# A Critique of Common Lisp

Rodney A. Brooks and Richard P. Gabriel
Stanford University

## Abstract

A major goal of the COMMON LISP committee was to define a Lisp language with sufficient power and generality that people would be happy to stay within its confines and thus write inherently transportable code. We argue that the resulting language definition is too large for many short-term and medium-term potential applications. In addition many parts of COMMON LISP cannot be implemented very efficiently on stock hardware. We further argue that the very generality of the design with its different efficiency profiles on different architectures works against the goal of transportability.

---

Support for this research was provided by the Air Force Office of Scientific Research under contract no.: F49620-82-C0092, and by the Defense Advanced Research Projects Agency DARPA/MDA903-80-C-0102

## 1. Introduction

The COMMON LISP language was designed by a committee of over 30 interested people from academia and industry. As the final touches were put on the specification a small core, 5 people, made most of the decisions.

The quinquevirate of final decision-makers are all well-respected and knowledgable Lisp implementors, and each one had a definite opinion on the best means of implementation. Several members of the COMMON LISP core group were working on—and believed in—Lisp implementations on micro-codable machines. The remainder of the COMMON LISP core group—including one of the authors of this paper, Dick Gabriel—was working on a highly-optimizing Lisp compiler for a supercomputer-class computer. This computer, the S-1 Mark IIA, has outstanding hardware support for numeric calculations, including complex numbers, and non-trivial support for Lisp function-calls and tags.

The main bulk of the decisions made in designing COMMON LISP were either made directly by this group or under considerable influence from this group. As it happened, there was no strong voice from Lisp implementors working on Vaxes, MC68000's, or any truly 'stock' hardware.

The original goal of the COMMON LISP committee was to specify a Lisp that was close enough to each member of a family of descendants of MacLisp that the guardians of those implementations would be willing to evolve them towards that common definition; this goal was achieved. A major subgoal of the committee was that programs written in this COMMON LISP would be source-transportable between implementations, regardless of the underlying hardware.

Every decision of the committee can be locally rationalized as the *right thing*. We believe that the sum of these decisions, however, has produced something greater than its parts; an unwieldy, overweight beast, with significant costs (especially on other than micro-codable personal Lisp engines) in compiler size and speed, in runtime performance, in programmer overhead needed to produce efficient programs, and in intellectual overload for a programmer wishing to be a proficient COMMON LISP programmer.

### 1.1 *What should have happened*

There are some desirable properties of computer languages against which any language design can be evaluated:

1. **Intellectual Conciseness**. A language should be small enough for a programmer to be able keep all of it in mind when programming. This can be achieved through regularity in the design and a disciplined approach to the introduction of primitives and new programming concepts. PASCAL is a good example of such a language, although it is perhaps too concise.

2. **Compiler Simplicity**. It should be possible to write a compiler which produces very efficient code, and it should not be too great an undertaking to write such a compiler—perhaps a man-year at the outside.

3. **Runtime Efficiency**. There should be minimal inherent runtime inefficiencies. Furthermore a small program should run in a small amount of physical memory. Small programs should not lead to inherently poor paging performance.

4. **Ease of Programming**. It should be simple and pleasant for the programmer to write efficient programs in the language. In an untyped language the programmer should not be forced to make large numbers of declarations in order to gain reasonable run-time efficiency.

The design of COMMON LISP should have heeded these design criteria or should have been prepared to do more than simply design the language. For example, regarding point 2, if the COMMON LISP designers believed that a language should be such that a simple compiler is impossible, then the COMMON LISP committee ought to take steps to produce such a compiler.

Lisp talent is in short supply; Lisp is not a very commercially lucrative business; and COMMON LISP is a complex language requiring not only a complex compiler, but a complex runtime system and many auxiliary functions. Expecting the Lisp community to provide COMMON LISP for little or no cost is not sensible; nor is it sensible to expect a software house to devote its resources to a project that does not justify its cost to that company with high profits.

Therefore, the COMMON LISP group was in the position of advocating, as a standard, a Lisp which is difficult to implement and which would not be available on many machines outside of the machines with which the implementors within the COMMON LISP were working.

The COMMON LISP design group did not heed these design criteria, nor did they provide very much help with compilers for stock hardware.

## 2. What happened

We believe COMMON LISP suffers from being designed by implementors with beliefs in what is possible, rather than by users with beliefs in what is probable. In particular we believe that the committee went wrong in the following ways:

- There was a belief (hope) that in the near future all Lisp users would have individual workstations, micro-coded for Lisp, with large amounts of physical memory. We argue otherwise in section 4.4.

- Too much emphasis was placed on the ability of micro-coding to hide the cost of type dispatching.

- The cost of invisible pointers was ignored for micro-coded machines and underestimated for stock architectures.

  Invisible pointers are like indirection bits in some architectures: Whenever a memory reference is made to a particular address, if that address contains an *invisible pointer*, the memory fetch logic fetches the contents of the memory location to which the invisible pointer points. Suppose that memory location, $m_1$, contains an invisible pointer, $ip_2$, that points to memory location $m_2$. Any memory reference that attempts to load from or store into $m_1$ will actually load from or store into $m_2$.

  *Displaced arrays* are a construct in COMMON LISP borrowed from FORTRAN. It is possible to alias two arrays to the same storage locations. That is, it is possible to state that the array elements $A(0)$–$A(j)$ are identical to the array elements $B(k)$–$B(l)$ where $A$ and $B$ are arrays and $j = l - k$. Notice that $A$ and $B$ may overlap with some elements of $B$ not part of $A$ if $k_0$.

  There are primarily two ways to implement this sharing: 1.) Invisible pointers can be used as the entries to $B$, or 2.) the two arrays could actually share storage. Assume that array $A$ has already been allocated. Then if method 2 is used, then allocating the array, $B$, could mean re-locating $A$, because there may be no room for elements 0 through $k-1$ of $B$ in front of $A$. This could result in a garbage collection to re-locate objects.

- Too many costs of the language were dismissed with the admonition that 'any good compiler' can take care of them. No one has yet written—nor is likely to without tremendous effort—a compiler that does a fraction of the tricks expected of it. At

present, the 'hope' for all COMMON LISP implementors is that an outstanding portable COMMON LISP compiler will be written and widely used. Recall that the Production Quality Compiler Compiler (PQCC) project at CMU was a research project just a few years ago; this project's goal was to produce a high-performance portable compiler for an algorithmic language—their goal has still not been achieved.

- Too much emphasis was put on existing implementations of inherently inefficient ideas—in particular MIT-style Lisp machine arrays. In COMMON LISP such arrays are called *general arrays* as contrasted with *simple vectors*. Even with special hardware the cost of structure references under these schemes is enormously more than it should be.

  Here are some examples of how a micro-coded Lisp machine could improve the performance of general arrays. COMMON LISP requires that any reference to an array be within the array bounds, and the dimensionality and bounds for each dimension are stored in an area called the *array header*. A micro-coded machine can cache the array header when the array is fetched; the array bounds can be verified in parallel with the index calculation. The semantics of COMMON LISP allow the dimensionality of arrays to change dynamically. If such an array is being access within a loop, like this:

```
(DO ((I 0 (1+ I))
     (M 0 (1+ M)))
    ((= I 101) ...)
    ...(AREF foo 54 M)...)
```

  there is no reason to presume that the array, *foo*, would retain constant through this loop. Therefore, the indexing arithmetic for *foo*, involving 54 and $M$, would need to be re-computed each time through the loop. This would require 100 re-computations of an arithmetic expression including a multiplication by 54. Micro-coded machines could do this several ways: 1.) The re-computation could be done in parallel with other operations needed to access the array, such as finding its base; or 2.) That part of the computation that remains constant through the loop could be cached somewhere, and only if the array were changed so as to invalidate this cached value would it be re-computed. In either of these cases, stock hardware would be at a disadvantage as compared with a micro-coded machine.

Method 2 requires at least on additional memory reference, a memory reference which may not necessarily be done cheaply by a cache-memory, stock-hardware system. Therefore a micro-coded Lisp machine will be able to implement this method more efficiently than a stock-hardware machine.

- Too much effort was put into extending transportability mechanisms to extremes. Examples of this are: potential numbers, branch cuts of complex numbers, and path-names. Most of these issues do not effect most users. They add to the cost for all users however.

  Quite an effort was made to define appropriate behavior for floating-point arithmetic. Programmers who do extensive floating-point computations spend a great deal of time understanding the hardware characteristics of their host computer in order to adjust their algorithms to converge or to attain sufficient accuracy. Most such programmers will expect that the languages they have on their machines will take advantage of the characteristics of those machines. In COMMON LISP a little too much control was placed on floating-point arithmetic. And certainly, although the correct behavior of a floating-point-intensive program can be attained, the performance may vary wildly.

- There was a belief that Lisp is the *correct* language for systems programming. This is a religious belief, especially in light of the fact that only a few operating systems, for example, have been written in Lisp so far.

- There was an implicit assumption that all programs should be transportable *invisibly*. That is, it was assumed that the user would never be concerned with the underlying hardware or operating system, and furthermore that the user was incapable of organizing large systems into large transportable pieces with small system dependent modules.

- There was an implicit assumption that incredibly complex existing programs, such as editors, should become source transportable with the right language definition. Some of these programs were written with large efforts towards tuning them for a particular hardware/software environment.

- A spirit of compromise left many issues **undefined**. Thus, although programs written in COMMON LISP would be transportable, it is trivial to write programs which look syntactically like they are written in COMMON LISP but which dynamically are not, and **will not work** in another COMMON LISP implementation.

An example is (**EQ** 3 3): It is not specified whether this is true or false. A second example is SUBST. SUBST is a function which takes a tree and a description of old sub-trees to be replaced by new ones. That is, let $T$ be a tree, $t$ be a subtree of $T$, *new* and *old* be trees, and $p$ be a predicate; then *new* will be substituted for $t$ in the result of SUBST if $p(t, old) \neq$ nil.

SUBST does not specify whether structure sharing occurs between the input tree and the output tree. Thus in one implementation, some structures may be EQ after a SUBST, while in another these structures will be only EQUAL.

A third example is the meaning of '‘(A ,B). Backquote (‘) is a read macro in COMMON LISP which is useful for building tree structures from templates. ‘(A ,B) is roughly equivalent to (**LIST** 'A B). COMMON LISP does not specify whether the template and the result share any substructures. Therefore, the form of what is produced by a backquote expression is implementation-dependent, and hence it is difficult to write a machine-independent pretty-printer that will print tree-structures in backquote equivalent forms the same way over new implementations.

The result was a definition of COMMON LISP which is enormous. It requires either specially micro-coded machines, a fantastically complex compiler (which relies on copious declarations from the poor incompetent user), or very slow running time. Due to the complexity of the definition any programs written in COMMON LISP which do run in other COMMON LISPs may have vastly different performance characteristics, redeemable only by gross changes to algorithms and data structure representations.

To be a bit more specific about this last point, programmers who use a particular dialect of Lisp will adjust the style of their programming to the performance profiles of that Lisp. Let us consider a conceptually valid, but actually bogus, example.[1] In one dialect the construct:

((**LAMBDA** (X) < form$_1$ > ... < form$_n$ >)
<expr>))

[1] This is bogus because a real programmer would use LET rather than either of the two alternatives discussed, and we can presume that LET will expand into the most efficient construct in any implementation.

may be much more inefficient than:

(**PROG** (X)
(**SETQ** X <expr>)
< form$_1$ >
⋮
(**RETURN** < form$_n$ >))

An unwitting programmer might believe that the performance profile which has the LAMBDA application slower than the PROG form is universal over all machines. If the programmer realizes that this may not be the case, he can write a macro which initializes variables and then performs some computations. In some implementations this macro could expand into a LAMBDA application, while in others it could expand into a PROG form. The choice would then be machine dependent.

The effect, of course, is that COMMON LISP code might be trivially transportable in terms of operational semantics, but not in terms of performance.

Thus the robe of transportability hides the fact that programmers still need to program for their own implementations, and the programs so produced may not be transportable to the degree one might be led to believe.

The true test is that, even though the bulk of COMMON LISP was reasonably well-defined in the summer of 1982, there are still no widely-available COMMON LISP's, despite many large efforts (e.g. CMU Spice project, DEC COMMON LISP project, Rutgers COMMON LISP project, Symbolics COMMON LISP project, S-1 COMMON LISP project).

### 3. Common Lisp is Good

COMMON LISP adopted many of the good ideas developed by the various MacLisp descendants. In particular, much of the good theoretical activities during the late 1960's and the entire 1970's with respect to Lisp was considered by the COMMON LISP committee.

Among the good ideas is lexical binding. A good Lisp compiler tries to code lambda-variables as lexical variables as often as it can, so that references to them are to locations on the stack via a display offset. Dynamic variables requires explicit binding and unbinding code to reflect lambda-binding, while lexical variables only require simple display updates to reflect this.

Because lexical binding is frequently what programmers intend when they write LAMBDA's, and because lexical binding leads to programs that are easier to debug and understand, lexical binding is a definite improvement over the dynamic binding that older Lisps support.

SETF generalizes assignment in a natural and understandable way. If a programmer writes a piece of code with an abstract data structure, and he wants to update that data structure, he has a stylized way of doing this that interacts well with the structure-defining forms. If he defines his abstract data structure with **defstruct**, then he is assured that SETF can be used to update his data structure without any further work by him.

Closures are a means of capturing an environment for further use. They are a means for achieving 'own' variables which can be shared among the closures in a module-like arrangement. Although closures has not been fully exploited yet, the intellectual simplification they represent is important.

Much of the integer division standards clean up the arithmetic situation, especially with respect to the rounding modes (truncate, floor, ceiling, and round), although not all hardware can support these modes easily. Nevertheless, they are easily coded by compilers.

Multiple values solve a problem that has been poorly solved by programmers—how to get several distinct pieces of information back through a function-return. Usually a programmer will either define special variables or construct a data structure to accomplish this. Because almost any hardware could support a more efficient means of accomplishing this, placing this within COMMON LISP is a good idea.

## 4. Common Lisp is Bad

### 4.1 *Generic Arithmetic*

One of the major ways that COMMON LISP falls down is that all arithmetic is generic. Although this might be a good idea for people whose use of arithmetic involves adding a couple of numbers every several hundred million instructions, it is not a good idea for those who want to write a lot of numeric code for non-Lisp machines.

Currently there is no computer that excels at both Lisp processing and numeric processing. For people who want to do number-crunching in Lisp on such machines, writing type-specific numeric code requires wordy type-declarations. To write an addition of two

integers requires writing either:

(**DECLARE** (**integer** x y))
(+ x y)

or:

(+ (**the integer** x) (**the integer** y))

Neither of these is esthetically pleasing, nor is it likely that the average Lisp programmer is going to bother all the time.

Micro-coded machines have an advantage in that they can leave until runtime decisions about what the types of objects are. For example, one could write $(+\ x\ y)$ in COMMON LISP and a micro-coded machine could determine the types of $x$ and $y$ on the fly.

### **4.2** *Arithmetic Explosion*

A large class of number types was introduced. These include 4 sizes of fixed precision floating point numbers, rational numbers, complex numbers made up of pairs of these (after long debate the committee at least restricted the types of the real and complex parts to be the same), fixed precision integers, and arbitrary precision integers. All these number types take up a large number of tag values and require extensive runtime support, as the types are orthogonal from the functions which operate on them. Besides addition, subtraction, multiplication and a few varieties of division, there are trigonometric functions, logarithmics, exponentials, hyperbolic functions, etc.

In addition, the inclusion of complex numbers has opened a Pandora's box of confusion and controversy surrounding the selection of branch cuts. The current COMMON LISP definition follows a proposed, but yet to be accepted, APL definition. Mathematicians do not have definitions for the various branch cuts, but they tend to define them a particular way depending on the situation.

### **4.3** *Function Calls Too Expensive*

Function calls are the hallmark of Lisp. Everyone expects that Lisp code will be sprinkled with many function calls. With the argument-passing styles defined by COMMON LISP, function calling must be expensive on most stock hardware. COMMON LISP allows *optional*, *keyword*, and *&rest* arguments (along with a few others). Optional arguments are arguments that may or may not be supplied; if they are not supplied, then they are

defaulted by the called function. Keyword arguments are useful for those times when one cannot remember the order of arguments. For instance, MISMATCH takes two sequences, a flag stating whether to scan from left to right or from right to left, and starting and ending indices for both sequences. Oh, and there's a test that is used to test the mismatch. Who could remember the order, so one can write:

(**MISMATCH** $S_1$ $S_2$ :from-end t :test predp
:end1 2 :start2 baz)

where the symbols prefixed by (:) are keyword names and the symbols after those keyword names are the arguments associated with those keywords. And one more thing: it also is possible to supply a not-test instead of the test to not mismatch using the keyword *:test-not*.

An &*rest* argument will be bound to a list of all of the arguments supplied that are neither bound to required parameters nor to optional parameters. Included in this list are all keyword-name/keyword-value pairs which appear to the right of the &*rest* parameter in the defining lambda-list. Thus a programmer is able to supply as many extra arguments as he wants to functions defined with an &*rest* argument in its lambda-list.

We will now look at a particular function, SUBST—and how it might be defined in COMMON LISP—to illustrate the expense of function calls and the advantage that micro-coded machines might have over stock hardware.

Having briefly looked at SUBST before, let us look at it a little more closely. The COMMON LISP definition of SUBST states that it takes a *new* tree, an *old* tree, a *:test* or a *:test-not* keyword argument (but not both), and a *:key* keyword argument. We have explained all of these arguments except for *:key*. Recall that the *:test* keyword argument is a predicate that is used to decided whether to perform a substitution on a particular subtree. The *:key* argument is a function of one argument which takes a subtree as its argument, and returns an object to which *old* and *:test* are applied. For example,

(**SUBST** 'here 'b '((a b c)(a b c)) :key #'cadr)

will return (HERE HERE) because we have asked SUBST to substitute HERE for all lists in the tree with B as their second element.

Here is a possible definition for SUBST:

```
1  (defun subst (new old tree &rest x
2                &key test test-not key)
3   (cond ((satisfies-test old tree :test test
4                                :test-not test-not
5                                :key key)
6          new)
7         ((atom tree) tree)
8         (t (let ((a (apply #'subst
9                            old new
10                           (car tree) x))
11                (d (apply #'subst
12                           old new
13                           (cdr tree) x)))
14               (if (and (eql a (car tree))
15                        (eql d (cdr tree)))
16                   tree
17                   (cons a d))))))
```

Notice that if a subtree is not altered it is not copied (lines 14–17). The interesting part is the *&rest* argument, $x$; it is used to do the recursive calls to SUBST in lines 8–13. In those lines we APPLY SUBST, and $x$ holds all of the keyword arguments, which are then passed on to SUBST. We use APPLY so that SUBST does not need to figure out which keyword arguments were actually passed and so that SUBST does not have to default any unsupplied arguments.

What happens when SUBST is invoked? First it checks to see how many arguments were supplied; if too few were supplied, that is an error—SUBST expects 3 required arguments. Next it must associate the keyword arguments with the correct lambda-variables. If no keyword arguments were supplied, then this amounts to a check of the number of keyword arguments passed. Finally it must construct the list of arguments for the *&rest* argument, $x$. If some keyword arguments are passed, then this requires a list to be constructed.

Constructing a list usually requires allocating storage in the heap. To avoid such CONSing for each function call, an implementation can choose to allocate this list on the

stack, as a vector. That is, the arguments to SUBST will be supplied (in most implementations) on the stack. If the *&rest* argument is never returned as part of the value of a function, then there is no need to actually CONS up that list. However, to treat the items on the stack as a list requires either a very fancy flow analysis by the compiler or else CDR-coding.

Supporting CDR-coding is easily done with a micro-coded machine if there are enough bits in the word to accomodate the CDR-code bit. On the other hand, if the *&rest* argument list is to be returned as part of the value of the function, then a CONS in the heap must be done sooner or later. This can either be accomplished with extensive compiler analysis or with special hardware and micro-code using that hardware.

If a programmer uses SUBST without any of the keywords, then the overhead required is to check the number of required arguments, to check whether any keywords were supplied, and to construct a null list for $x$. Checking the number of required arguments supplied is necessary for all functions, but the other checks are not. If the programmer uses the COMMON LISP SUBST routine, as he is encouraged to do, then he faces this overhead.

If the programmer uses some keywords, then the overhead is very much higher. If the call to SUBST is simply

(**SUBST** new old tree :test #'baz)

then one might hope that the association of keywords supplied with appropriate lambda-variables could be eliminated at compile-time, but this could only happen if there are entries for each combination of keyword arguments or if there were separate functions for each such combination.

Thus, the programmer faces either a performance or a space penalty for this generality on stock hardware implementations.

### 4.4 *Computer Size*

COMMON LISP requires a large computer memory. To support only the function names and keyword names alone requires many kilobytes. To support the many copies of keyword-taking functions to improve the speed of COMMON LISP on stock hardware would require megabytes of memory. The compiled code for functions is large because of the amount of function-call overhead at the start to accomodate both possible keyword and optional arguments and also the inline type dispatches.

Memory costs, while declining, still account for most of the cost of a large computer system. For high-performance computers, it is essentially the case that one buys the memory and gets the CPU free.

In the next few years much research will be done with distributed computing. If each computing node needs to cost $100,000 and up, very large networks of them will not be built.

### 4.5 *Good Ideas Corrupted*

Another common theme of the COMMON LISP design is to take a very good simple idea and to over-generalize it. This has the effect of complicating both the implementation and the intellectual effort required to understand and use the idea.

FORMAT is a way to do stylized printing—the user specifies a template with empty slots, he specifies the contents of the slots, and FORMAT prints the template with the slots filled in. The idea is borrowed from the FORTRAN FORMAT statement.

Here is an example of a FORMAT statement:

```
(FORMAT NIL
        "~ @(~ @[~ R ~] ~ ^ ~ A. ~)"
        23 "losers").
```

Try to guess what it does—you'll find out after the next paragraph.

Although this idea is simple, COMMON LISP took FORMAT to its ultimate generalization. With FORMAT it is possible to do iteration, conditional expressions, justification, non-local returns within iterations, and pluralization of words according to the values for the slots. The language used by FORMAT is not Lisp, but a language that uses strings with special escape characters that either represent substitutions, substitutions with specific printing control, or commands. This language is somewhat large by itself, but the main problem is that it isn't Lisp. Many language designers believe that the diction for expressing similar constructions should be similar. With FORMAT the COMMON LISP committee abandoned this goal.

The sample FORMAT statement prints:

**Twenty-three losers.**

To a lesser extent the same charge can be leveled at SETF. SETF is a generalization of SETQ. One can SETQ the value of a variable like this:

(**SETQ** X <value>)

If one wants to alter the CAR of a CONS cell, one must write:

(**RPLACA** <cell> <value>)

SETF allows one to say, instead:

(**SETF** (**CAR** <cell>) <value>)

SETF works on all of the data structures in COMMON LISP, for instance, arrays, property lists, vectors, and user-defined structures. The principle is that if part of a data structure can be accessed using some form, that SETF ought to be able to change the value of that part.

So far, so good. Now note that often a programmer will define a data structure as a complex of macros: Some macros create a data structure and others access them. When the user chooses to define a data structure with a complex of macros, he is usually content to define special value-altering macros.

SETF has been generalized so that it has extremely complex mechanisms for describing the semantics of user macros to SETF. This way SETF can be used to alter the values of user-defined data structures. For such complex macros it makes little sense to try to force them into the mold of being a data substructure reference.

### **4.6** *Sequences*

The COMMON LISP definition introduces the notion of *generic sequence*. A sequence can be represented as a list or a one-dimensional array. Many of the standard Lisp functions (e.g., LENGTH, REVERSE, NREVERSE, DELETE, SORT) have been re-defined to work on any sequence, and a whole host (about 24 base functions each with a large number of keyword modifiers) of new functions.

Lists and one-dimensional arrays have fundamentally different access time characteristics (constant for arrays, linear for lists). They have even greater fundamentally different

characteristics for being shortened and lengthened. We believe that trying to hide these differences behind generic function names will be a source of programmer overhead and will lead to inefficient programs.

As generic functions, the efficiency of dispatch on argument type for all the sequence functions needs to rely on either 1.) detailed declarations which implies the need for greater programmer overhead and compile time use of those declarations with associated increased compiler complexity or 2.) a cost to runtime performance, especially on stock hardware.

In addition the sequence functions are the primary example of keyword–combination explosion, with the associated costs on compiler complexity and possibly paging performance at runtime. The large number of function/keyword possibilities is itself a source of programmer overhead.

### **4.7** *Arrays*

COMMON LISP arrays are incredibly complex. As a concession to non-micro-coded machines, vectors, simple vectors, and simple strings were included as types. The resulting tangle of types (all simple vectors are vectors, all simple strings are strings, all strings are vectors, but no simple strings are simple vectors) is clearly a political compromise and not something that would exist after careful design.

Simple vectors are a fixed-size set of pointers in contiguous locations; they are called *simple* because they don't have fill pointers, are not displaced, and are not adjustable in size. Cleary such vectors can be implemented much more efficiently on stock hardware than can general arrays. Unfortunately the default assumption is that a vector is not simple unless explicitly declared simple—making the programmer using stock hardware have to work harder to acheive efficiency, and making programs which work well on micro-coded machines perform poorly on stock hardware.

### **4.8** *Subsets of* COMMON LISP

We believe that these points argue for a standard subset of COMMON LISP, especially for small, stock hardware implementations. The COMMON LISP committee has not defined a standard subset, and it appears that at least one subset, if not many such subsets, will appear. Unless these subsets are co-ordinated and widely adopted, then we will see one of the major goals of COMMON LISP—transportability—lost.

## 5. Conclusions

Our goal in this paper is to discuss the shortcomings of COMMON LISP. In doing so we do not want to only criticize COMMON LISP: Both authors agree that the world is better having a COMMON LISP than not. We wish only that COMMON LISP could have been designed with more, varied machines in mind.

We fear that COMMON LISP will not live up to some of its goals because of these design decisions. Perhaps good compilers will come along and save the day, but there are simply not enough good Lisp hackers around to do the jobs needed for the rewards offered in the time desired.

## References

[**Brooks 1982**] Brooks, Rodney A., Gabriel, Richard P., and Steele, Guy L. *S-1 Common Lisp Implementation* in **Proceedings 1982 ACM Symposium on Lisp and Functional Programming**, Pittsburgh, Pa, August, 108–113.

[**CL Committee 1984**] Many People, *Mail messages concerning the development of Common Lisp* in **COMMON.∗[COM,LSP]@SAIL**.

[**Knuth 1965**] Knuth, Donald E. *Fourteen Hallmarks of a Good Programming Language*, Unpublished.

[**Steele 1984**] Steele, Guy Lewis Jr. et. al. **Common Lisp Reference Manual**, Digital Press, 1984.

[**Weinreb 1981**] Weinreb, Daniel, and Moon, David. **LISP Machine Manual**, Fourth Edition. Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, July 1981.

[**Wichman 1984**] Wichman, Brian A. *Is ADA Too Big? A Designer Answers the Critics* in **Communications of the ACM**, (27)98–103.