Software Fault Prevention by Language Choice: Why C is Not My Favorite Language

RICHARD FATEMAN

Computer Science Division
Electrical Engineering and Computer Sciences Department
University of California—Berkeley
Berkeley, California 94720-1776
USA
fateman@cs.berkeley.edu

Abstract

How much does the choice of a programming language influence the prevalence of bugs in the resulting code? It seems obvious that at the level at which individuals write new programs, a change of language can eliminate whole classes of errors, or make them possible. With few exceptions, recent literature on the engineering of large software systems seems to neglect language choice as a factor in overall quality metrics. As a point of comparison we review some interesting recent work which implicitly assumes a program must be written in C. We speculate on how reliability might be affected by changing the language, in particular if we were to use ANSI Common Lisp.

1. Introduction and Background 168
2. Why Use C? 169
3. Why Does Lisp Differ from C? 171
4. Root Causes of Flaws: A Lisp Perspective 173
   4.1 Logic Flaws 173
   4.2 Interface Flaws 178
   4.3 Maintainability Flaws 179
5. Arguments against Lisp, and Responses 179
6. But Why is C Used by Lisp Implementors? 185
7. Conclusion 185
   Appendix 1: Cost of Garbage Collection 186
   Appendix 2: Isn't C free? 187
   Acknowledgments and Disclaimers 187
   References 188

ADVANCES IN COMPUTERS, VOL. 56
ISBN 0-12-012156-5

1. Introduction and Background

In a recent paper, Yu [1] describes the kinds of errors committed by coders working on Lucent Technologies advanced 5ESS switching system. This system's reliability is now dependent on the correct functioning of several million lines of source code.[1]

Yu not only categorizes the errors, but enumerates within some categories the technical guidelines developed to overcome problems.

Yu's paper's advice mirrors, in some respects, the recommendations in Maguire's *Writing Solid Code* [2], a book brought to my attention several years ago for source material in a software engineering undergraduate course. This genial book explains techniques for avoiding pitfalls in programming in C, and contains valuable advice for intermediate or advanced C language programmers. It is reminiscent of (and acknowledges a debt to) Kernighan and Plauger's *Elements of Programming Style* [3]. Maguire's excellent lessons were gleaned from Microsoft's experience developing "bug-free C programs" and are provided as anecdotes and condensed into pithy good rules.

The key emphasis in Yu's paper as well as Maguire's book is that many program problems are preventable by individual programmers or "development engineers" and that strengthening their design and programming capabilities will prevent errors in the first place.

Yet the important question that Yu and his team, as well as Maguire, never address is this simple one: "Is the C programming language appropriate for the task at hand?"

We, perhaps naively, assume that the task is not merely "write a program that does X." It should be something along the lines of

> Write a correct, robust, readable, documented program to do X. The program should be written so that it that can be modified, extended, or re-used in the future by the original author or others. It is good (and in some cases vital) that it demonstrate efficiency at run-time in time and space, machine independence, ease of debugging, etc.

The task might also include incidental constraints like "Complete the program by Tuesday." For obvious reasons, for purposes of this paper we are assuming that the task constraints do not include "You have no choice: it must be written in C." *It is unfortunate that this constraint is implicit in much of what has been*

[1]It would be foolhardy to rely on the perfection of such a large and changing body of code. In fact, the code probably does not function correctly. A strategy to keep it running is to interrupt it perhaps 50 times a second. During these interruptions checks and repairs are made on the consistency of data structures before allowing the resumption of normal processing. Without such checks it is estimated that these systems would crash in a matter of hours.

written, and that for many programmers and writers about programming it is nearly subconscious: so much so that problems that appear only in C are apparently thought to be inherent in programming.

While the C programming language has many virtues, it seems that the forced selection of this language directly causes many of the problems cited by Yu, specifically when the goal is to produce reliable programs in a natural way.

Many of us are well aware that the Department of Defense made the determination that for building reliable real-time embedded programs, C was not a suitable language. The resulting engineering process gave birth to the language Ada.[2] Ada has not caught on in civilian programming for a variety of reasons. Rather than examining the C/Ada relationship, here we will look primarily at a comparison of C to Common Lisp, a language we think has many lessons for how to support software engineering in the large. While Common Lisp is widely used and highly regarded in certain niches, it is not a mainstream programming language.

2. Why Use C?

C evolved out of the expressed need to write programs to implement in a moderately high-level language the vast majority of operating systems functionality for the UNIX operating system for the 16-bit PDP-11 computer. It was in turn based on the language "B" used for UNIX on the PDP-7 computer. The intent, at least after the initial implementation, was expanded to try to make this code nearly machine independent, despite the numerous PDP idioms that show through.

UNIX and C together have evolved and spread to many different computer architectures. C in particular has also generated successor languages in which one usually sees many of the original choices that were incorporated in C, combining ideas of data structuring (object oriented), economy of expression, and program control flow, with a particular syntactic style.

The human/computer design balance in which C and UNIX originated probably made good sense in the early 1970s on many computers. C even looked *avant garde* in 1978 when Digital Equipment Corp's VAX 11/780 computer became a popular successor to the PDP-11. The manufacturer's operating system was written in a mixture of lower-level languages (Assembler, BLISS) and so C seemed "high level." In fact, DEC (now Compaq)'s Alpha OPEN-VMS software continues to include substantial BLISS source code.

[2]How much better would the situation be if 5ESS were written in Ada? That would be another paper, I think.

C worked well when computers were far more expensive than today: a standard configuration VAX of that time would be a 256-kB, 1-MIPS machine with optional (extra cost) floating-point arithmetic. In 1978 such a machine supported teams of programmers, a screen-oriented editor was a novelty, and at UC—Berkeley, much of the Computer Science research program was supported on just one machine.

C has certainly endured, and this is a tribute to the positive qualities of the design: it continues to occupy a certain balance between programming effort and efficiency, portability versus substantial access to the underlying machine mechanisms. Even its strongest advocates must acknowledge that C is not "optimal": certainly smaller code could be provided with byte codes, and faster code by programming in assembler.

A strong practical support for C is the fact that it is nearly universally implemented on computing platforms, being available on many of them in a refined development environment. Add to these rationales those provided by employers in choosing C: There is a relative abundance of C programmers coming from school. There is an expectation that established programmers will know C. In fact this contributed to the design of Java, whose syntax is based in part on the assumption that programmers would find a C-like syntax comfortable.

However, times have changed. Today we expect a single programmer to command a machine 400 times larger in memory, and 400 times faster than that in 1978. Why should we expect a language design oriented to relatively small code size, oriented toward an environment in which simplicity of design dominates robustness, to continue to be an appropriate choice?

Why is it used at Berkeley? Many faculty know C fairly well. We often use UNIX in some form, and even Microsoft Windows or Macintosh systems provide C. C is "good enough" for many student projects. It is at a low-enough level that the transition from C to assembler can be used easily in a tutorial fashion to demonstrate the relationship of higher-level language notions to their implementation at the level of machine architecture. By being the implementation language for the UNIX operating system, additional programming in C provides access to nearly every feature short of those few machine-dependent concepts available only to the assembly-language programmer.

Unfortunately, class projects lead students to believe that this is the way it should be, even though nearly all aspects of the project violate real-world programming task requirements. How many real projects have perfectly defined and idealized requirements specified in "the assignment"? How many projects would be deemed complete and given a passing grade when they show first signs of producing a correct answer? A probable typical student project is unreliable, under-designed, under-documented "demoware." It's also written in C. While the real world leaves behind so many aspects of the student project, why should the programming language still be the same?

While C++ as well as Java and class libraries have changed the outlook of programmers in dealing with complexity through object orientation (and Java has taken a major positive step in automatic storage allocation), there are still areas of concern: these languages seem to be major sources of inefficiency in programming effort, ultimately reflected in the difficulty of using them in building correct large systems.

3. Why Does Lisp Differ from C?

Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden implementation of half of Common Lisp.
— Philip Greenspun, 10th rule of programming

Today's Common Lisp is descended from Lisp 1.5 of 1960, one of the oldest languages in use today,[3] and yet Common Lisp is in some respects one of the newest languages. Today it is defined as a 1994 ANSI standard (X3J13).

Most of the evolution since 1960 was driven by programmers *optimizing their own productivity environment. Compared to commercial installations of the time, little emphasis was placed on efficient batch processing. Instead, memory and computation resources were deployed specifically for programmer support.* This meant time-sharing when others were using batch. This meant single-user workstations when others were using time-sharing. This meant graphical interfaces when others were using text-line interfaces. In a typical development artificial intelligence project, one or a few programmers would set to the task of building a fast prototype to try out ideas. Often this required the building of a kind of new application-specific "language" on top of the Lisp foundation.[4] The notion of reliability was rarely a goal, typically being less important than flexibility,[5] but tools for debugging were always a very high priority. In academia and in industrial research laboratories, often the most advanced programming environments were developed on Lisp systems, including those at Xerox, BBN, Symbolics, MIT, Stanford, Carnegie-Mellon, and here at UC—Berkeley.

[3]Only the Fortran heritage is longer.

[4]The tradition of bottom-up programming in functional languages means that the components tend to be testable in relative isolation, they are more likely to be reusable, and this leads to a greater level of flexibility when the higher-level functionality is implemented. Often this is combined with a top-down design philosophy.

[5]The ease of prototyping in a language is key: in "Accelerating Hindsight, Lisp as a Vehicle for Rapid Prototyping" *Lisp Pointers*, 7, 1–2, Jan–Jun 1997, Kent Pitman articulates the reasons. In brief, early review and discovery of problems lead to a rapid realization of what needs to be fixed. Since hindsight is "20–20" this early feedback leads to better results. In the traditional, but now usually disregarded model of software development (the waterfall model) critical problems are discovered rather late in the development cycle.

In my opinion this evolution has matured to support the tasks of design and programming addressed professionally.[6] In our experience, a C programmer first writing in Lisp will use only that subset of tools already existing in C, and thus may initially write rather poor (nonidiomatic) Lisp. A fair comparison of programming languages requires somewhat more than finding the common subset of them. We believe that reaching a given level of productivity and proficiency can be aided by today's Lisp language design.

This problem of writing in a familiar form can be observed more generally. In a Web-based tutorial on Lisp Robert Strandh of the University of Bordeaux[7] expands upon the common observation that students (and indeed others) are often inefficient in their work. Instead of learning how to use tools properly, they flail ineffectively with what they already know. He suggests that people can be divided into *perfection-oriented* and *performance-oriented* groups:

> The people in the category perfection-oriented have a natural intellectual curiosity. They are constantly searching for better ways of doing things, new methods, new tools. They search for perfection, but they take pleasure in the search itself, knowing perfectly well that perfection can not be accomplished. To the people in this category, failure is a normal part of the strive for perfection. In fact, failure gives a deeper understanding of *why* a particular path was unsuccessful, making it possible to avoid similar paths in the future.
>
> The people in the category performance-oriented, on the contrary, do not at all strive for perfection. Instead they have a need to achieve performance *immediately*. Such performance leaves no time for intellectual curiosity. Instead, techniques already known to them must be applied to solve problems. To these people, failure is a disaster whose sole feature is to harm instant performance. Similarly, learning represents the possibility of failure and must thus be avoided if possible. To the people in this category, knowledge in other people also represents a threat. As long as everybody around them use tools, techniques, and methods that they themselves know, they can count on outperforming these other people. But when the people around them start learning different, perhaps better, ways, they must defend themselves. Other people having other knowledge might require learning to keep up with performance, and learning, as we pointed out, increases the risk of failure. One possibility for these people is to discredit other people's knowledge. If done well, it would eliminate the need for the extra effort to learn, which would fit very well with their objectives.

[6]Lisp can also be used to great advantage by novices: for example, a simplified version of Lisp (Scheme) is a popular pedagogical language. This is not our concern here.

[7]Available at `http://dept-info.labri.u-bordeaux.fr/~strandh/Teaching/MTP/Common/Strandh-Tutorial/Dir-symbolic.html`.

Of course this is a simplification, and individuals normally contain aspects of each category; as an example, a perfectionist mathematician may be performance-oriented when it comes to computing.

4. Root Causes of Flaws: A Lisp Perspective

Our thesis is that the C programming language itself contributes to the pervasiveness and subtlety of programming flaws, and that the use of Common Lisp would benefit the program implementation and maintenance effort.

Yu's paper [1] on problems in the 5ESS system indicates 10 major coding fault areas (and an extra "other" category) and gives proposed countermeasures. Not all the countermeasures are easily applied, regardless of language. In particular, how is one to achieve "better thinking" or "more time" or "better education"? Such sections we will not address here.

We will look at the other coding fault areas given in each of the remaining major sections. We emphasize, along with Yu, three of these that account for more than 50% of the total. We spend most of our space on the first of these, partly to keep this paper from ballooning out of reasonable length.

4.1 Logic Flaws

The largest area was logic flaws, accounting for 19.8% of the faults encountered. These are errors that occur when the control logic causes a branch to an incorrect part of the program or logically computes an incorrect value.

How many of these are easily (we are tempted to say, automatically) corrected by using a language better adapted than C to writing more usually correct programs? (We give examples in Lisp when appropriate.)

4.1.1 L1. Initialize All Variables before Use

This is done automatically by Lisp for ordinary scalar local variables when created. Initial default values can be specified for every array. Declarations and initializations of global variables can be done via `defvar`, `defconstant`, `defparameter` depending on how "constant" they are. Arrays can be initialized as well.

4.1.2 L2. Control Flow of Break and Continue Statements

Conditional control flow with `if`, `case`, and `cond` is clearly indicated in correctly indented code, and Lisp code is correctly indented in the normal development environment. The traditional complaint of non-Lisp programmers that there are too many parentheses is simply not an issue: A programmer types as many parentheses as necessary, watching a suitable editor "flashing" the balancing parenthesis of a construct, and indenting as necessary. Errors in structure are easily detected. Beyond this, one can do far better with proactive editor assistance, as suggested by Fry [4], in making sure that coding reflects the expected control flow.

Presumably one of the C problems being cited by Yu is that `break` and `continue` statements can occur in expressions deeply nested inside the `switch` or `for` statements to which they refer. Thus you end up with what amounts to a `goto` statement but one whose target is not apparent. Worse yet someone editing the code may not see your `break` or `continue` statement and surround it with another `switch` or `for` statement, thus inadvertently changing the target.

Lisp has a similar problem with the `return` form statement, which can appear inside various constructions (officially those that have a "block" body: `let`, `let*`, `prog`, `do`, `do*` `dotimes`, `dolist` among others). With a deeply nested `return` you may not be able to tell which form it's returning from (especially with user-defined macros surrounding the form). It's good Lisp practice in any situation in which it is not entirely obvious what the target of a return is to use the named `block` statement and convert the `return` to an explicit `return-from` with the label of the block.

With C if you want to be sure of getting to some place you must use the `goto` statement, with all the baggage that that might entail.

4.1.3 L3. Check C operator associativity and precedence

The first example given in Yu's paper (simplified here) was `if (x->y.z & r==s) ...`, which should have been `if ((x->y.z & r)==s) ....` This would be expressed in Lisp approximately as

```
(if (equal (logand (slot-value (slot-value x y) z)
            r)
           s) ...)
```

where we assume a corresponding encoding of structures in C and Lisp, and that `x` is an object of type `y`. There are neater ways of encoding structures and accessors that would look different from the use of `slot-value`, so this is only an approximation.

Other examples in Yu's paper include bugs based on a programmer's misunderstanding of the order of various operations with respect to incrementation (and of course the implicit agreement of other programmers who have walked through the code as to the misinterpretation): `*n++` which should have been `(*n)++`.

Of course much of this is (he argues) bad practice in C coding: even if the programmer had gotten it right the first time, the next human reader of the code might misunderstand it. In fact, one could argue that in all possible places a pair of parentheses, even those that are unnecessary, should be inserted in properly engineered code.

This is a particularly irksome language issue. Note that the K&R C programming language has 15 precedence levels, of which 3 classes of operator are right-to-left associative. The symbol * occurs in TWO levels, the characters + and > in various combinations each occur in THREE distinct levels, and the character - occurs in FOUR levels.

By contrast, all operators in Lisp are delimited prefix operators with no associativity or precedence. Even C's `a * b + c` which might not involve much mystery is arguably clearer as Lisp's `(+ (* a b) c)`. If you doubt such clarity helps, ask a C programmer to explain: `a * * b + + + c`. How sure?

4.1.4 *L4. Ensure Loop Boundaries Are Correct and L5. Do Not Overindex Arrays*

Lisp has no perfect solution because off-by-one errors cannot be removed syntactically in general. However, it is possible via standard looping constructs to make it clear that the number of iterations corresponds to the number of elements in a set or elements in an array (Common Lisp has the notion of a sequence that includes lists and arrays. Some constructs are available that work on either data structure.): `(dotimes (i 5)(f i))` computes `(f 0)` through `(f 4)`. If `A` is any sequence (list, array), then `(dotimes (i (length A)) ..(elt A i)..)` will refer to each element in `A`.

For sets represented as lists, there are alternative forms of iteration such as `(dolist (i '("hello" "goodbye")) (g i))`.

There is also the more recently introduced modern functional mapping construct `(map)` which takes one argument to specify the result type, a function *f* of *n* arguments to be applied, and *n* sequences. Thus `(map 'array #'+ #(1 2 3) #(4 5 6))` produces `#(5 7 9)`.

Numerous functions are provided to search, select, sort, and operate on sequences. The meaning of the operation does not require the decoding of a potentially unfamiliar and possibly erroneous C idiom. Instead it relies on the understanding of a function on sequences such as `remove-duplicates`.

While we are talking about sequences, we should observe that other storage types are available in the language: there is a hash-table primitive data type.

Other kinds of logical termination conditions can be imposed by additional iteration constructs. There are several common macro packages that seek to make looping "easier" by interspersing key words like `until` or `unless` with accumulation operations like `sum` or `collect`.

4.1.5 L6. Ensure Value of Variables Is Not Truncated

In C if a wide value (say 16 bits) is assigned to a narrow storage spot, some bits are lost, apparently without being noticed. This cannot happen in Lisp in assigning values to variables since variables will ordinarily take on "any" values. That is, `(setf x y)` does not ever change or truncate `y`. If one stores a value in an object defined using CLOS,[8] then one has rather substantial freedom in checking any attributes of the value being deposited by the `setf` method, and if it matters, this should certainly be checked. In properly engineered code it is likely that one would not be satisfied with a type check, but plausible ranges or other assertions might be checked as well. This could be done (as they say, "transparently") because the process of setting values can be overloaded. Although `setf` can be compiled down to a single instruction in the simplest case, it is not *confined* to be such a simplistic implementation as "=" in C.

At one time I would have felt compelled to defend some level of overhead in CLOS as being a reasonable price to pay for full-fledged object orientation. Given the advent of C++ and Java, it seems the battle has been fought elsewhere and apparently won.[9]

4.1.6 L7. Reference Pointer Variables Correctly, L8. Check Pointer Arithmetic, and L9. Ensure Logical OR and AND Tests are Correct

Yu does not give an example, but many C programs have such bugs when first written, and detecting them is painful. Lisp does not have "pointer variables," and it does not do pointer arithmetic, so incorrectly incrementing pointers does not happen. Dereferencing pointers cannot be done incorrectly because it is not done at all.

[8]The Common Lisp object system.

[9]The object system in Common Lisp is more general than that in Java, C++, Smalltalk, and Simula. Among other features, CLOS has multiple dispatch, meaning that the operation being invoked can be selected using the types of *all* of its arguments. It also supports multiple inheritance, available in C++ but in Java only via interfaces. Some features of CLOS are surprising: dynamic class definition allows one to (for example) add slots and methods to a class after instantiating some elements! Common Lisp also has its meta object protocol (MOP), which can be used to build both more targeted and efficient or more elaborate and general object systems.

Logical operations on bitstrings are done using logand, logior, and logxor, and Lisp provides a full selection of logical bit operations. Truth-valued decisions can be made with and and or as well as not. These are all delimited prefix operators. It is unlikely to be confused with the masking operations, since they have substantially different names, not formed by stuttering one character. C's use of any nonzero value as a Boolean true appeals has limited appeal if you are concerned with readability. In Lisp the value NIL is the only false value.

4.1.7 L10. Assignment and Equal Operators

C uses the easily confused = and == syntax. Lisp uses the rather distinct setf and equal operations. In fact there are some alternatives to equal depending upon what is being compared. The nuances of eq and eqn are relevant for optimization, but probably not of concern here.

4.1.8 L11. Ensure Bit Field Data Types Are Unsigned or Enum

Lisp has bit strings; an enumerated data type can be defined, but would probably be handled via abstraction. Small sets are often represented by lists, but could be stored in hash tables or trees or other structures, depending on efficiency criteria.

4.1.9 L12. Use Logical AND and Mask Operators as Intended

This probably refers to the confusing syntactic notation for masking operations in C. In Lisp this is done by the usual parenthesized prefix. While this does not entirely prevent misunderstanding, prefix and and logand are more distinct than C's infix & and &&.

4.1.10 L13. Check Preprocessor Conditionals

There is no example of preprocessor conditional errors in Yu's paper, but we can imagine that this is partly an extension of C's confusing conditionals applied to the preprocessing stage. Conditional code expansion based on the environments at compile-time and source-file-read-time is provided in Lisp through various macro capabilities. The potential confusion of multiple configurations can be a source of errors in any case, and we're not sure Lisp has a lock on a fix here.

4.1.11 L14. Check Comment Delimiters

Lisp has several kinds. Since my comments are displayed in the editor in a color different from that of program text, it is hard to confuse them on the screen. I do not understand why this elementary tool has somehow been lost in the 5ESS programmers' environment. Perhaps monochromatic hardcopy is the primary source code repository, and comments are not displayed in a distinct manner. One might think that the use of a particularly dull editor, one unable to tell that it was displaying comments or program, could be to blame. In any case, in C it's hard to see where a comment ends in large comments, and the comments in C don't nest—you can't easily comment out a function that itself contains a comment. Lisp has comments "to the end of the line" as well as bracketing comments.

4.1.12 L15. Checking the Sign of Unsigned Variables

There are none in Lisp. Variables don't have signs. Numeric values have signs, but asking for the sign of a bitstring or some other encoding that is not a number is an error.

4.1.13 L16. Uses 5ESS Switch Defined Variables Properly

There would likely be some variation of this issue in any implementation language.

4.1.14 L17. Use Cast Cautiously

Yu's paper describes bugs caused by number conversion/truncation using casts. Why use cast at all? Are we saving bits? Presumably the storage of data in records would be done by an assignment, or perhaps a write into a file. Basic data types in Lisp are manifest. One can ask of a value "are you an integer?" and then use it appropriately. One can also produce a new value by coercion: say of an integer to a character. One cannot refer to a primitive value of one type through storage equivalence as though it were another in legal code. If cast in C (to support untagged union types) is used to squeeze the most out of storage, it should make any programmer think twice: it's not a great idea in the first place, but at least one would hope that proper support of data abstractions as well as the use of explicit tags would reduce this source of error.

4.2 Interface Flaws

This class of flaws consists of apparent disagreements between function definitions and their uses. The caller assumes an argument is a pointer, but the function disagrees. A consequence of some such disagreements can be that an erroneously passed copy of a large structure may overflow a stack. Many of these errors would not occur in Lisp, although there is still the possibility of using arguments in the wrong order, or simply calling the wrong function. Rather than insisting that functions with no return values be declared of return type void, it has been historically convenient in Lisp to decide that every function returns a value; if nothing else comes to mind, perhaps a condition code. Common Lisp allows multiple returned values (any number including 0 values), which removes the necessity for "in/out" or "output parameters" in argument lists. We discuss this "functional" orientation again when we provide arguments against Lisp, but for now, let us say that Lisp allows interfaces that are rather more versatile, allowing optional, keyword, and default arguments. Argument-count checking can be done at compile time and also enforced at runtime.

4.3 Maintainability Flaws

Major flaws in maintainability seem to include insistence on extra parentheses and bracketing to guard against the case of insertion of statements breaking control flow. That is, in C one should write `if a {b;}` just in case a statement is later inserted before or after the statement `b`. The otherwise correct `if a b;` is not as easily maintained. The Lisp `cond` has no such problem.

5. Arguments against Lisp, and Responses

We have heard the argument that Lisp is slow because it is interpreted, or is bad because it uses a garbage collector (GC) for storage reallocation. This is hardly tenable when Java is being promoted as a substitute for C, or when heuristic garbage collectors are promoted for C or C++.[10]

The pauses that plagued old Lisp systems during GC are no longer likely: a commercial Lisp garbage collector is likely to be based on a quite efficient "generational" scavenger. In an interactive environment, time-sharing delays, network transmission delays, and computation time are likely to be of the same general time scale as pauses for GC. Real-time collectors (say, restricted to 10-ms time slices) are perfectly feasible.[11] In long-running "batch" jobs, GC delays are not of concern in any case.

Lisp is now smaller than some net browsers or editors, and fits in memory that costs a few dollars at your corner computer store. Some Lisp systems can produce run-time executable code packages trimmed to exclude most development

[10]Available at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[11]See Appendix 1.

features, most particularly the compiler and debugging tools; further trimming can be done if it is possible to detect at "dump" time that eval and its friends cannot be used, and that the only functions used are those invoked explicitly or implicitly by user code.

It is not always possible to eliminate every bit of code not needed in an application, and so these run-time systems are rarely as small as the "minimal C code" needed to perform a simple task. (One could eliminate the garbage collector if one knew that only a small amount of store was ever needed. Deducing this automatically would be rather difficult.) As one mark, the minimal run-time-only binary from a commercial Lisp vendor, Franz Inc., is about 750 kB. For typical commercially supported Lisp systems one may need to pay a license fee to redistribute run-time-only binaries. This is sometimes cited as a factor in academic software projects' decisions to avoid Lisp, although the rationale does not bear close scrutiny.[12]

A license fee for redistribution of binaries is apparently not an issue in serious commercial Lisp-based software development where manpower and other costs dwarf the cost of buying such rights.[13] In fact if Lisp is properly considered not as a language, but as an "enabling technology," similar to say, a real-time OS (Wind River), or CORBA (Visibroker, etc.), or an object-oriented database (Poet or ODI), then fees or royalties are treated as an accepted norm related to the value added by the system. The reality is that availability and support on mission-critical issues (including updates as hardware and operating systems change) may simply be worth the price in the real world: the alternatives are limited or just as costly (i.e., building and maintaining a "free" implementation or purchasing from another vendor). While we may be used to a C compiler being free, it may actually be simply one that someone else nearby purchased. We address this further in Appendix 2.

One might be concerned about error conditions—"What if the garbage collection procedure cannot find more memory?"—except that one must face (and in a bullet-proof program, solve) similar challenges about "What if malloc returns 0?" or for that matter "What if the run-time stack overflows?"

Recovery from such situations inevitably is going to depend on features of the environment external to the language definition. Lisp as a system provides error-handling standards, and particular implementations may provide additional debugging or recovery tools. A system that has a simple description has just one advantage—namely simplicity—compared to a more sympathetic but more

[12]For fans of free software there is a GNU common lisp (GCL) as well as a CMU Common Lisp. Furthermore, the Lisp tradition is such that major vendors have "lite" Lisp packages free for the downloading.

[13]I am grateful for information on this topic from Franz Inc., J. Foderaro and Samantha Cichon, March 15, 1999.

complex system. This simplicity advantage rapidly disappears when the error handling must be written from scratch: simply crashing with "bus error" is not usually an adequate emergency action.

While Lisp can be implemented interpretively, directly, or via a byte-code system, as can Java or C, today's Common Lisps are usually oriented to producing compiled machine code from user programs. Lisp speed in critical programs can be further optimized by advisory declarations. There is some evidence that execution time is comparable to compiled C [5]. Additionally, early compilation also provides extra checking on syntax, argument counts, semantic program analysis, etc.

Functional programming is a perplexity in efficiency. In particular, the functional paradigm is favored by many Lisp programmers. While this leads to a kind of modularity that is helpful in debugging (in particular, tracing functions completely reveals the sequence of operations and operands), it can be wasteful. While programmers in C or other languages *can* use the same functional style, such a choice is somewhat less typical.

Let us explain the situation. Assume that you have one instance of a complicated data structure denoted `A`. You write a loop that repeatedly updates `A` to be a new combination of the old `A` and the value of a variable `i`: say `(dotimes (i n) (setf A (combine A i)))`. The ordinary interpretation of this would be to have Lisp construct a new object `C` where the value of `C` is `(combine A i)`. Then `A` is set to "point to" the same structure as `C`. The old value of `A` then becomes garbage and is eventually reclaimed from memory. This happens n times, and so n versions of `C` are produced with $n - 1$ of them being discarded. By contrast, a state-oriented (not functional) style of programming would be to alter or update "in place" all the components of `A`, typically by "passing in `A` by reference." In this model there is never a "new" or an "old" `A`: just the single `A`. This appears to be economical in storage, and indeed unless the functional `loop` above is cleverly optimized or somehow finessed algorithmically, the functional applicative style of programming loses in terms of efficiency.

There are three possible remedies in Lisp. The first is rarely useful: to declare that `A` is a `dynamic-extent` variable, and hope that the system will be clever enough to stack-allocate `A`. This is pretty hard to set up unless `A` is initialized to a constant: otherwise, it is not obvious that its initial value is unshared. The `dynamic-extent` declaration support seems to be most likely used for the processing of `&rest` arguments. More likely is that the compiler would not be able to make an effective optimization of such a declaration because the result of `combine` would be difficult to compute on the stack (unless it were perhaps a constant list).

The second remedy, appropriate for management of a set of large objects, is to implement a kind of subset storage allocation method. For example, if one were inclined to explicitly manage a collection of input–output buffers, one can set up a `resource` initialized to some number of fixed-length byte arrays, and use them one or more at a time via explicit allocation and deallocation. The payoff comes when a deallocated buffer is reallocated without being garbage collected. The mechanism can be implemented in standard Lisp in 18 lines of code in an example given by Norvig [6], and in another 10 lines, a `with-resources` macro is defined, regulating return of resources on exit from a dynamic scope.

The final remedy is the most well-known historically among Lisp programmers, requiring attention to the concrete data-structure level. It lends itself to abuse and can contribute to debugging mysteries: using in-place alteration or so-called destructive operations.[14] Historically this was done by functions `rplaca` and `rplacd` but in Common Lisp these are more easily specified via the `setf` mechanism. Consider changing the second element of the list `x = (R S T)` to `V`. Here's how:

```
(setf x '(R S T)) ==> (R S T) ;; initialize
(setf (second x) 'V) ==> V ;;
x ==> (R V T)
```

A functional program would create and return a NEW list `(R V T)` and leave the value of `x` alone. Any one of the lines below would do the job, returning as the value of `y`, the new list. The briefest is cryptic but no faster.

```
(setf y (cons (first x)(cons 'v (rest (rest x)))))
(setf y (cons (car x)(cons 'v (cddr x))))
(setf y '(,(car x) v ,@(cddr x)))
```

Why use the functional version then? Changing the arguments to a function by a "side effect" is considered bad taste. It makes debugging more difficult: you can't fix a bug in function `f` and try out `(f x)` if `x` is broken by a bug in `f`. Thus, side effects are used by most Lisp programmers cautiously. Since C programmers may not be able to retry `f` so easily, this is really an indictment of the C (or any batch) programming environment. The C process includes "remaking" the world by recompiling `f` and perhaps other programs, reloading and reexecuting the whole test framework up to the point of the error. The Lisp programmer would edit `f` or make some other change, and type `(f x)`.

What about data types? Isn't it wasteful to store data in Lisp's linked lists?

This depends on the alternatives, and how tight one is for space. Modern Lisp is not only about lists, but has arrays of small-numbers, single- or double-floats, bit-strings, 2-d bitmaps, character-strings, file handles, and a vast collection of

[14]This may sound dangerous, and it is. That is one reason that C is so error prone, because that is how virtually all C language programs with pointers are composed (that is, dangerously).

"objects" (including methods), etc. While C has some primitive raw objects, it is certainly possible that Lisp has the right mix of features at the right cost, and using its built-in data types can unleash a vast armamentum of program tools. Many Common Lisp implementations allow the definition, allocation, and manipulation of C structures directly, but this is used almost exclusively for communication with C libraries requiring such stuff, and rarely, if ever, for its own sake. With a sufficiently low-level approach one *can* build specialized data-structures that are more space-efficient than any higher-level language's normal structures, whether this is C or Lisp. We generally don't make much of such issues in comparisons: implementations of C typically waste some number of bits in each 32-bit pointer for machines that have an actual address space less than 4 GB.[15] The implementations also use 8-bit bytes for characters, when 7 or fewer bits[16] might be adequate. In almost all cases, the argument for space efficiency, even though proffered as a reason for using C, is rarely taken entirely seriously. If it were believed that a 10% improvement in speed or size were critical in competitive markets (say, in embedded systems where the vendor has control of all parameters: choice of CPU, etc.), then a strong argument exists in favor of assembly language, not C. In fact, critical components in Lisp implementations may be provided in assembly language, and the prospect exists for a programmer to write in assembly language within Lisp: after all, a typical commercial Lisp system has a compiler and assembler available even at run-time. The argument for assembly language programs where speed and size are truly critical still exists. We suspect that some C programmers, even though they will claim that C is "fast," fail to use the compiler's optimizer, and are therefore substantially slower than they could be! In such circumstances, *any* argument for speed is questionable.

Norvig [6] attacks the common myth that Lisp is a "special purpose" language for artificial intelligence, whereas languages like Pascal [7] and C are "general purpose":

> Actually, just the reverse is true. Pascal and C are special-purpose languages .... The majority of their syntax is devoted to arithmetic and Boolean expressions, and while they provide some facilities for forming data structures, they have poor mechanisms for procedural abstraction or control abstraction. In addition, they are designed for the state-oriented style of programming: computing a result by changing the value of variables through assignment statements. [6, p. ix]

[15]Even today, almost no programming systems have 2^{32} bytes of RAM installed. Why do we not use 24-bit pointers, or even 16-bit "word-aligned" pointers?

[16]If you can make do with upper-case letters and numbers you have 64 different values in a mere 6 bits.

Another point sometimes raised in justifying the use of C is its obvious compatibility with external libraries and programming interfaces supplied with an operating system. Since virtually all Lisps allow for the calling of "foreign" functions that may be in libraries (or in extremis, written in assembler or C), this is not a serious barrier. Some Lisp systems come packaged with rather complete API setups, which are in effect the provision of the appropriately declared linkages from Lisp to the library. Programs requiring call-backs can also be handled. A more significant issue may be the fact that the compilers directly supported by hardware manufacturers may evolve along with advances in the hardware, and these are likely to be compilers for C or (for scientific computing) Fortran. Thus, MMX extensions in C are provided from Intel. Since those portions of the Lisp run-time system and library that need access to the hardware tend to be written in C, some of these improvements are incorporated in Lisp. We concede that user programs intended to directly access new hardware features as soon as they are released may need to be written in assembler or a language that has been extended in an appropriate way. That language today is likely to be C and/or Fortran.

A final issue is familiarity with languages. This has had entirely too much influence in language selection. All else being equal, it is sensible to use a programming language when there is a large market of relatively skilled programmers familiar with it.

Are there Lisp programmers out there? All computer science graduates at UC—Berkeley (as well as many nonmajors), about 900 per year, are introduced to the Lisp dialect of Scheme. Many also learn C++ or Java. The most productive programmers may very well be those who find Lisp most attractive. We see companies that hire primarily on the basis of "experience in C programming" and quiz prospective hires on C-language obscurities. Such a strategy may fail to identify candidates with the key traits that eliminate the other causes of flaws: one would hope that companies wish to hire the candidates of high intelligence, and capable of creative problem solving. Indeed, the strategy of quizzing on C obscurities may *repel* the very best and the brightest.

As a variation on this theme of "we are writing in C because that is what more people know" we have heard anecdotally that it is difficult to assemble a high-quality team that can handle a mix of languages: given that if Lisp is introduced late into a project, or must interface to an existing library, then some percentage of the preexisting code (in C) must be "sucked in," requiring understanding of two languages. It is scary to think that some software producers view the key to productivity as targeting their development system as well as their hiring practices for lower-quality programmers. While in some areas it may be advantageous to be able to hire in quantity, it has seemed fairly evident that overall programmer productivity favors quality.

6. But Why is C Used by Lisp Implementors?

Some poking around shows that most, if not all, recent Lisp systems are implemented *partly in C*! Why? Because virtually all general-purpose hardware/ operating system combinations offer C compilers and a way to interface to their operating system through C. Since one must "bootstrap" from something, C is more convenient and more easily portable than assembly code. Assembly language coding is, however, sometimes required to incorporate low-level machine descriptions when no other satisfactory method can be found, and usually a good compiler will need to know about the assembly-level operation codes of the machine it is compiling for. Above that minimal level, (95+ %) of Lisp is implemented in Lisp (or a Lisp subset) language. For example, we know of no instance in which a Lisp compiler is written in a language other than Lisp.

In fact we feel reasonably comfortable with the view that the C programming language, subject to the constraints of today's world, is a good vehicle for implementing that small kernel of a (presumably different and better!) programming language. The question we have addressed here can be reemphasized: once you or someone else has implemented that better language, why should you continue to write in C?

7. Conclusion

It is unfortunate that so much commercial programming has fallen into the trap of using an essentially low-productivity language, and addressing shortcomings by a combination of advice, exhortations, and maxims. While tools like version control and interactive development frameworks help to some extent, they do not correct language flaws.

Would you consider undergoing surgery knowing that the tools in the operation included = and ==, and that the use of the wrong one would result in your death?

Significant complex applications have been programmed in Lisp, including Web-based commerce (stores and business-to-business), computer-aided design, document analysis, control and simulation systems, visual interfaces, and the traditional application areas including artificial intelligence, expert-system building, and programming language experimentation.

While we are not aware of controlled experiments that demonstrate the cost-effectiveness of Lisp vs Java vs C, we are forced to rely primarily on anecdotal evidence, personal experience, and most heavily, common sense.

We expect that programming in Lisp will continue to be especially appropriate for time-critical delivery of reliable complex software. We also expect that when there is a full accounting of all costs for a project, it will be seen as cost-effective as well.

Appendix 1: Cost of Garbage Collection

For purposes of argument, let us make the hypothesis that a programmer could otherwise keep storage straight and do foolproof allocation and return of storage, without any programming overhead recordkeeping (such as reference counts). It is certainly possible to do this with small programs where we can get away with deferring all deallocations until the end of the run, and let the operating system free the storage, at "no cost." You do this right, you win.

Winning is highly unlikely in the case of large, continuously running systems. In fact, such systems tend to be written with their own allocation programs (perhaps to keep a stock of particular sizes on hand and avoid running out when `malloc` fails), may use more storage, have more bugs, and be slower than a carefully crafted system. There is some evidence that rolling your own code will not be better than good implementations of "conservative garbage collectors" that heuristically guess at what might be collected: an attempt to partially mitigate the probability of storage leaks in C or C++. There are even Java GCs based on this idea.

A comparison of these to the run-time cost of doing garbage collection properly requires a detailed analysis on particular benchmarks, quite beyond the scope of this paper. However, we will try to give some plausibility arguments to support our contention that the cost in all but highly unlikely scenarios will be quite small. We could even make an argument that GC will, for many realistic scenarios, be faster than direct use of `malloc`.

We will, by hypothesis, assert that the GC algorithm is correct. The more sophisticated algorithms are not trivial, but these programs are reasonably mature, and have been beaten on mercilessly by many users for many years. Let us discuss briefly the efficiency issues.

There are two places to notice the cost.

The historically obvious lumped cost of doing the garbage collection has already been mentioned, and is highly satisfactory.

The generation-scavenging ideas that make possible a rather unobtrusive execution require that the system perform some recordkeeping so that the information needed for garbage collection is maintained in a consistent state. The technical requirement in modern generation-scavenging garbage-collection Lisp systems is that the programs must keep track of `setf` or other destructive changes in pointers in *old space*. In the case that a pointer from an old generation to new space is created, the system must make note of this garbage collection "root" that would otherwise not be known except by expensive scanning of old generations. No marking need be done for creating or modifying a pointer from new space.

An important optimization is that no marking and therefore no checking is needed for the large percentage of variables that are stack allocated, local within a function, and are naturally going to be used for marking, if they are still on the stack when a GC is prompted.

The added cost for a `setf` (from new space) is usually four instructions, most likely overlapped: A call,[17] a load of the new-space border, a compare, and a conditional jump back. The less likely route is about 35 instructions (on a Pentium), when a pointer from old space must be renewed.

Appendix 2: Isn't C free?

It's not always the case that the free G++ (GNU C) compiler is the one you should use, but even so, an alternative C compiler is likely to have already been paid for. We have already mentioned the availability of open source or GNU-licensed versions of Common Lisp system (see the Association of Lisp Users home page for descriptions: `www.elwood.com/alu/table/systems.htm`).

Does it make sense nevertheless to buy Lisp (and even buy new versions year after year)?

We quote from a Lisp user (3/17/99) on the `comp.lang.lisp` newsgroup, L. Hunter, Ph.D. of the National Library of Medicine (Bethesda, MD, USA):

> I'd like to point out that it is equally important (or perhaps even more so) that *someone* be paid, and paid well, to make "industrial strength" versions of the language. Top notch programming language people are expensive, and I want as many as we can collectively afford to be working on LISP. Moving the language into the future, and even just keeping up with the onslaught of new platforms, standards, functions, etc., that we hardcore users need is not something that is likely to happen for free. Lisp is NOT Linux – there isn't nearly the motivation nor the broad need driving Lisp development.

ACKNOWLEDGMENTS AND DISCLAIMERS

Thanks for comments from John Foderaro and Duane Rettig of Franz Inc., as well as George Necula of UC—Berkeley. Remaining errors of omission and commission are the author's own. The author also admits to not only liking Lisp, but to being one of the founders of Franz Inc., a vendor of Lisp systems and applications (www.franz.com). Although he has a potential to profit personally from the more widespread adoption of Common Lisp, he obviously thinks others have a potential to profit from using Lisp as well!

[17]Why not an inline expansion? It appears that adding to the bulk of the the code weighs more heavily against performance than the call. I am grateful to Duane Rettig of Franz Inc. for information on this matter.

REFERENCES

[1] Yu, W. D. (1998). "A software fault prevention approach in coding and root cause analysis." *Bell Labs Technical Journal*, **3**, 2, 3–21. Available at `http://www.lucent.com/minds/techjournal/apr-jun1998/pdf/paper01.pdf`. See also Yu, W. D., Barshefsky, A., and Huang, S. T. (1997). "An empirical study of software faults preventable at a personal level in a very large software development environment." *Bell Labs Technical Journal*, **2**, 3, 221–32. Available at `http://www.lucent.com/minds/techjournal/summer_97/pdf/paper15.pdf`.

[2] Maguire, S. (1993). *Writing Solid Code*. Microsoft Press, Seattle, WA.

[3] Kernighan, B. W., and Plauger, P. J. (1974). *The Elements of Programming Style*. McGraw–Hill, New York.

[4] Fry, C. (1997). "Programming on an already full brain," *Communications of the ACM*, **40**, 4, 55–64.

[5] Fateman, R., Broughan, K. A., Willcock, D. K., and Rettig, D. (1995). "Fast floating-point processing in common Lisp." *ACM Transactions on Mathematics Software*, **21**, 1, 26–62.

[6] Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, CA.

[7] Kernighan, B. W. (1981). "Why Pascal is not my favorite programming language." ATT Bell Labs, Murray Hill, NJ. Available at `http://www.lysator.liu.se/c/bwk-on-pascal.html`.