series package is that, instead of depending on a Platonic super compiler, we can maintain readability *and* efficiency if we provide code transformations that translate our code into more efficient forms that are subsequently compiled by our realistic compilers.

3 Efficient Code

Modern Common Lisp compilers try their best to generate efficient code. Some common tricks include open-coding primitive functions, inlining, and constant folding. Nevertheless, such optimizations affect only relatively small parts of the program and cannot make radical changes, such as the ones provided by the Series package.

But there's much more that a compiler can't do. In general, it is impossible for a compiler to predict all the possible program transformations that can be useful. Besides, many program transformations can only be done if the program being transformed satisfies a set of properties, which can be difficult or impossible to prove.

3.1 Avoiding Intermediate Structures

To give a feeling about the kind of properties that we are talking about, we will describe our own experience. During the development of a software system where sets are a fundamental entity, we found ourselves writing mappings, filterings, and foldings over sets.

Let's suppose that the constructors for sets are `empty-set`, which returns an empty set, and `insert-set`, which accepts an element and a set and returns another set with the element included if it was not there already. We also have a generic `fold` operation[2] that accepts a combining function, a starting value, and a set and combines all the elements of the set. Let's also suppose that sets can be read and written using brackets.

First, let us look at some operations that generate intermediate sets that are immediately discarded. For example, the previous algorithm can also be applied to sets:

```
(fold-set #'+ (empty-set)
          (map-set #'square
                   (filter-set #'oddp set)))
```

It is easy to see that after the filtering the resulting set is given to the mapping and then it is simply discarded. Mappings after filterings is such a common combination that we can win a lot if we combine both operations in a single one that computes the same result without generating the intermediate

[2] Since there isn't any order on the set elements, we don't need to distinguish between a left or right fold.

set. We will call this combined operation `map-after-filter-set` and we just rewrite the previous computation as:

```
(fold-set #'+ (empty-set)
          (map-after-filter-set #'square
                                #'oddp
                                set))
```

Of course, the same idea can be applied to the folding after mapping after filtering, so we can just rewrite it again as follows:[3]

```
(fold-after-map-after-filter-set #'+ (empty-set)
                                 #'square
                                 #'oddp
                                 set)
```

Unfortunately, the code is much less readable. The eyes must read back and forth to know what is being mapped or used as filter. Readability is lost in favor of efficiency. We would like to have both.

A possible solution is to provide a function with keyword parameters, as was done in several standard Common Lisp functions. Here is an attempt:

```
(fold-set #'+ (empty-set)
          set
          :map-set #'square
          :filter-set #'oddp)
```

but this doesn't look as clear as the original and, more important, it can be very misleading, as the following example shows:

```
(fold-set #'+ (empty-set)
          set
          :filter-set #'oddp
          :map-set #'square)
```

Obviously, both applications produce the same result, although they look radically different.

3.2 Efficient Special Cases

Before we look at other solutions, let us see another example. We can define the mapping of a function over a set as follows:

```
(defun map-set (f s)
  (fold-set #'(lambda (e s)
                (insert-set (funcall f e) s))
            (empty-set)
            s))
```

Some examples of use follow:

```
> (map-set #'square '{1 2 3 4})
{1 4 9 16}
> (map-set #'square '{-2 -1 0 1 2})
{0 1 4}
```

[3] Obviously, the number of possible combinations can be very large, but that's not an issue in this article. The important point is that, somehow, we must have more efficient ways to compute the result, so that no intermediate sets are needed.

We can note the similarity between the definition of mapping for sets and a similar definition for lists:

```
(defun map-list (f l)
  (foldr #'(lambda (e l)
             (cons (funcall f e) l))
         '()
         l))
```

Unfortunately, whatever the implementation of sets we use, the `insert-set` operation can never be as efficient as a `cons` because before inserting an element in a set, we must check that the element isn't already there. On the other hand, in the mapping case, we sometimes know that the mapped function is injective, and this implies that each mapped value has a unique image, making the checking for repeated elements redundant. This allows a more efficient implementation of the mapping for the special case of an injective function.

Accordingly, let us assume that there is an operation that, given an element and a set, returns another set that includes the element, simply assuming that the element didn't belong to the set in the first place. Let's name this operation `dont-check-insert-set`. With this operation, we can define a new mapping operation:

```
(defun injective-map-set (f s)
  (fold-set #'(lambda (e s)
                (dont-check-insert-set (funcall f e) s))
            (empty-set)
            s))
```

Having two mapping operations is not what we would call "programmer-friendly." If we can use the same "+" operation to add two floats and two bignums, we would also like to use the same `map-set` to map both an injective function and a non-injective one.

A possible solution to this problem is to tag each function as injective or not, and let a dynamic dispatch mechanism determine the correct mapping function. We could use CLOS—the Common Lisp Object System [Kee89], [Ste89]—and more specifically, the MOP—the MetaObject Protocol [KdRB91]—to define a metaclass `injective-function` for injective generic functions. The `map-set` function would then be specialized for injective generic functions and for all other functions. In the first case, the function builds the resulting set using the more efficient `dont-check-insert-set` while in the second case it must use the slower `insert-set`.

Using this approach, we could define a generic injective function, e.g., the symmetric function that, given a number, returns its symmetric:

```
(defgeneric symmetric (x)
  (:generic-function-class injective-function))

(defmethod symmetric (x)
  (- x))

> (map-set #'symmetric '{-2 -1 0 1 2})
{2 1 0 -1 -2}
```

Unfortunately, such definitions constrain us to only map generic functions (and not ordinary functions and lambdas). Also, we cannot use already defined functions as, most probably, they weren't defined as generic ones or, even if they were defined that way, the generic function class would not be the one we need. Finally, using generic functions will have a negative impact on performance that can outweigh the speedup obtained by avoiding checking membership.

Another solution is to use some mechanism similar to the special operator `the`. According to the language definition [AI96], `the` receives a value-type and a form and it specifies that the values returned by form are of the types specified by value-type. The consequences are undefined if any result is not of the declared type.

Compilers can take advantage of `the` declarations to produce more efficient code. For example, the function

```
(defun fixnum-add (x y)
  (the fixnum (+ (the fixnum x) (the fixnum y))))
```

compiled with high levels of optimization (and low levels of safety) should be significantly faster than a generic `+`.

It would be nice if we could use the same operator to write

```
> (map-set (the injective-function #'symmetric) '{-2 -1 0 1 2})
{2 1 0 -1 -2}
```

and have the code compiled as if `injective-map-set` was being used in the place of `map-set`.

Unfortunately, Common Lisp doesn't provide much support for such intervention in the compiler workings. Although it is possible to define new data types, we cannot modify the compiler to use them properly. But there's one thing we can do that modifies the compiler behavior: compiler macros.

4 Compiler Macros

According to the ANSI Common Lisp definition [AI96], "the purpose of the compiler macro facility is to permit selective source code transformations as optimization advice to the compiler".

The use of compiler macros is described as follows [AI96]:

> When a compound form is being processed (as by the compiler), if the operator names a compiler macro then the compiler macro function may be invoked on the form, and the resulting expansion recursively processed in preference to performing the usual processing on the original form according to its normal interpretation as a function form or macro form.

Compiler macros are defined using `define-compiler-macro`. The following example (from [Ste89]) instructs the compiler how to optimize calls to an hypothetical function `plus`:[4]

[4] The standard says that "The *name* [of the compiler macro] can be a *function name* naming any function or macro." My reading of "can" allows the name of a compiler-macro to *not* be associated with any function or macro. Obviously, since compiler macros are dealt with exclusively by the compiler, compiler macros not associated with a function or macro aren't useful at the interpreter level.

```
(define-compiler-macro plus (&whole form &rest args)
  (case (length args)
    (0 0)
    (1 (car args))
    (t form)))
```

With this compiler macro, calls of the form `(plus` *expr*`)` are replaced upon compilation by *expr*, and calls of the form `(plus)` become simply `0`. This is similar to what current Common Lisp compilers do when compiling forms that use the arithmetic operators. Like `plus`, these operators accept a variable number of arguments but when this number is known at compilation time, it is possible to generate more efficient code.

What we want to stress is that a programmer need not be concerned with changing an algorithm so that it runs faster. Instead, he can concentrate on providing a simpler version of the algorithm *and* a set of compiler macros that optimize the algorithm for the most important cases.

Returning to our previous example, we can define a compiler macro such that whenever the compiler sees `(map-set (the injective` *function*`)` *set*`)` it transforms it into the more efficient `(injective-map-set` *function set*`)`.

This choice has the advantage of being integrated with the usual type declaration of Common Lisp. Unfortunately such integration is just apparent and can be broken very easily. A trivial example is as follows:

```
(let ((func (the injective #'1+)))
  (map-set func set))
```

Converting such code to a more efficient form is difficult to do with compiler macros, unless we use a full-fledged code-walker. In practice, it is hard to use the "type declaration." Using a mechanism that only in form resembles another one that has a predefined and well-know behavior is asking for trouble. This led us to think that it would be better to create an orthogonal mechanism with well defined semantics.

5 Tagging Code

Our idea is to define some functions that can be used as tags for function parameters. Again, using our example, we could write

```
(map-set (injective #'1+) set)
```

During development the function `injective` returns another function that, for each application, checks whether the computed value was also computed for any other different arguments. In this case, the function reports an error, otherwise caches the computed value and the corresponding arguments and returns the computed value.

When the debugging phase is over, we can introduce the compiler macro

```
(define-compiler-macro map-set (&whole form &rest args)
  (if (and (consp (first args))
           (eq (first (first args)) 'injective))
    `(map-injective-set ,(second (first args)) ,@(rest args))
    form))
```

and then the previous code is translated into the equivalent form:

```
(injective-map-set #'1+ set)
```

One major advantage is that, instead of having to know the names of the different mapping functions over sets, we just know the generic operation and some qualifiers that we can apply to speed up some calls. But there is an even bigger advantage, namely the increased readability of the code. It is still the good old `map-set` but one argument is more thoroughly specified, by "declaring" it as an injective function.

The programmer should be warned that the declaration is only useful if used to tag an argument of an application. It is not a type declaration in the Common Lisp sense and it is useless, as regards efficiency, to use it to type a let binding. However, it might be helpful for debugging purposes.

The concerns regarding the use of the `injective` function have a correspondence with similar concerns regarding the deprecation of the `:test-not` keywords or `-if-not` functions after the introduction of the `complement` function in Common Lisp. This change in the language can have a negative impact on efficiency unless compilers can recognize the most common uses. For example, upon seeing

```
(remove-if (complement #'numberp) list)
```

the compiler can treat the expression as the equivalent and presumably more efficient

```
(remove-if-not #'numberp list)
```

On the other hand, this equivalence might be harder to prove if the compiler sees

```
(let ((func (complement #'numberp)))
  (remove-if func list))
```

6 Defining Multiple Transformations

Given an intended code transformation, it is not difficult to provide a compiler macro to implement it. The problem is that we want to have many such transformations. In the `map-set` case, for example, we might provide transformations to deal with mapping injective functions, transformations to combine mappings with filterings, and so on.

Since compiler macros must be defined for a given operator name, we have to look at the operator arguments to identify the proper transformation. Moreover, we want to extend the available transformations dynamically. This precludes an approach based on a compiler macro that distinguishes all the possible uses because it would become unmanageable and we would need to have access to the entire compiler macro definition.

Our solution to this problem is to use a data-driven approach ([CRMM87], [AS84], [Nor92]). We use the operator name to index a table of functions. Each function receives the entire form as argument and computes the replacement code or returns the original argument to decline providing such replacement, just as in standard compiler macros.

This data-driven approach is just the first level on top of compiler macros and is implemented with the macro `define-compiler-hint`. Its syntax is as follows:

```
(define-compiler-hint operator (form)
  body*)
```

The unique parameter *form* receives the entire form being analyzed. The *body* can look at the *form*, can return it unchanged or can return a new form that will be used in place of the original form. This macro allows us to define several "compiler macros" for the same operator. It should be obvious that there's always a unique compiler macro that calls all the transformation functions. When more than one of these functions provides a transformation, the compiler macro refuses to transform the code and gives a warning instead.[5] Note that this restriction regards only the transformation of code. It is perfectly valid to have several functions *looking* at the code, but only one of them can provide a transformation.

We now look at some examples of use.

7 Code Optimization

One of the best uses for code transformation is to optimize critical parts of a program. To give an example, we present the code transformation to treat the `injective` "declaration".

```
(define-compiler-hint map-set (form)
  (if (and (consp (rest form))
           (consp (first (rest form)))
           (eq (first (first (rest form))) 'injective))
    (let ((replacement
           `(injective-map-set ,(second (first (rest form)))
                               ,@(rest (rest form)))))
      (warn "Optimizing ~S to ~S" form replacement)
      replacement)
    form))
```

The transformation allows us to keep the code readable but is, in itself, hard to read. To solve this problem, we defined some syntactic sugar to allow us to use patterns (as defined in [Nor92]) both to identify the arguments of the form and to specify a replacement. Such sugaring is provided by the macro `defoptimizer`. Using this macro, the programmer can write the previous transformation more easily as follows:

```
(defoptimizer (map-set (injective ?m) ?s)
              (injective-map-set ?m ?s))
```

Here, the symbols preceded by a "?" represent variables that can match anything. All the other symbols are treated as constants. The idea is that when the first pattern in the optimizer matches a given form, we replace the variables in the second pattern for its values and the result is used in place of the original form. It is important to note that we can use repeated pattern variables in the second pattern without risking double evaluation.

There are many more possible optimizations based on this tool. Hiding destructive functions is such an example. The use of destructive functions is seen in the non-pure functional languages as a necessary evil for the sake of efficiency. Common Lisp is no exception and provides plenty of destructive

[5] Code transformations are too important to be treated with ambiguity.

functions. Unfortunately, due mostly to historical reasons, the names of the destructive functions do not have a simple correspondence with the names of their non-destructive counterparts, which forces the programmer to know twice as many names. This problem can easily be solved by introducing a new "declaration" that allows some values to be destructively modified. This way, the use of destructive functions can be seen just as another optimization done automatically by the compiler. To this purpose, we introduce a `fresh` function/tag that declares its arguments as freshly made. The following definitions take advantage of such declaration.[6]

```
(defoptimizer (remove ?e (fresh ?l) . ?opts)
              (delete ?e ?l . ?opts))

(defoptimizer (append (fresh ?l1) ?l2)
              (nconc ?l1 ?l2))
```

In other cases, we can even avoid the declaration. Some examples follow:

```
(defoptimizer (append (mapcar ?f . ?l1s) ?l2s)
              (nconc (mapcar ?f . ?l1s) ?l2s))

(defoptimizer (remove ?e (mapcar ?f . ?ls) . ?opts)
              (delete ?e (mapcar ?f . ?ls) . ?opts))
```

8 Deprecated Code

We can take advantage of this mechanism to define other utilities. For example, it can also handle a *deprecation* mechanism by which we declare certain compound forms as deprecated and we suggest a replacement. This is very useful when we provide a new version of a library.

As an example, let us suppose that we had implemented an `insert-set` function using the old MacLisp [Pit83] function `memq`:

```
(defun insert-set (elem list-set)
  (if (memq elem list-set)
      list-set
      (cons elem list-set)))
```

Then, we decided to transport our code to Common Lisp, where `memq` doesn't exist. Most probably, we will be warned that an undefined function was being used, but our compiler would be more friendly if he could say:

```
 While compiling INSERT-SET:
Warning: The form (MEMQ ELEM LIST-SET) is deprecated.
Use (MEMBER ELEM LIST-SET :TEST #'EQ) instead.
```

Better yet, the compiler could even allow our code to run because it would have replaced it by the appropriate form.

This behavior can be accomplished with the macro `defdeprecate`, another extension to the `define-compiler-hint` macro. For the previous example, we can write the following:

[6] The consequences of writing compiler macros for functions or macros in the COMMON-LISP package are undefined. For our examples to work, some symbols of the intended package must be shadowed.

```
(defdeprecate (memq ?elem ?list)
              (member ?elem ?list :test #'eq))
```

Then, when the compiler tries to compile our previous `insert-set` function, we will be warned but the code will be able to run.

In some cases, the deprecated forms don't even have a reasonable replacement. The CLtL2 [Ste89] function `augment-environment` that manipulated compilation environments is such an example. For this case, we can change `defdeprecate` so that the definition

```
(defdeprecate (augment-environment ?env . ?args))
```

instructs the compiler to use a replacement that, when evaluated, generates a well-documented error saying that the form is obsolete. Another option is just to abort the compilation process.

9 Bad Style

Bad style is a subjective judgment. Nevertheless, there are certain coding practices which most programmers classify as bad. For example, it is generally accepted that long `car-cdr` combinations such as `cdaadr` are difficult to read and should be avoided. Using an inefficient way to compute a result when there are equally understandable but more efficient alternatives is also considered bad style.

The problem of bad style is very important for students which don't have enough experience to know which language features should be avoided. If the compiler could warn about these features the students would become better programmers sooner.

With another extension to our overloaded compiler-macros It is trivial to inform the compiler about the features that should be avoided. For that effect, we defined a macro `defdangerous` very similar to `defdeprecate` that expands into a compiler hint that never replaces the code but emits a warning whenever it sees a sign of bad coding style. The macro accepts an optional argument to be used as a suggestion.

A more or less non-controversial list of forms which raise "red flags"—i.e. which "are often symptoms of problems—even though technically most of them do happen in completely legitimate situations as well"—is presented in [NP93]. They can be defined as follows:[7]

```
(defdangerous (eval ?x))
(defdangerous (gentemp . ?args))
(defdangerous (append . ?args))
(defdangerous (ignore-errors . ?forms))
(defdangerous (cddr ?list))
(defdangerous (cdadr ?list))
(defdangerous (cdadar ?list))
```

More complex examples can be given, including a suggested rewrite, covering usual "mistakes" such as:

[7] Again, this implies that some symbols of the intended package must be shadowed.

```
(defdangerous (if ?test ?conseq)
              (when ?test ?conseq))
(defdangerous (if (not ?test) ?conseq)
              (unless ?test ?conseq))
(defdangerous (apply ?f (mapcar ?g ?l))
              (reduce ?f ?l :key ?g))
```

10 Conclusion

This article describes some problems that Common Lisp programmers have to solve in order to adapt the compilation process to their own needs. The best solution to those problems would be to use a code walker. Unfortunately, it is difficult to write a portable code walker that can correctly handle the complete Common Lisp language. A much simpler solution is to take advantage of compiler macros.

A compiler macro describes code transformations to be done during the compilation process. Usually, compiler macros are difficult to write, read and extend. Fortunately, Common Lisp has enough introspective capabilities and syntax extension mechanisms to allow easy and portable extensions to the compiler macro workings. This article presented some of these possible extensions. Three different uses were presented: (1) Code optimization, (2) Code deprecation, and (3) Code quality assessment.

On the practical side, these extensions have been used in the reengineering of a large AI system. In this system, the set is one of the fundamental data-types. There are several types of sets, but they are all manipulated using generic functions and, in particular, higher-order functions such as mappings, filterings and foldings. In spite of the increased readability of these operations over their imperative counterparts, the system was becoming less and less efficient. To solve this problem, the code was profiled and the critical parts were identified. For these parts, 6 optimizers were defined, in particular, to map injective functions and to combine maps, filters, and foldings. The most elaborate combination found was a fold after an injective map after a filter, that was obtained by cascading transformations. The optimized code for this last combination makes a single pass over the set without producing any intermediate sets.

These optimizers made 30 code transformations in the system, although they were developed to affect a much smaller number of critical parts. This way, the code remained unchanged (or with added information, such as the `injective` declaration) but speed was greatly improved. Besides, when we need to debug the system, we can switch off the optimizers.

The fact that an optimization is done via compiler macros instead of hand manipulation has the added advantage of being automatically applicable to other parts of the system.

The deprecation warnings were also very helpful during the more radical changes in the system, where operators were removed or renamed. It allowed an easier and shorter learning curve for programmers used to the old version of the system.

11 Acknowledgements

We would like to thank the support of Sofia Pinto and João Cachopo. We also want to thank Gail Anderson and the helpful comments of the reviewers. This work was partially supported by grant PRAXIS XXI BD/11071/97 and project PRAXIS XXI 2/2.1/TIT/1568/95.

References

[AI96] American National Standards Institute and Information Technology Industry Council. *American National Standard for Information Technology: programming language — Common LISP*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996. Approved December 8, 1994.

[AS84] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[CRMM87] Eugene Charniak, Christopher Riesbeck, Drew McDermott, and James Meehan. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Potomac, Maryland, second edition, 1987.

[GJ94] Andrew Gill and Simon L. Peyton Jones. Cheap deforestation in practice: An optimiser for haskell. In *IFIP 94*, pages 581–586, Hamburg, Germany, August 1994.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A programmer's guide to CLOS*. Addison-Wesley Publishing Company, Cambridge, MA, 1989.

[Nor92] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, California, 1992.

[NP93] Peter Norvig and Kent Pitman. Tutorial on good lisp programming style. Lisp Users and Vendors Conference, August 1993.

[Pit83] K. M. Pitman. *The Revised MACLISP Manual*. Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, May 1983. MIT/LCS/TR-295.

[Ste89] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, second edition edition, 89.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 90.

[Wat89] Richard C. Waters. *Common Lisp: The Language*, chapter Appendix A. Digital Press, 89.