Unpublished.

9. *Some "professional" Ada compilers are so big and slow that they should be laughed out of any conference on programmer productivity*. A very slow compiler—regardless of the quality of the code it generates—saps the life out of programmer productivity. I recently had an occasion to compile a 5,000-line program on several different Ada compilers. The "professional" compiler took 40 minutes to compile on a Sun-4, with all optimization turned off, and it had to be terminated after 1.5 hours with the default optimization turned on. An "educational" Ada compiler took less than 2 minutes on the same program on a 486 machine, and ran 3X faster, as well. (The large compiler, unoptimized, took 20 Mbytes of swap space, while the smaller one took 1.1 Mbytes on the same program.) I have also heard "horror" stories directly from people who have experienced 1 *week* (24 hours/day) compilation times on 200,000-line Ada systems. Some programmers use compilation times to make coffee or eat snacks. With some of these Ada compilers, one could actually take vacations!

In summary, the Ada language remains an excellent programming language in the hands of people who understand it and have the appropriate tools. For people with backgrounds in several programming languages and environments, however, they may also find a number of frustrations. Luckily, the frustrations are outweighed by the advantages.

Sincerely,

Henry G. Baker

Unpublished.

Thus, an obsolete requirement—the ability to recover from multiple syntax errors—forces Ada vendors to ignore one of the real successes of Computer Science research—the mechanically generated parser—and go back to labor-intensive and buggy hand-crafted parsers. (Computer scientists should at some point come up with a way to tally the life-cycle costs of Baroque syntax; my personal belief is that after factoring in issues such as the net present value of all federally-sponsored R&D on syntax analysis, error recovery, syntax-sensitive editors, and other syntax-specific programming tools, the L-word starts to look pretty good.)

6. *In retrospect, Ada's insistence on no subsets was a serious strategic error*. Even at the time Ada was being created, it was acknowledged to be a large language. Large languages are a problem in several ways—they are hard to implement, and they are hard to teach. A large amount of experience could have been gained on both counts with smaller "serial" subsets of Ada without the complexity of subtypes and derived types. In particular, many undergraduate courses still teach only "serial" Pascal or C, so a small serial Ada would work just fine. Secondly, some Ada compilers were attempted which themselves were written in Ada. For this purpose, it would be hard to argue that all of the complexities of tasking, record representations, etc., were required. More large programs of these kinds would have given earlier feedback into the Ada process itself, and thereby remove some of the more glaring nits from Ada83. (This early feedback could also have averted some of the bigger problems with *rendezvous*, but that is another story.)

(Some of our best languages have come from "special-purpose" languages that were invented solely to write a complex compiler. For example, the BCPL language ("basic CPL") was invented for the sole purpose of writing a compiler for the CPL language of world-class (i.e., PL/I) complexity. As could be expected, BCPL was wildly successful at the time, and CPL sank without a trace. In fact, BCPL can be considered the "uncle" of C, as C is a makeover of the BCPL language to take advantage of the clever instructions in the PDP-11 architecture.)

7. *The Ada process itself exhibits the major faults of the "waterfall" model*. The waterfall model of software development is a unidirectional model in which the process continuously converges the original ambiguity of the problem into the concreteness of a final program. The major problem of this process is that it is impossible to "back up" and reconsider certain decisions in the light of information that later becomes available—e.g., a certain data structure decision is just too slow to be usable, or a construct we planned to use cannot be compiled successfully by the target compiler. Although one of the major original goals of the Ada language was for programming "embedded systems", there are a large number of problems with the Ada model of tasking that surfaced only when people attempted to actually build such systems. Unfortunately, there was no mechanism in the Ada process to reconsider Ada's choice of tasking and synchronizing primitives in the light of this new information. There has been much progress in understanding these problems and their solutions, but fixing them is now will be much more difficult and expensive than it would have been in 1983.

8. *A language created by a political process requires an on-going legislature to maintain it*. Notwithstanding my earlier point, the lack of a sitting Ada committee with the power to make changes has allowed the language to fall into disrepair. In a living language, times and needs change, and changes in usage require changes in the language. Ada has a number of minor problems which have persisted for over ten years, but without a sitting legislature, there is no way to get the problems fixed. Thus, little problems fester until they put large portions of the language at risk. For example, the "mainframe" orientation of the Ada I/O facilities is seriously out of date in today's window-oriented workstation environments, but there was never any mechanism to make the appropriate changes. The Ada-9X committee will fix many of these problems, but is then expected to go out of business for another ten years. Perhaps "small" languages like C duck thousands of bullets every day by simply refusing to become embroiled in such transitory phenomena as I/O standards.

Unpublished.

3. *The problems with Ada stem directly from its political heritage—i.e., Ada is good in spite of, rather than because of, its creation process*. Ada is the direct result of a political process, exactly like the tax code of the Internal Revenue Service. Like the tax code, it has a certain overall rational consistency, but also large numbers of maddeningly inexplicable loopholes. There is a good scientific reason to believe that the old saw—"a camel is a horse created by a committee"—is essentially correct, because we find that much fractal behavior is the result of competitive processes. Therefore, we can expect that *any* product of such a competitive political process will have an equivalently fractal appearance. Since the boundaries of fractal curves are exceptionally complex, it is no accident that the boundaries of the Ada language are also very complex.

The complexity of the boundaries of the Ada language has led to a new type of computer scientist—the *language lawyer*. This is a person who missed his calling as a real lawyer or a rabbi, but who refuses to leave the computer science field. The language lawyer's job is to interpret the arcane wording of the Ada Bible ("LRM") and come out with pronouncements such as "if a discriminant constraint is applied to a private type or to an incomplete type before its complete elaboration, the second part of the check cannot be performed when the subtype indication is elaborated because no subcomponent declarations exist" (AI-00007/19—ever wonder why they provided so many leading digits in these numbers? Did they know something we didn't know?).

4. *Ada's poor experience with compiler validation raises serious doubts about many touted mechanisms for ensuring software quality, including various certification schemes*. The fact that so many early Ada compilers passed validation despite incorporating so many bugs as to make them totally unusable should be cited as excellent evidence that validation schemes are inherently flawed. This is not an indictment of the people who created the validation tests or the people who organized the testing; by all accounts, the ACVC and the AJPO performed a remarkably good job on both counts. The problem with the ACVC process is the fact that these tests were *synthetic* tests, rather than tests on a large number of working programs, which is the *ad hoc* validation system used for all C compilers. The reason that people are happy with C compilers, even though they don't all handle the subtleties of ANSI prototyping in the same way, is that people know that these compilers have successfully compiled the whole of the Unix system, all of its tools and services, and probably the compiler itself. Ada compilers, on the other hand, are guaranteed to handle the various quirks of comma-separated variable initialization, but this is no guarantee that they can handle the common constructs of a large-scale system. As one ACVC wag has it, the ACVC tests test compilers around the *perimeters* of the region of acceptability, but do no testing at all in the *center*, where most user programs lie.

One could make an economic case for the fact that the free market system already provides for the best (i.e., the most cost-effective) compiler validation system yet invented. This is because people will refuse to buy a product that won't compile and execute the programs that are important to them. The market system also allows the customers to make certain trade-offs—e.g., compiler speed vs. obscure features—that the current heavy-handed approach does not allow.

5. Ada's excessively picky ACVC tests have led, in some cases, to a requirement for expensive and buggy compilers. The paradigm in effect at the time of Ada's creation was that of batch processing. In a batch processing environment, one would be lucky to get one compilation per programmer per day. In such an environment, the ability to find every syntax error in a single compilation was highly prized, and therefore clever "repair schemes" are required to recover from various kinds of syntax errors. In fact, the ACVC apparently requires compilers to find and correctly describe a variety of syntax errors, even after another syntax error has already occurred. Although parsing technology is quite sophisticated—being one of Computer Science's most extensively researched topics—it apparently is not sophisticated enough to mechanically create parsers which can handle the Ada syntax, together with its ACVC requirements for syntax repair. Therefore, compiler vendors are required to hand-craft their parsers (or at least extensively hand-modify them) in order to meet the ACVC requirements. As a result, these vendors are in no mood to spend heavily on speeding these parsers up, because any changes are likely to cause invalidation.

Unpublished.

Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

(818) 501-4956
FAX: (818) 986-1360

November 9, 1992

ACM Forum
Communications of the ACM
1515 Broadway
New York, NY 10036

Dear Sirs:

As a sometime member of the Ada community, I enjoyed your special issue on the Ada programming language (Nov. 1992). However, I feel that the net result of the series of articles did not provide a balanced view of the Ada language and community—probably because most of the contributors were too closely involved to have a larger perspective. I would therefore like to provide a bit of additional perspective from someone who is enthusiastic about Ada, but who is also remainded on a daily basis of the problems facing Ada users.

1. *Ada is a much better language than it has any right to be, given its parents and its process of creation.* In particular, Ada is a much better language than most computer science academics give it credit for. Its strong typing is not only stronger than Pascal, C and C++, but it also carries over into separately compiled modules. Ada's system of package specifications is light years ahead of C's kludgey "header" files, which haven't progressed beyond the capabilities of mid-1960's Cobol. As a result of this strong typing, one can (and often does) make rather significant changes to the type structure underlying a large program and have the program continue to work without facing weeks of depressing crashes due to some obscure module interaction. Ada's strong typing can thus encourage a more aggressive stance during program maintenance and upgrading than can ever be tolerated in large-scale C programs. Your special issue covers well the other advantages of the Ada language, such as abstract datatypes and overloading.

2. *In retrospect, the creation of Ada was an incredibly audacious task, which I liken to the task of building the U.S. Interstate Highway system.* The thought that a single language could satisfy all the diverse software needs of the U.S. Department of Defense, whether for processing spread sheets or spread spectra, was incomprehensible. Like the Interstate system, huge amounts of dirt had to be thrown around, and great political battles ensued over who would get the access roads, rights-of-way, and interchanges. It was an accepted principle that the highways would connect up the home cities and states of the most powerful Congressmen. Like the Interstate system, the roads were built for high speed travel—so long as you wanted to go where they went. Unfortunately, also like the Interstate system, a large number of planned interchanges never got built as a result of political infighting and/or budget constraints, thus guaranteeing that certain expensive roads and interchanges would go nowhere! Thus, using Ada today has much in common with commuting to work on certain parts of the U.S. Interstate system—one can zip along at high speed for quite a while before being shunted off onto a state road (or worse) to face 23 unsynchronized traffic lights in a row before getting back on the highway again. In some cases, an exit will terminate in a dead end, in which case you may have to turn on the 4-wheel drive and go through someone's back yard to get to the next entrance ramp. Using C is more like driving on an old secondary mountain road from the 1930's with pot-holes and no guard rail; it isn't as fast, it isn't as safe, and it doesn't have the amenities, but at least the route is continuous and well-marked.