Bungee Jumping off the Ivory Tower: Payoffs and perils when fledgling computer professionals are imprinted on Lisp

John Hodgkinson
Gensym Corporation
125 Cambridge Park Dr
Cambridge MA 02140
(617) 547-2500
jh@gensym.com

Abstract

From the viewpoint of at least one corporate consumer of computer education, Lisp-based training confers many professional advantages, even if the student never goes on to use Lisp in real life. Among these advantages are a grasp of computer science fundamentals that comes from simply knowing the Lisp language (Whorf was right, in this domain at least) and good software engineering habits (incremental development, abstraction, modularity, object-orientation). These habits are widely believed to be advantages, but Lisp is not widely seen as helping students acquire them. This paper will corroborate the connection between Lisp-based training and these good habits. On the other hand, some of these habits may not be good at all (this is one reading of Gabriel's "worse is better" theory). This paper will argue that Lisp-based training is more flexible than training based on popular imperative languages at discouraging the bad in these habits. Then this paper will describe some true drawbacks of Lisp-based training. These drawbacks include stylistic jingoism, excessive delegation of performance issues to the Lisp implementor or to the machine, superstitious avoidance of the more complex (albeit more casually implemented) Lisp facilities, stubborn reluctance to distinguish between compilation, runtime, and delivery environments, and a tendency to make use of Lisp's extensibility to re-invent the wheel merely because the tread size is not quite right. Ironically, Lisp itself contains the cure for the diseases it spreads. This paper concludes by offering, from real-world experience, some suggestions for modification, not abandonment, of Lisp-based training.

Why teach computer fundamentals using an arcane, ancient language that has found limited use in the world beyond the ivory tower? Why not use a language that is found everywhere, or build a special pedagogical language from scratch? Arguments about the best way to conduct computer education can go on without end, but unfortunately, Darwin always has the last word. If the Lisp language is too fascinating for its own good, then using it to train computer professionals is ill-conceived, as ill-conceived as the topheavy tail feathers that make a bird of paradise tip over as soon as it displays them. If, on the other hand, the marketplace rewards elegance, universality, and flexibility, then Lisp can substantially augment a student's fitness to survive when jumping off the ivory tower.

Gensym Corporation is a growing, profitable software company that produces G2, a real-time expert system shell. G2 is installed at over 2000 sites around the world, often in applications where performance and reliability are crucial. G2 runs on a wide range of workstation and PC platforms (including Sun, HP, IBM RS6000, Windows, and Alpha), with more ports in progress. To maintain and extend G2 and its satellite products, Gensym requires the best developers and engineers. To that end, we invariably look for computer professionals with a background in Lisp.

We do not look for a Lisp background simply because some of G2's source code is written in this language. Since we translate our Lisp sources to C for consistent behavior across platforms, many of our developers will not end up working directly with the Lisp sources. As well, the Lisp we use at runtime is restricted to a small subset of Common Lisp functionality. We have compelling performance reasons not to use generic arithmetic or closures at runtime, for example. In order to provide real-time response, we have rewritten many Lisp facilities that are known to create garbage in memory, including `cons` itself. For portability's sake, we have rewritten other Lisp facilities whose behavior is not yet standardized, such as `loop` (we need the ability to define new iteration paths) and a subset of the Common Lisp Object System. Finally, we confine our Lisp calls to those that translate to portable, reliable C code (we have found the notion that C is intrinsically portable or reliable to be a pernicious myth). Even with a Lisp background, therefore, new developers will follow a steep learning curve.

Nevertheless, a Lisp background confers a broader advantage, something that might be called a **superior world view**. Granted, world views are intangible things, but "intangible" does not imply "inconsequential." A world view can be seen as a set of habits that supply strategies for solving problems. Both good habits and effective strategies are necessary, as we will see.

It is hard to mention world view and language in the same breath without recalling the Whorfian hypothesis. Roughly speaking, this is the notion that some thoughts are easier to think in some languages than in others. This hypothesis has been largely discredited for natural languages. One disconfirming experiment, for example, shows that there appear to be language-independent ways of making color judgments. But computer languages are another story. Programs are easier to implement in some languages than in others. The most trivial confirmation is the fact that code disassemblers make life easier.

Computer education can take advantage of this **modified Whorfian hypothesis**. A major part of educating computer novices is teaching them that "everything you know is wrong." Intuitive algorithms (like bubble sort) are seldom the fastest. Time and space are not independent entities, but co-exist with unavoidable tradeoffs (you can't make your program as fast as possible and then go on to make it as small as possible). Some problems are inherently serial, and can't benefit from any parallel strategy. True randomness is hard to find. A symbol is not merely a name, but a unique name, guaranteed in various ways not to clash with other names that may look identical. And so on.

How does Lisp prevail here? Lisp is a language that is fundamentally indifferent to machine model. This indifference goes deep: Lisp is oblivious to whether the machine is a specific piece of hardware, an architecture (like RISC), a paradigm (like the von Neumann model), or a mathematical abstraction (like a Turing machine). The semantics of a given Lisp construct span

from low level to high level (a `setq` is at first glance like a move instruction, but `setqs` take place within a special binding environment, something outside most instruction sets). A blunt way of putting things is that, having learned a language in which everything you know is already wrong, there is no additional confusion in learning that your common sense about algorithms is also wrong. A more accurate way of saying this is that Lisp permits students to reason equally well at **any distance from the machine**. Destructive operations like `(setf car)` and `nconc` may be used when dealing with pointer manipulation, whereas copy-and-drop operations like `append` may be used when memory is not the focus. Awareness of the various distances from the machine can also help when evaluating optimizations. For instance, an optimization like cdr-coding is useless in G2, which must manipulate cons structures destructively. In contrast to Lisp, C and C++ have specific memory and evaluation models. It is happy coincidence that these models cover many popular pieces of hardware, but how long will this remain so? To paraphrase Tsiolkovsky, the von Neumann bottleneck may be the cradle of computation, but you can't live in the cradle forever.

One world-view strategy provided by Lisp training is abstraction. The benefits of functional abstraction are well known, among them lexical scope, code reuse, and the ability to parametrize program behavior. And, as the size of a program increases, so does the need for textual abstraction. Through its macro facility, Lisp permits **true textual abstraction**, unlike the mere token-manipulation provided by the C preprocessor. True textual abstraction lets us perceive computationally interesting events — evaluation, binding, value return, and so on — in a different arrangement from the one mandated by core language syntax. We can focus on a different ordering, or on a salient subset of events. Inability to focus in this way makes program maintenance and extension more difficult, and in some cases even impossible (try adding a new modify macro to += and friends in C or C++).

Yet the risks of abstraction are becoming known, too, as [Gabriel, 1992] has pointed out. A poorly designed but pervasive abstraction can leverage into big trouble if it has to be modified, and if the abstraction was not designed by an expert, eventual modification is all too likely. An abstraction that fails to co-design data and control flow is fragile and limited, although such abstractions are in the overwhelming majority. Slot-based abstractions encourage programmers to conflate the instantaneous state of an object with its history. The more opaque an abstraction's interface, the more its clients are reduced to slow methods (like empirical tests) to discover how to use it in boundary cases. Yet the more transparent the interface, the more its clients are tempted to dispense with the abstraction altogether. This dialectic often turns programs into a mixture of abstraction sites and unabstracted code, and can even sometimes make them harder to understand than before abstraction set in.

But the superior world view furnished by Lisp is not simply an unstructured set of glamorous problem-solving strategies like abstraction. It is the habit of judicious choice among them. A blindly applied abstraction can do as much damage as a missing one. Some of the wrong reasons

for using an abstraction are mistrust of its clients, misplaced desire for elegance, and fastidiousness about textually repeated code. These reasons are language-neutral, but their remedy depends on having a flexible language like Lisp. The best way to learn how to abstract is to write a bad abstraction and suffer the slings and arrows of outraged clients. Lisp gives this vital process a rapid turnaround time. Code may be inlined or placed out of line, sites of the abstraction's use may be examined, type declarations heeded more or less strictly, and (Schemers, please look the other way for a moment) variables may be selectively captured or shadowed. Even experts can benefit from this **ease of turnaround**, since it permits abstractions to be improved incrementally or radically. Installing a good habit is painful, exercising a good habit should be guiltless pleasure — Pavlov wields a two-edged sword.

"Worse is better" may be irrevocably loose in the world, but the world contains two types of people: those who like it and those who don't. Lisp users seem to have the unanalyzed notion that "the right thing" will somehow triumph by virtue alone. Alas, "the right thing" often needs help. Most developers find themselves responsible for a **living program**, one they did not completely write themselves, and one on whose behavior many users depend. There is always a tension between tearing out large, ugly parts of such a program and maintaining its behavior. One compromise is the gradual re-writing of parts of the program to conform to current practice. We have found that Lisp facilitates this task. The same flexibility that eases abstraction permits us to erect a scaffold around offending sections of the program, and make them conditional on a wide variety of contexts. Some techniques we use to achieve this include runtime flags, optional function arguments, reader features, and package discipline. These last two techniques are especially germane to this paper.

Since we maintain a single set of sources for all Lisp environments, we admit that reader features are often a necessary evil. However, our use of a Lisp-to-C translator, itself a Lisp program, requires that macroexpanders work across different Lisp implementations as much as is practical. In order to intern symbols in their proper package at macroexpansion time rather than read time, we have used Lisp's macro facility to circumvent the Lisp reader, interning symbols only when their package is known to exist, producing no-op code in other Lisp implementations.

We have used package discipline during our transition from CLtL Common Lisp to ANSI standard Common Lisp. It is worth going into some detail here to illustrate the concept of **necessary hair**, a middle ground between abstraction and the refusal to abstract. This concept alludes to the fact that some tasks are inherently special-case, performance-intensive, or refractory of abstraction. Implementing some data structures (for example, buffers) is neither elegant nor abstract. Worse, it seems that such implementations must be written over and over, even in object-oriented systems. Abstraction may come down the road, but the necessary first step, for a commercial firm at least, is to get the program to perform well enough to justify its existence.

Our source code is read into a single package, call it `g2`. In order to access core Lisp symbols, the `g2` package started out by directly using the `lisp` package (see Figure 1). But we wished to upgrade to ANSI Common Lisp, which specifies that core Lisp symbols should reside in the `cl` package instead of in the `lisp` package. A number of other considerations hold for our use of core Lisp symbols. First, we only use a subset of Common Lisp symbols, and we would like to guard against inadvertent use of any Lisp symbol outside our canon. (For example, we do not use the Lisp pathname or file I/O facilities because they create garbage.) Further, we must shadow some of the Common Lisp symbols that we do use, because of mistakes in an implementation, or because a straightforward C translation is not always reliable. (We have found, for instance, that the C boolean-or construct `||` is not implemented correctly on all platforms.) And we occasionally need to bypass the shadowing on a symbol; that is, we need to refer directly to a core Lisp symbol (say for performance reasons). Because of these considerations, our sources sometimes contain explicitly package-qualified symbols in the core Lisp package, whatever its name may be. One way for us to upgrade to ANSI, therefore, would be a simple textual replacement of every explicit `lisp` package-qualifier with a `cl` qualifier.

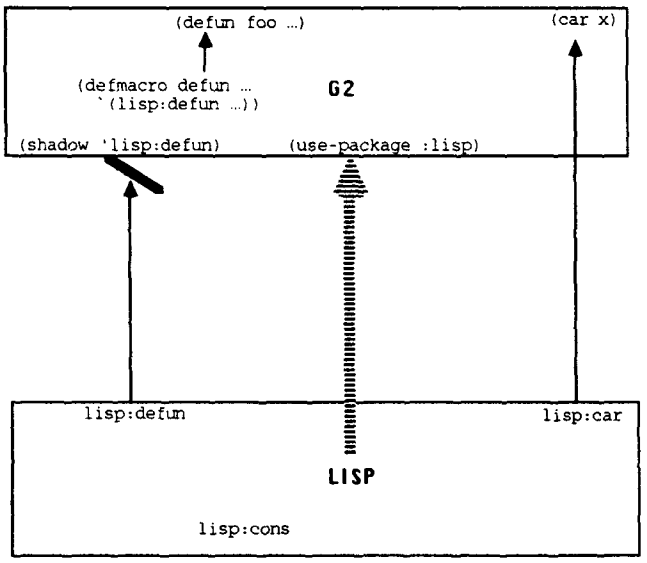

Figure 1. The Naïve Way.

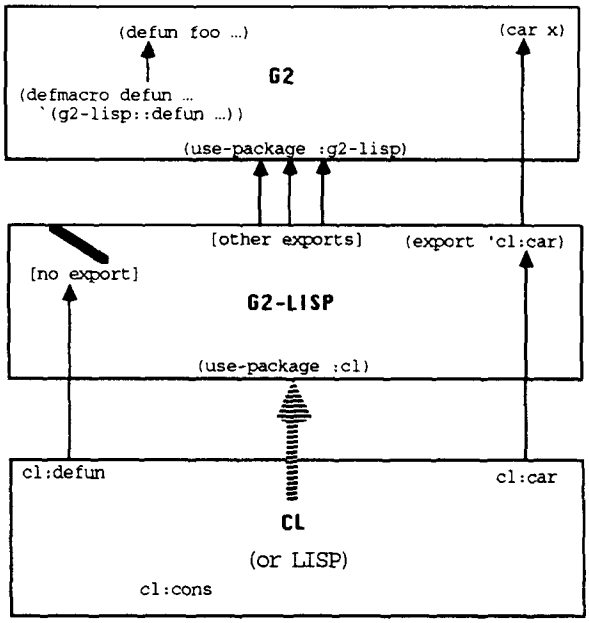

Figure 2. Necessary Hair.

But one of our hard-won heuristics states that a change that happens once is likely to happen again. So, instead of "the wrong thing" — editing the sources — we now indirect through a package of our own, call it `g2-lisp` (see Figure 2). Depending on implementation, the `g2-lisp` package uses either the `lisp` or the `cl` package. The `g2-lisp` package exports only those Lisp symbols that are in our canon, minus those symbols we wish to redefine. (This replaces our use of the Common Lisp `shadow` function — to "shadow" is now to "decline to exp(".) Our source-code package `g2` then uses the `g2-lisp` package to access all the Lisp symbols in our canon that we don't have to shadow, and redefines the Lisp symbols that we do have to shadow. For base cases in these redefinitions, we use an explicit `g2-lisp` package qualifier to refer to the core Lisp symbol. So when the source code contains an unqualified symbol, it will either indirectly access the core Lisp symbol or our redefinition, depending on our needs.

So where is "the right thing" hiding in all this hair? The Lisp reader now protects from us from accidentally using a core Lisp symbol that is not in our canon. The heretical symbol is simply read as a new symbol, with no connection to the core Lisp package. When the compiler subsequently issues a warning about an undefined function or undeclared variable, we are led directly to the scene of the crime. And we are immune from any future changes in the name of the core Lisp package, whether from an over-solicitous vendor, or because of a coup d'état in a standards committee. We therefore believe the complexity of our fix is necessary, since any simpler fix (such as manipulation of package nicknames or large-scale source edits) fails to do everything we need.

Not only is the Lisp world view superior in conferring good software engineering habits, but it is a superset of the world views promulgated by other, more popular but less powerful, languages. Such **world view supersetting** makes it easier to learn other languages, even inferior ones. Lisp is rich enough to contain the seeds of many other languages, without forcing a complete implementation of those languages. Lisp is a paragon of incremental, interactive development. Core Lisp syntax is uniform and simple. (Experience with peripheral, more complicated Lisp syntax facilities, such as reader macros, backquote, and format, could help more advanced students understand guild secrets like the multiple evaluations in UNIX and VMS command-lines.) Lisp lacks as well the confusing variety of associativity rules offered by other languages. All these factors allow the programmer to concentrate on the task at hand rather than the token placement necessary to achieve the task. Lisp, translated to C, can be as universal as C, but without its drawbacks.

Fun versus profit. We have to bring up the Puritan ethic here, since there is something about Lisp users that makes them disbelieve it. To them, fun programming is useful programming, an idea perhaps encouraged by the power of the Lisp language. Wielding Thor's hammer is generally more pleasurable than waving around its Fisher-Price variant. But wielding to what purpose? [Baker 1992] presents a series of ingenious definitions of Lisp special forms in Lisp itself. Granted, a metacircular definition of a special form is by no means a mandated implementation.

But, despite the healthy contempt for lawyers evinced in the paper, this kind of definition brings casuistic disputation to new heights. Now, thanks to multiple ways of expressing the same special form, we get to argue about, not only the meaning of boundary cases, but the meaning of core cases. In other words, now we get to argue about the shape of the pin and the definition of dancing, as well as the number of angels. This of course should be tempered with a lot of appreciation for the neatness of it all. Try doing metacircular definitions in C without writing an entire compiler.

[Allard & Hawkinson, 1991] note that performance must be designed into the program "from the ground up." It is not something to veneer on later, with desultory declarations and embarrassed inlinings. It is work, not fun, to locate performance-critical parts of the program you are designing, and add indirections to overcome some of the known peccadilloes of Lisp. We have extended our Lisps with many mechanisms that allow considerations of efficiency to take part in the early stages of program development. We have suites that let us bypass generic Lisp arithmetic, and unboxed number declarations that let us take advantage of raw C floats and integers where possible. Since we know the layout of many tree structures beforehand, we have instituted list accessors whose behavior is undefined when given `nil` as an argument. We have introduced an inlined funcall mechanism for use when indirectly calling a function that might as well be a C function pointer. The necessity for these facilities may be difficult to convey to students, who already have too much to think about. But until our new developers grok the way things must be, we give them the ability to choose functional, error-checking (slower) operations in development environments, that expand properly to inlined, error-oblivious (faster) operations in distribution environments.

This brings us to another inexorable impediment to fun: the difference between interpreted and compiled code (or, more generally, the difference between interpretation-based languages and compilation-based languages). Although the Common Lisp Committee has done yeoman's work in fleshing out Lisp compiler semantics (in the face of outcries from both purists and vendors), this is an ironclad barrier. Now that we translate to C, we have had to adjust to native C debuggers. Ironically, Lisp has given C development environments their best ideas. But it is simply not feasible to run such development environments on every platform where we deliver. So we have to resort to the debuggers bundled with the C compilers we use. The differences among debuggers in various Lisp implementations are matters of ideology, and thus easily ignored. But the C debuggers on various platforms have fundamental variances, most notably when we try to reenter the code at different places, traverse the call stack, or refer to complex data structures.

As an aside, all debuggers we've encountered have a serious defect. They assume a universe in which multiple processes are rare. For a real-time, distributed system this is, to put it mildly, not the case. This might be a way for Lisp to gain an edge. UNIX and VMS environments encourage multiply threaded programs, but as far as these environments are concerned, manipulating threads

profitably is a black art practiced by laconic, temperamental alchemists called system programmers. Lisp could break up this priesthood. Debugging multiple processes is inherently interpretive, since it is inherently asynchronous. And, while it may be too much to expect from a C debugger, the ability to macroexpand C constructs is increasingly important as we encounter more and more platform-specific C compiler bugs in our porting. We have implemented this ability in-house, but integrating it with existing C debuggers looks difficult.

In order to track down particularly evasive C compiler bugs, we have had to dabble further with C preprocessing. A rough way to characterize such bugs is that the C compiler on a given platform sometimes seems to make local, intermittent mistakes, mistakes that don't appear on other platforms. The best diagnosis we can make in such cases is that a certain pattern of C code is not handled properly. These **patterns of deceit** are sometimes particularly complex sequences of operations or particularly deeply nested constructs. Sometimes they are neither. In order to search megabytes of translated C code for such patterns, we have had to devise C tokenizers of varying degrees of completeness. We then arrange for the translator to produce a different sequence of C expressions on all platforms. After all, we must maintain the "portability" of our C code. (C provides a certain consistency across platforms, but this is a long way from true portability.)

One principle of teaching is (per Einstein) to make things "as simple as possible, but no simpler." C seems to misread this into "as complex as you can get away with, and never less complex than von Neumann." The notions of sequence points and implicit type conversion in C are an illustration. From students' point of view, the compiler is allowed to rewrite their code, not in the benign, semantics-preserving way of a Lisp compiler macro, but in a way that changes numeric results and reorders control flow. C library functions can be enormously counter-intuitive as well. The function `ungetc()` is allowed to "un-unget" characters if a buffer-clearing operation has occurred before the next character is re-read. Since C contains neither compiler nor dynamic linker, constructs like include files, preprocessors, and external compilers present an additional barrier to understanding. Since in some sense **Lisp is Lisp all the way down**, Lisp conceals no such confusing interactions with the operating system. As well, Lisp macros can handle most anomalous evaluation models. As a final coffin nail, the notion of graceful degradation is absent from C. Common Lisp compiler safety settings can insert runtime checks that prevent mistakes like type mismatch and array overflow, whereas the C philosophy is that novices or prototypers deserve what they get.

In closing, our experience with fledgling Lisp programmers prompts a series of suggestions for those who produce them.

Don't turn out any more Lisp zealots. Instead, turn out tolerant polyglots whose milk tongue is Lisp. Zealotry causes friction, not only friction within groups which should be fighting on the same side, but friction with the mainstream, which can then thankfully marginalize Lispers as

temperamental squabblers.

Don't be timid about teaching students in a language they will probably not end up using. Pedagogical uses and commercial uses of even the same language will diverge wildly. Cross-language expertise matters more than a slavishly exact simulation of the student's eventual tools and surroundings.

Memory exists, memory matters, and students need to know this. Just as camping in the wild can make students appreciate civilization, one tack might be to force students to be aware of memory management before they can thankfully relinquish it to the garbage collector. Even if they never again had to manage memory by hand, this would prepare them for the real world, at the very least for one small corner of it. They will also be in a position to evaluate new memory management strategies. Since it lacks pointer encapsulation, C++ offers something called "heuristic garbage collection." Who said C++ wasn't on the cutting edge? But this is a cutting edge some students may wish to avoid.

Emphasize what Lisp is abstracting away from — memory models, numeric representations, file systems, compilation targets. This has the effect of making students wise beyond their years, decreasing the population of students who at their first job interview can discourse knowingly on abstraction but who draw a blank when asked about performance. This emphasis also puts the sense of "the right thing" in its place, as an ideal to strive for and sometimes reach, rather than a non-negotiable demand to insist on in the face of collapsing deadlines or revenues.

Point out that no language can abstract away from every dull detail. Lisp is not a magic filter that transforms every dingy problem into an interesting testbed for elegance and abstraction. Some problems cannot be solved elegantly and abstractly. Worse, some problems whose first-pass solution is elegant will later reveal complexities that require dismantling the elegance. Even at its best, abstraction just moves complexity to a different part of problem space. Keep in mind, though, that when it is time to sweep something under the rug, Lisp's flexibility provides a better broom.

Inculcate a respect for existing standards, and a skepticism for new standards. At the very least, "new standard" is a contradiction in terms (think about it). Even if programmers have to pick and chose among fragments, use of existing standards will position them fairly close to any eventual standard. Then the nonstop brawl about emerging standards is relegated to its proper place, a game with its components of luck and skill. We love a good fight as well as the next guy, but we have to hedge our bets. We have had to pick and choose fragments of standards for streams, error handling, array representation, object orientation, and graphics, among others. Existing standards should have the respect due them, no more. Remember that handwaving is reprehensible wherever it occurs, even in a standard. A case in point is the fact that the X model believes in infinite

memory and asynchronous error reporting. These are contradictions in terms, too.

Demonstrate the ease with which Lisp implements new paradigms (parallel processing, object orientation, metacircularity, distributed computing, realtime constraints). Students could thereby acquire immunity both to messianic claims of new languages (like Dylan, the last OO language you'll ever need, once agreement comes on how to do iteration, macros, and arithmetic) and bombastic claims of old (C++, wearer of the emperor's new clothes, the emperor being C).

Lisp is not a new Latin. Latin was the language of a corrupt people who conquered the world and then lost it. Lisp is not a new Esperanto. Esperanto was an idealistic attempt at a universal language that failed by refusing to face reality. Some might say that Lisp's best hope is to become a new French, a language spoken by snobs with a side interest in diplomacy. My own hope is that Lisp becomes a new English, spoken everywhere (though not necessarily as a first language), and powerful enough to express everything from VCR instruction manuals to Shakespeare.

References

Allard, James R., and Lowell B. Hawkinson, "Real-Time Programming in Common Lisp," Communications of the ACM, September 1991, vol. 34, no. 9.

Apple Computer Eastern Research and Technology, Dylan: An Object-Oriented Dynamic Language, Apple Computer, Cambridge, Ma., 1992.

Backus, John, "Can Programming Be Liberated from the von Neumann Style?," ACM Turing Award Lectures 1966-1985, ACM Press, New York, N.Y., 1987.

Baker, Henry G., "Metacircular Semantics for Common Lisp Special Forms," Lisp Pointers, Oct.-Dec. 1992, vol. V, no. 4.

Crystal, David, The Cambridge Encyclopedia of Language, Cambridge University Press, Cambridge, U.K., 1987.

Gabriel, Richard P., "Worse is Better," Proceedings of the 1992 Lisp Users and Vendors Conference, San Diego, Ca., 1992.

Gensym Corporation, G2 3.0 Reference Manual, Cambridge, Massachusetts, 1992.

Harbison, Samuel P., and Guy L. Steele Jr., The C Language, 3rd edition, Prentice Hall, Englewood Cliffs, N.J., 1991.

Steele, Guy L., Jr., Common Lisp: The Language, 2nd Edition, Digital Press, Bedford, Ma., 1990.

Stroustrup, Bjarne, The C++ Programming Language, Addison-Wesley Publishing, Reading, Ma., 1987.

Wessells, Michael G., Cognitive Psychology, Harper and Row, New York, N.Y., 1982.

Whorf, Benjamin Lee, Language, Thought, and Reality, M.I.T. Press, Cambridge Ma., 1956.