THE CASE FOR LARGE LANGUAGES

Conrad Weisert
Information Disciplines, Inc.
1205 Madison Park
Chicago, Illinois 60615

ABSTRACT

Large programming languages like PL/I and Ada are in disfavor with the methodology establishment these days, being viewed as hard to learn, cumbersome to use, and costly to implement. Our growing body of experience with smaller languages like Pascal and C, however, raises questions about their ability to handle essential programming tasks in a natural and effective way. The issues are further clouded by confusion over the roles of both popular "mainstream" languages like Basic and Cobol and so-called "fourth generation" languages.

What does "large" mean? A language can be "large" in different ways:

1. By encompassing in its intended scope facilities for handling a wide range of programming needs.

2. By allowing a high degree of generality of expression.

3. By providing, either initially or through incremental growth, numerous facilities for specialized situations.

Condemnations of large languages should be aimed mainly at the third category, the messy hodge-podge of miscellaneous "features". The other types, on the other hand, can be viewed positively, not only for programming tasks that require a large subset of their facilities, but especially for their support of the aims of structured programming. Before we consign such languages to oblivion, let's identify the areas where they're more effective than small languages and evaluate the trade-offs in giving up those capabilities.

We'll examine the background of these issues, and discuss pros and cons with examples from real-life programming situations.

reprinted from STA-5 Proceedings 270
May, 1988

1. INTRODUCTION

1.1 Conventional Wisdom

The structured revolution is over. We no longer argue over whether structured methods are superior to other approaches. Arguments over programming languages, however, continue much as before (except for some of the names), often without regard for the very principles of structured programming that are now, in theory, universally accepted.

Conventional wisdom now favors "small" programming languages, like Pascal and C, that can be learned quickly and implemented cheaply. In contrasts "large" programming languages, like PL/I and Ada, are in disfavor, viewed as hard to learn, cumbersome to use, and costly to implement. Conventional wisdom further maintains that good languages are the creation of an individual or two rather than the product of a bureaucratic organization. Large languages and the organizations that spawned them have become objects of ridicule in some circles. The mere mention of one of these "dinosaur" languages often provokes knowing snickers from the cognoscenti. When is comes to programming languages "large" is our "L-word".

The flaw in the conventional wisdom is that we do more with programming languages than just build compilers, write textbooks, and teach courses. We also have to *write programs* to solve problems. That's where a growing body of experience is raising doubts about the small languages' ability to handle essential programming tasks effectively.

1.2 Catering to the Programmer

Although anyone may express opinions about programming methods and tools, the final judge must be the *programmer* who uses them. No one who lacks direct recent experience in building software can fully appreciate the worth of some handy feature or the penalty from some frustrating limitation. By "no one" I mean to include many "experts" on languages and methodology, people who may come into contact with programmers, who write a little code from time to time as examples for books or courses, and who may at one time have spend their days doing real programming. It is all too easy to forget how programming really is in the enthusiasm of prescribing how it should be.

Alas, working programmers are seriously underrepresented in setting standards, in establishing curricula, in writing books, or in otherwise influencing opinion. Even inside their own organizations programmers often have little to say about the tools they must use or about the terms of contracts under which they must work. Except for some user groups there are few channels for programmers to convey their desires and frustrations back to the bodies that control the evolution of languages.

For this discussion, anyway, let's restore the balance. In examining the issues I want to consider almost solely the point of view of the programmers. However much some language characteristic may appeal to (or offend) an MIS manager, a compiler vendor, a teacher, or some internationally known "methodologist", the ultimate arbiter here is the experienced and competent working programmer.

2. WHAT IS A LARGE LANGUAGE?

2.1 Aspects of Language Size

As the term is used by critics and supporters a programming language can be large in several ways:

1. By encompassing in its intended scope facilities for handling a wide range of programming needs.

2. By allowing a high degree of generality of expression.

3. By providing, either initially or through incremental growth, numerous facilities for specialized situations.

We all concur in condemning a language that is large in the third sense, the hodge-podge of miscellaneous features and legalistic rules (although we may disagree over whether a particular language exhibits that property). The other two characteristics, however, can be seen as positive, especially for programming tasks that require a large subset of the facilities. Before consigning all large languages to oblivion, we should identify the areas where they're more effective than small languages, and evaluate the trade-offs, if any, in giving up those capabilities.

2.2 The Special Case of Cobol

The most widely used language for large-scale business applications doesn't fit cleanly into either category. Uniquely among widely used languages Cobol can be viewed as both large and small. It exhibits many disadvantages of both and offers few advantages of either.

Cobol is small in the first two senses. It lacks capabilities we need for routine programming tasks, and the features it does offer are often limited by strange restrictions. Surprisingly, since "business oriented" is embedded in its very name, some of Cobol's most serious limitations lie in its inability to handle situations that arise in everyday *business* applications. In the third sense, however, Cobol is a huge language. It imposes a gigantic repertoire of rules that are hard to remember and clumsy to apply.

Even PL/I's severest critics concede that it provides immensely more capability than Cobol. Yet a typical Cobol manual contains about the same number of pages as a PL/I manual. How can this be? Because a large part of any Cobol manual is devoted to explaining endless special cases, restrictions, limitations, and combinations thereof. If we assigned a team of mediocre lawyers to specify a programming language, they'd undoubtedly come up with something like Cobol.

Since so many Cobol rules tell programmers what they may *not* do, we might think of Cobol as a *negatively large* language. This lets advocates of small languages disown Cobol. But since some of its limitations are typical shortcomings of small languages, advocates of large languages can also disown it. I shall, therefore, cite some Cobol examples as illustrations of small language deficiencies.

3. THE IMPACT OF MISSING FEATURES

3.1 Background and Examples

The original Algol report [1] described a then state-of-the-art programming language without reference to any *input-output* capability. Since few, if any, programs run without requiring input or producing output, the world was thus presented with a new standard language in which it was impossible to write any useful complete program.

Algol partisans gave different explanations for this curious omission. A common one was that input-output is too complicated and too dependent on specific hardware or operating system architecture. A more revealing argument was that input-output facilities are messy and that building them in would have contaminated Algol's beauty and elegance. Whatever the reason, the world had a general-purpose programming language that lacked a fundamental facility everyone agreed was essential for writing any serious complete program. We may thus regard Algol-60 as the spiritual father of small languages.

We've had many more examples since then. For example:

- C and Modula-2 echo Algol-60's snobbery toward built-in input output. "Input and output facilities are not part of the C language." [2, p.143]
- Some of the most popular artificial intelligence languages either can't do everyday arithmetic expression evaluation (perhaps also to avoid compromising their purity of concept) or require some awkward and unnatural artifice to do so.
- Many purportedly general-purpose languages don't include *character strings* as built-in data types. They include C, as Paul Abrahams [3] recently noted in a persuasive appeal to correct that shortcoming. Some other languages, like Pascal and Cobol, support character strings so crudely that it's almost impossible to write even the simplest general-purpose routines to operate on them.

None of these facilities – input output, expression evaluation, or character strings – is in any way exotic, sophisticated, or specialized. Programmers need them in everyday, mundane tasks in a full range of applications, business or scientific, real-time or batch, procedural, applicative, or rule-based.

3.2 Reaction

How ought a programmer to react upon discovering that the language he or she is using lacks some feature needed for solving a problem? Ideally, we'd want the programmer to conclude that the problem is beyond the scope of that language and, therefore, to choose some other language for that assignment. In practice, however, that happens only rarely. Language commitments are often too deep to be abandoned for mere reasons of inadequacy.

Instead the programmer will usually do one of three things:

1. Develop (or obtain) "library" subroutines, written in some other language, to provide the needed features.

2. Contrive illegal, implementation-dependent techniques to circumvent the limitation.

3. Distort the program design so as to avoid the needed feature, often with fatal effect on structured programming principles.

Any of these reactions may impose a huge cost, one that continues its impact long after the programmer has finished the assignment.

3.3 "Library Routines" versus Language Features

Just about all implementations of C and Modula-2 are accompanied by a set of "standard" library routines, including all needed input-output facilities. Such routines are described in textbooks and courses on those languages. If they're so standard, we may ask, are they really different from language features in any practical sense? Why should a programmer care whether a feature he or she uses is officially part of the language or an add-on library routine, as long as it's available when needed?

In many instances, of course, we don't care. A programmer learns a repertoire of techniques, which includes the use of language elements, "standard" modules, and local modules. But in two crucial cases the impact is very real.

First, the programmer feels helpless when something doesn't work as expected. For example, a respected text [4, p.123] on Modula-2 explains:

> "Modula-2 tackles the input/output problem somewhat along the lines of Algol-60 in that there are no input or output *statements* in the language. Instead a library module containing input and output *procedures* may be provided. It is intended that each implementation of the language provide the same capabilities for input and output, although the actual implementation may differ substantially from one computer system to another. This approach . . . differs from Pascal in that the language definition does not say what these procedures must be. . . One set of procedures can be designed that provides capabilities almost identical to those of Pascal."

I myself recently had the awful experiences of needing a simple input routine in Modula-2. We merely needed to get lines of text, an operation that's trivial in Cobol, PL/I, and even Basic.

Pascal's READ procedure will do the job nicely, but our "almost identical" Modula-2 procedure would not. A **`ReadString`** library procedure seemed promising until we discovered that it considers every blank a line terminator! But my complaint was brushed aside by the local Modula-2 guru: Don't blame the language, don't blame the compiler, we're not responsible; it's only an "implementation-dependent" library routine. I never learned who, if anyone, is responsible for such routines, nor could I track down an authoritative opinion on just how the procedure is *supposed* to work. Even after coding and debugging a messy procedure of my own to circumvent the problem, I had no confidence that it would still work in a different Modula-2 environment.

In that one rather trivial example we paid a number of penalties. First we wasted a lot of time in ultimately futile trial-and-error activity. Then we had to code, debug, and document a procedure that we'd never have needed had we been using a language with built-in I-O. Finally, the users may encounter future conversion and maintenance costs as a result of our program's doubtful portability. These are *real* costs, not matters of convenience of esthetics.

A second case where we feel the difference between library routines and built-in language features arises when something is impossible to implement through procedures or subroutines, i.e. because it requires information only a compiler could know. In the same Modula-2 assignment I was just describing, we needed to display some intermediate values for debugging, something programmers need to do almost every day. Amazingly, even this turned out to be a major challenge, because each of the standard output routines is specific to one particular *data type*. The programmer codes **`WriteInt`** to print an integer, **`WriteChar`** to print a character, and so on, and strings these together to create the usual type of printed line we'd get from a Fortran or PL/I FORMAT specification.

So what was the problem? The central philosophy of Modula-2 is built around hiding the very same data types that these output procedures insist upon knowing. I was working with objects whose type was defined, as good practice dictates, in a separate module. Modula-2 wouldn't permit me to pass such data to the "standard" output procedures, because the data attributes didn't match. Note the blatant conflict with the aims of structured design: The very language facilities intended to help us localize knowledge of data representation also force us to spread that knowledge throughout our program!

The "solution" recommended by our Modula-2 guru was that whenever we define a new type we should also build specialized output procedures for objects of that type. Thus programmers are faced not only with a lot of extra coding, but with having to debug their most basic debugging tools! What can be more wasteful and frustrating to a 1980s programmer than to be forced to spend valuable time confronting problems that were fully solved back in the 1960s?

3.4 Illegal Techniques

Programmers go to ingenious lengths to circumvent the lack of some essential feature, often disregarding the rules of the language upon finding something that works in a particular implementation. Some of these traditions become to deeply established that programmers no longer think of them as "illegal".

A notorious long-running example of this phenomenon was the development of techniques for manipulating character data in Fortran II, a *small* language that supported only numeric data. In the early 1960s even the ACM Communications [5] saw fit to publish a seemingly endless series of techniques (and corrections to previously published techniques) for fooling compilers into interpreting floating-point numbers on the IBM 7090 as groups of six characters. A few years later third-generation computer architectures no longer supported the 36-bit word, and none of the thousands of Fortran programs that had made use of these techniques could be run in the new environment without major, costly change.

Once in a while such a technique may get legitimized as a standard language feature and become a positive contribution. Most often, however, it only serves to limit compatibility and greatly increase ultimate conversion costs for users.

3.3 Distorting Program Designs

Undoubtedly, the most costly impact of missing features occurs when a language steers programmers toward inappropriate, hard-to-maintain program designs. A very common example originates in Cobol's prohibition against passing a file name as a subprogram parameter. That restriction not only makes it impossible to write any sort of generalized I-O routine, but also encourages programmers to embed their highly specialized file manipulation in grotesquely oversized "main" programs.

This effect can be clearly seen in the majority of Cobol textbooks and courses, which apply the term "structured" to a hideously monolithic style of program organization in which a massive global DATA DIVISION goes on for pages and pages. That sort of "watered down structured programming" yields, for the Cobol world, only a small fraction of the potential benefits of the structured revolution.

The global cost over the past twenty-five years of this one omission can easily be estimated in billions of dollars. Think of how many times Cobol programmers have struggled to code and debug a basic sequential file update from sorted transactions. Think of how many bugs in those programmers have slipped into production to cause havoc months later. Think of how many programmers have laboriously coded report-printing logic to count lines, handle pagination, and manage multiple-level control breaks. We can't assume that they would have used generalized modules for all or even most of those thousands upon thousands of instances, but surely *some* of them would have done so had their language only permitted it.

But the negative impact goes beyond just the huge cost of redundant programming, unnecessary debugging, and hideous maintenance problems. Many programmers, maybe a majority, whose first language is Cobol never come to perceive these limitations as a serious problem. For them such needs never arise; that's just the way programming is. The notion of independent subprograms to do I-O is foreign to them. When first presented with a second language that imposes no such restrictions they wonder why anyone would ever want to write a generalized I-O subprogram, and confidently declare that there'd be no benefit in doing so [6].

This kind of blind spot, the so-called "Cobol mentality", can afflict programmers throughout their careers, whether or not they continue to use Cobol. The blond-spot phenomenon, of course, isn't limited to Cobol, but Cobol programmers often exhibit its most extreme form.

4. IMPACT OF STRONG TYPING

4.1 Automatic Data Conversion

In the earlier Modula-2 example we needed to display intermediate values for debugging. The programmer would just like to say, "Print this *thing*, whatever it is." With a few exceptions a PL/I **PUT LIST** or a Basic **PRINT** does this trivially. But a language with strong typing, like Modula-2 or Pascal, offers no obvious way to *build* the facility we need. Thus the very languages that lack built-in input-output also impose additional obstacles to out building the needed facility ourselves.

Upon hearing my sad tale a colleague blinded by years of Cobol programming observed smugly that his language's DISPLAY statement would do exactly what I needed. He was wrong. Consider this misleadingly straightforward code from an actual program:

```
PERFORM UNTIL END-OF-FILE = YES;
     ADD 1 TO RECORD-COUNTER;
     PERFORM PROCESS-RECORD;
     READ INFILE RECORD
          AT END MOVE YES TO END-OF-FILE; END-READ;
END-PERFORM;
CLOSE INFILE;
DISPLAY  RECORD-COUNTER    'Records processed.';
```

How many programmers realize, until they try it, that the above statement won't print the value of the numeric (**COMPUTATIONAL**) variable **RECORD-COUNTER**? IBM's implementation handles this the way we'd like as an unofficial "language extension", but the more orthodox VAX [7] manual explains, in the second of seven "general rules" for the **DISPLAY** statement: "No editing or conversion occurs during **DISPLAY** execution."

Consider what this means. The statement Cobol provides for the purpose of displaying values will not display ordinary numeric data in a readable form! Nor need a compiler give any warning; that **DISPLAY** statement was *legal*. The program just moves the data item into the line to be printed and "prints" the indecipherable bit pattern. Now if we surveyed a million programmers, would we turn up even one who favors the way **DISPLAY** actually works? Who on earth was responsible for specifying such a useless and misleading interpretation? How recently have those people had to debug a program themselves?

Both the Modula-2 and the Cobol examples illustrate the value of one of the most controversial language features: automatic conversion of data to conform to the context in which it is used. In this sense, C is much more reasonable than Pascal or Modula-2, since it, like many large languages, handles many conversions automatically in the manner many programmers would prefer.

It is fashionable to condemn automatic data conversion as a source of traps for the unwary and of hidden performance degradation, but such arguments are unconvincing. Yes, we can devise examples that yield bizarre results, but they arise rarely in actual disciplined programming. Yes, an undeclared or misdeclared variable can trigger costly run-time conversions, but compiler-generated warnings and attribute listings bring them to the attention of the careful programmer.

To add to the programmer's frustration, some of the same languages that refuse to perform implicit data conversion also impose obstacles to the programmer's doing it explicitly. If Cobol supported function references, for example, the programmer could code something like:

```
DISPLAY  DISPCONVERT (RECORD-COUNTER)  'Records Processed';
```

But what a Cobol programmer must actually do is (a) code a separate **MOVE** statement to effect the conversion and (b) declare a special variable just for that intermediate result in the (physically distant on the listing) **DATA DIVISION**. Needless to say, this offers opportunities for error, impairs program readability, and clashes with the central structured design criterion of localizing knowledge.

4.3 Default options

Related to the automatic conversion problem and equally controversial is the application of *default* assumptions to incompletely specified data declarations or other program elements. Again we could devise examples where what one doesn't know can hurt one, but in general sensibly chosen defaults only help simplify the programmer's tasks.

Ill-chosen defaults, of course, do carry a penalty. Cobol's assumption that every data item has **USAGE DISPLAY** unless specifically declared **USAGE COMPUTATIONAL** has led to immense performance degradation in thousands upon thousands of programmer written by careless or naïve programmers. This penalty should be blamed not on automatic conversion, which is desirable, but on the choice of the least efficient alternative as the default **USAGE** attribute.

5. IMPACT OF LACK OF GENRALITY OF EXPRESSION

I recall vividly a principle of language design I first heard in 1962 in a class taught by the staunch Algol supporter and Fortran critic Alan Perlis:

- Wherever a language allows a constant, it should also allow a variable.
- Wherever a language allows a variable, it should also allow an expression.

Three years later and early PL/I manual [8, p. 10] put it more generally: "If a particular combination of symbols has a useful meaning, that meaning is allowed."

Fortran was a major violator of these principles from the start, e.g.:

- Allowing **J+1** as a subscript, but not **1+J**, **J+K**, or **J(I)**
- prohibiting variable field-widths in **FORMAT** statements

Today, of course, Cobol takes the prize for arbitrary and pointless restrictions. For example:

- Expressions are prohibited in just about every context except the **COMPUTE** statement and some conditional expressions. (This restriction alone accounts for cluttering up the **DATA DIVISION** with all sorts of otherwise unnecessary intermediate items.)

- For years we could only test conditions, not set them; now (in a major breakthrough) we can set them to **TRUE** but not to **FALSE**.
- Table dimensions (**OCCURS**) can't be specified at level-1, and **VALUE** can't be specified directly for table elements.
- We can factor the **USAGE** attribute for a group data item, but not the **PICTURE** attribute.

In *The Psychology of Computer Programming* [9] Gerald Weinberg examines the multiple penalties we incur in using languages that permit little generality expression. Programmers waste time looking up details in manuals. They waste their own and computer time inadvertently violating restrictions and correcting their "mistakes". Worse yet, Weinberg points out, they fall into bad coding habits trying to avoid half-remembered restrictions that may not even be real ones. Worst of all, they often carry over these peculiar techniques into their use of less restrictive programming languages.

Although any individual instance of this phenomenon may seem minor, the aggregate cost is significant, even within one large program.

Lest we assume that such problems are limited to *old* language like Fortran and Cobol, recall the extended exchange of letters in the ACM *Communications* Forum a couple of years ago on the resurrected **GO TO** controversy. Writer after writer submitted "solutions" to a coding problem posed by the original contributor. Many were in Pascal and were based on nested looping with various housekeeping variables. A PL/I version [11], however, needed only a single loop and APL (as usual) needed none. The difference was due to Pascal's lack of generality. The large language PL/I expressed simply and directly what had to be laboriously proceduralized in the small language Pascal.

These aren't esthetic, theoretical arguments, but very real practical ones. There's no greater waste than coding and debugging some pattern of code for the hundredth time. Few program characteristics impair readability and maintainability more than a high ratio of internal housekeeping code to the actual application problem solution.

6. IMPACT OF LACK OF EXTENDABILITY

Sometimes programming language deficiencies can be overcome by building higher-level tools in the language itself. *Functions*, *macros*, and *data types* are useful and natural facilities for defining new language elements in terms of existing ones. Cobol, the language most in need of being extended, lacks all three. Of the major procedural languages, PL/I (which needed it the least) was the first to provide its own macro capabilities. Of the newer languages, C offers macro facilities, avoiding the dead end of Pascal and holding out hope for the future usability of at least that one "small" language.

In Cobol we can do a little "language extension" with CALLable subprograms and a little with **COPY REPLACING**, but few Cobol programmers perceive the results as significantly easier to use than plain Cobol. Besides, other Cobol restrictions severely limit the possibilities. Because of Cobol's strict division structure, for example, we can't cleanly package a COPY module of procedural code that needs a local data item. In this most popular programming language, almost all programming tasks start just about from scratch with bare Cobol.

In Pascal or Fortran, of course, we routinely build up function definitions to raise the level of the language. Even there, however, we run into frustrations when some other language obstacle precludes implementing a needed facility. Imagine trying to build even a simple SWAP statement without macros. Imagine trying to develop an array-sort module in a language that insists on knowing the attributes of the array elements down to a fixed string length. Imagine trying to package logic for updating a sequential file from sorted transactions either in a language that lacks record structures or in a language that won't allow procedure names to be passed as parameters.

In some languages we can get around *some* of these obstacles to build up higher-level language constructs. The only languages I know of today in which we can get around *most* of them are *large* languages.

7. CATERING TO THE COMPILER'S CONVENIENCE

Pascal was supposed to support structured programming. Indeed, Pascal typifies for many people the very essence of what "structured" means. Yet some of Pascal's limitations run directly counter to the spirit of the structured revolution.

Consider, for example, the sequence in which modules (procedures and functions in Pascal) appear on the program listing. A fundamental tenet of structured programming is that one should be able to read a listing sequentially in something approximating *top-to-bottom* sequence. The higher-level *control* routines should appear before the lower-level *action* routines they invoke. Pascal, however, imposes exactly the opposite discipline; modules have to appear in bottom-to-top sequence! The reader is confronted with all the details before seeing any of the big picture.

The explanation for that restriction has nothing to do with considerations of programming style, readability, or "structure". Pascal imposes this requirement solely for its own convenience, to that information for a one-pass compiler will be available when first needed.

"Reserved word" lists, too, have little to do with good programming practice. While most of us would frown upon the frivolous use of keywords as data names – **`MOVE MOVE TO TO`** -- what programmer has never run afoul of the long and continually growing lists of forbidden names in Basic, Pascal, or, especially, Cobol? Every new version of the Cobol standard sets off an international uproar over the need to expunge or convert conflicts with the newly added reserved words in thousands of existing, formerly legal programs. Some organizations, in an effort to avoid such problems permanently, have adopted standards that actually prohibit the use of any common English word or hyphenated pair of common words as a data name!

8. "FOURTH GENERATION" LANGUAGES – WHERE DO THEY FIT?

I dislike the term "fourth generation language", both because it's historically misleading and because it implies to the gullible that all procedural languages are obsolete and their use unenlightened. Despite the confusing terminology, some of these high-level (and usually proprietary) tools are useful and powerful and deserve a place in the repertoire of any professional programmer.

It's distressing, however, that the designers of some of these newer languages learned so little from the mistakes and the successes of the so-called "third generation" languages they propose to replace. There's little excuse for modern tools that, despite their power, embody some or all of the shortcomings of the small (or just bad) languages that we've been discussing. There's no excuse for modern tools that ignore the lessons of the structured revolution.

The more easily we can extend our current languages, the less we'll need completely new languages. Using PL/I's preprocessor, for example, programmers have built impressive facilities that we can view as specialized languages having properties like "very high level", "user-oriented", or "application specific". With such capabilities we're not forced to draw a rigid line between old and new tools or between language *generations*. Instead programmers can draw upon a continuous range of facilities of various levels targeted to a broad range of situations. It's regrettable that most of today's new tools offer no capability at all for this kind of extension.

Bruce Rosenblatt [11] has noted an order-or-magnitude productivity ratio between PL/I (used in an enlightened way) and both Fortran and Cobol. This impressive factor is consistent with my own observations in a number of organizations. Note that this productivity ratio qualifies PL/I as a *bona fide* "fourth generation language" according to James Martin's [12] criterion that "its users obtain results in one-tenth the time with Cobol or less."

9 CONCLUSION

We've looked at a number of shortcomings of popular small languages that have been successfully overcome in large languages. The consequences of struggling to use small languages that exhibit such shortcomings are extremely severe. It's naïve to dismiss these problems as mere issues of convenience or esthetic preference. Their impact on every non-trivial computer program is enormous.

Many of these shortcomings, furthermore, are in sharp conflict with the basic aims of structured design and coding: localization of knowledge, sequential readability, avoidance of redundant effort. Even some languages that provide all the approved structured flow constructs impose severe obstacles to organizing and developing real programs. Many organizations and individuals using these inadequate tools mistakenly believe that they're practicing a "structured approach", even though they've realized few benefits from the structured revolution.

The pioneering large languages are now dead (Algol-68) or dying (PL/I), and there's no realistic way of resurrecting them. We've lost forever the opportunities those languages once offered us. They were born too soon, in an era when inertia and fear of change were the dominant forces shaping our industry's choices of methods and tools. Their example, however, remain as relevant lessons. We need to heed them, both in choosing among existing tools and in developing new ones. Programmers must demand better.

REFERENCES

[1] P. Naur (Ed) et al: "Report on the Algorithmic language ALGOL 60", *Communications* of the ACM, March, 1960.

[2] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*, 1978, Prentice Hall

[3] Paul Abrahams: "Some Sad Remarks about String Handling in C", ACM SIGPLAN *Notices*, October, 1988.

[4] Gary A Ford and Richard S. Wiener": *Modula-2 – A Software Development Approach*, 1985, Wiley.

[5] Conrad Weisert: Letter to the Editor, *Communications* of the ACM, August, 1962

[6] "Cobol Coped With" (letter to the editor), Datamation, November 1, 1970

[7] VAX-11 Cobol Language Reference Manual, Digital Equipment Corporation (AA-H631B-TE), 1982

[8] PL/I Language Specifications, IBM (C28-6571-0), 1965

[9] Gerald Weinberg: *The Psychology of Computer Programming*, 1971, Van Nostrand Reinhold

[10] Conrad Weisert: Letter to the Editor, *Communications* of the ACM June, 1987

[11] Bruce Rosenblatt: "The Successors to Fortran – Why Does Fortran Survive?", *Annals of the History of Computing*, January, 1984

[12] James Martin: *Application Development without Programmers*, 1982, Prentice Hall.