The Relationship Between COBOL and Computer Science

BEN SHNEIDERMAN

Based on interviews, reviews of the literature, and personal impressions, the author offers historical, technical, and social/psychological perspectives on the fragile relationship between COBOL and computer science. The technical contributions of COBOL to programming language design are evaluated. Five proposals for computer science research on COBOL and fourth-generation languages are described.

*Categories and Subject Descriptors: D.3.0 [**General**]—standards; D.3.2 [**Language Classifications**]—COBOL; K.2 [**History of Computing**]—COBOL, software; K.3.2 [**Computers and Education**] Computer and Information Science Education—computer science education*
General Terms: Design, Languages, Standardization

For a computer scientist to write sympathetically about COBOL is an act bordering on heresy. It requires courage because academic colleagues and data processing professionals are both likely to be suspicious of my motives. Therefore, I feel my first obligation is to make clear my intention and orientation.

I believe that COBOL has had a strong and largely positive influence on the emergence of computer usage. The development of COBOL was a pioneering effort that advanced the state of the art in practical data processing and language design. COBOL clearly had many flaws, some of which have been overcome by revisions to the original language. Designers of other languages have learned from the mistakes and overcome the problems with novel constructs. This paper offers three perspectives on the rise of COBOL (historical, technical, and social/psychological) and suggests directions for future cooperation.

My orientation is as a computer scientist whose early work was in database systems and programming. More recently I have turned my attention to psychological or human-factors issues of programming, database query facilities, and human-computer interaction (Shneiderman 1980). I have taught introductory programming courses in FORTRAN, BASIC, Pascal, COBOL, APL, and PL/1 and have written or coauthored textbooks using the first three of these languages.

Author's Address: Department of Computer Science, University of Maryland, College Park, MD 20742.
© 1985 AFIPS 0164-1239/85/040348–352$01.00/00

Historical Perspective

Five aspects of the historical development of COBOL can be traced as important influences in the alienation of COBOL from the computer science community. First, academic computer scientists did not participate in the design team. The developers of COBOL were from the commercial community: the manufacturers and users of large data processing systems in industry and government (see the minutes in this issue for lists of the participants).

In 1959–1960 very few academics could have been classified as computer scientists, of course, and few of them could have made useful contributions, but engaging them might have been beneficial.

The developers of the Ada language realized this possibility and made ambitious and successful efforts to elicit the participation of academic and industrial researchers. Computer scientists can do more than contribute ideas; they are often effective in disseminating new concepts through their publishing, teaching, and lecturing efforts.

The second historical aspect is that the COBOL developers apparently had little interest in the academic or scientific aspects of their work. The May 1962 issue of the *Communications of the ACM* was devoted to 13 papers describing COBOL and related issues. Every article was written by an industry or government person. These people did not have the academic frame of mind that involves referencing previous and related work; only four of the papers had any references. Sociologists of science who use citations to trace the flow of ideas would recognize this pattern as an indicator of intellectual separatism.

The third aspect is the decision of the COBOL developers not to use the Backus-Naur Form (sometimes called Backus Normal Form) notation as the metalanguage to describe COBOL. The COBOL developers were unaware of this work, which appeared in a conference report in June 1959. I don't see the failure to use BNF as central, but apparently the criticism at the time was severe (Sammet 1981, p. 233). The COBOL style of metalanguage has become widely used and might be considered as an important contribution.

A fourth concern is the process of describing COBOL to the academic and industrial community. Professionals could learn the language from programmer reference manuals, but a well-written book emphasizing the conceptual foundations of COBOL would have been an asset. It took a few years before successful introductions were available (McCracken 1963; Saxon 1963) to teach COBOL to novices. Note the contrast with the dissemination of Pascal. Niklaus Wirth published a precise description of the language in *Acta Informatica* (1971), an interesting book titled *Systematic Programming: An Introduction* (1973), the widely read *Pascal User Manual and Report* (1975), and a forward-looking textbook *Algorithms + Data Structures = Programs* (1976).

A search of the Library of Congress SCORPIO system revealed 252 books indexed under COBOL, 529 under FORTRAN, and 1054 under BASIC, demonstrating the relatively lower rate of publication on COBOL. Pascal is more recent, but already 208 books are indexed under that programming language.

The fifth historical aspect is that the people who might have accepted the title of computer scientist in 1960 were not interested in the problem domain of COBOL programs. The commercial file-processing problems were remote from the concerns of computer scientists, who generally dealt with numerical analysis, physics, engineering problems, and systems programming.

In summary, the people and the problems in the COBOL world were very different from the people and problems who were laying the basis for computer science. It is not surprising that they worked separately and in parallel. The development of computer science and data processing might have been substantially altered if these communities had taken the time to meet and work together.

Technical Perspective

Getting convergence on the technical successes and failures of COBOL was a difficult task. I spoke to 30–40 computer scientists and data processing professionals in trying to sort out the issues. The following analysis is my own view guided by the interviews.

First the successes. The strongest point of agreement was that COBOL contributed the record structure, explicit file structure definition, and the separation of data definition from procedural aspects. The COBOL record and file structure certainly influenced the design of PL/1 and Pascal. The notion of an aggregation of dissimilar items is a major advance over the FORTRAN array. The Pascal notion of variant records can be traced to the COBOL REDEFINES clause.

Explicit file structure definitions that included a hierarchy of names for fields and the separate DATA Division were the predecessors of the database management system (DBMS) concept. DBMSs are a vital part of computer science and are the source of a rich theory that is still developing rapidly.

Another contribution was the diverse set of control structures, which were quite sophisticated for the time. The COBOL IF-THEN-ELSE, in spite of its awkward scope delimiter rules, reduced the need for GOTOs and permitted the creation of more comprehensible code. The variety of PERFORM statements allowed convenient and powerful looping and some degree of modular design. The ease with which paragraphs could be created, named, and used facilitated hierarchical design.

A popular feature of COBOL that has appeared in other languages is the COPY statement. By including groups of statements from a library, organizational standards were easily enforced, programmers were encouraged to cooperate, and reuse of code became convenient.

Another interesting COBOL concept is the ENVIRONMENT Division, which allowed users to specify machine or compiler dependencies. It permitted definition of separate host and target computers, thus foreshadowing the idea of cross-compilers.

Finally, the COBOL community demonstrated the power of portability and standardization. In spite of some local variations, COBOL is largely machine independent. Programmers could successfully transfer their knowledge and often their programs from one

organization to another. Standardization also encouraged the development of software tools and reuse of code.

Some of the perceived technical failures of COBOL might have been avoided by early consultation with computer scientists, but other problems would not have been recognized until the early 1970s. The most serious omission is a function or procedure definition facility with parameters and local scope of variables. The original version of COBOL has only global variables; therefore a generic subroutine to sum the elements of an array, search a string, or print a bar chart was not easy to write. Two programmers working together had to coordinate carefully to ensure that they did not inadvertently both use the same variable name for different values. The lack of protected module boundaries also allowed complex and sometimes dangerous programming techniques. A DEFINE macro feature was in the original language description and appears to be the first such facility in a high-level language. Unfortunately, it was never implemented and was eventually dropped from the language. The 1974 revision to COBOL included the now-popular CALL USING feature, which permitted parameters and run-time creation of procedure names.

Computer scientists often complained about the wordiness of COBOL statements and the clutter of the optional noise words. The designers of COBOL apparently believed that the English-like statements would make programs readable by managers or other nonprogrammers, but the semantics of programming are at least as challenging as the syntax. Computer scientists whose background emphasized mathematical notation, which is precise and concise, felt that COBOL was too wordy and somehow unscientific.

I found and was sympathetic to several complaints about COBOL control structures. The scope of an IF-THEN-ELSE is delimited by a period, which is often missed by programmers when reading and even when writing programs. It might have been better to have used a keyword such as ENDIF. The PERFORM statement cannot contain a list of statements, only the name of a paragraph. Studying a program is tricky because the reader must hunt for the body of the PERFORM statement. With short loops, the overhead is annoying, and confusion can increase.

Poor string-handling facilities were cited by several people as a major weakness of early COBOL. Moving and copying of strings was convenient, but insertion and deletion of characters inside a string was difficult. The EXAMINE verb and the TALLYING and REPLACING options were included in the early COBOL specifications to facilitate plans to write a COBOL compiler in COBOL. The 1974 version of COBOL contained the somewhat more powerful INSPECT command, plus STRING and UNSTRING.

Several computer scientists complained about the lack of recursion in COBOL, but I think that it hardly would have made a difference in the use of the language.

Knowledgeable COBOL programmers in my survey had other small complaints, but I did not judge them to be vital.

Social/Psychological Perspective

Now we move on to some more speculative areas that reflect on the fundamental differences between the computer science and business data processing communities. The rejection of COBOL by most computer scientists is a product of their desire to avoid the business data processing domain, their pursuit of mathematically oriented theory, and often their lack of knowledge about COBOL.

When asked for his comments, one computer scientist who is a widely respected expert in programming languages boldly replied, "What's COBOL?" His world did not have a place for COBOL. In fact, several programming language texts (e.g., MacLennan 1983) do not include COBOL in the index. Another responded, "It's terrible . . . ugly," but had difficulty explaining why. I suspect this prejudice emerges from the bias of many computer scientists against the problem domain and the wordy, nonmathematical style of COBOL, rather than from any serious consideration of the technical weaknesses. Dijkstra (1982) wrote, "COBOL cripples the mind," but he was equally harsh on FORTRAN ("infantile disorder"), PL/1 ("fatal disease"), BASIC, and APL. Tompkins (1983) sought to defend COBOL, and Reid (1983) responded with many legitimate criticisms.

The bias against the problem domain is stated explicitly in Pratt's programming language textbook (1975; 1984), which says that COBOL has "an orientation toward business data processing . . . in which the problems are . . . relatively simple algorithms coupled with high-volume input-output (e.g. computing the payroll for a large organization)." Anyone who has written a serious payroll program would hardly characterize it as "relatively simple." I believe that computer scientists have simply not been exposed to the complexity of many business data processing tasks. Computer scientists may also find it difficult to provide elegant theories for the annoying and pervasive complexities of many realistic data processing applications and therefore reject them.

Tucker's programming language textbook (1977) has this evaluation: "We judge COBOL's programming

features as fair and its implementation dependent features as poor . . . its overall writing as fair to poor, its overall reading as fair and its data processing support as good. . . . [It has] tortuously poor compactness and poor uniformity." Not much to warm the heart of a COBOL programmer.

Several computer scientists remarked about the "trade school" nature of COBOL and that university professors did not like dealing with current practice, but sought to distinguish themselves with novel languages, theory, and an abstract, more mathematical orientation. One professor commented that he was "hostile to teaching what is used commercially," while a researcher sneered at the "folly of an English-looking language."

The desire to be aloof from current practice was a common theme and leads me to the

Theory Conjecture: Computer scientists like programming language theory, but find fault with any widely used language.

The Theory Conjecture should be comforting to COBOL supporters because it means that computer scientists will express displeasure for almost any programming language that is widely used. Since computer scientists desire to be with the state of the art, anything that is widely used must also be outdated. Commercial usage lowers academic prestige.

A related principle might be expressed as the

Egocentricity Conjecture: Computer scientists appreciate no programming language except the one that they design.

The role of a scientist is to innovate, so comments on other people's work are often in the form of criticism that lays the basis for a proposed improvement.

Summary

Jean Sammet's book on programming languages (1969) and her review of the history of COBOL (1981) offer lists of contributions of COBOL that are close to my own impressions:

- Readable language.
- Separate data declaration section with rich record structure.
- Decent control structures.
- Machine-independent, portable, standardized language.
- A useful alternative metalanguage.
- Creation of a successful and large community of data processing programmers.

In her review, Sammet (1981, p. 239) writes: "I personally do not see much language development . . . significantly influenced by COBOL," and goes on to say, "Most computer scientists are simply not interested in COBOL."

I do feel that COBOL was a major influence on PL/1, which was designed explicitly to include the popular features of COBOL, FORTRAN, and ALGOL. COBOL also had an impact on the use of record structures in languages such as Pascal and on the creation of database management systems.

Most important, COBOL greatly facilitated the enormous expansion of computer usage in data processing. The demand for programmers and computers benefited the entire industry and stimulated further scientific advances because there was such a large market for new ideas.

The Future: A Challenge to Computer Scientists

The story of COBOL is not over. The continuing changes to COBOL, refinements in design guidelines, and improvements in teaching strategies mean that the COBOL of today looks very different from the COBOL of 1960. COBOL and the so-called fourth-generation languages will still be around in 25 more years, and maybe computer scientists still have an opportunity to influence their evolution.

Let me propose five areas of beneficial COBOL and fourth-generation language research that might be of interest to computer scientists.

Code Optimization: There is a grand opportunity to apply traditional compiler optimization techniques to COBOL. Even more provocative would be to explore optimizations that are specific to the COBOL domain. Instead of eliminating redundant mathematical subexpressions, the COBOL compiler writers could concentrate on eliminating redundant MOVE or file operations. Global dataflow analysis would be a challenge in the COBOL context.

Formal Semantics: Precise descriptions of COBOL syntax are available, but there are variations in the semantics of some operations across implementations. Creating a formal semantic description of COBOL would be useful to implementors and might require novel techniques that could be applied in many programming language situations.

Maintenance Tools: The enormous and valuable libraries of COBOL programs are underutilized because tools are not adequate for indexing, searching, interpreting, and modifying this code. Library-science and expert-system concepts might be applied to making the voluminous "literature" of COBOL more readily available for reuse.

Programming-Style Research: Because COBOL is so widely used, we would see a substantial benefit if empirically tested style guidelines were available. Questions abound about the choice of mnemonic variable names, nesting of control structures, use of indentation, page formatting, modular design, commenting techniques, data structure design, etc. Empirical studies of program composition, comprehension, debugging, and modification by professional programmers would be valuable in resolving some of these issues and formulating a cognitive model of programmer behavior.

Software Engineering: In some ways COBOL is a convenient language as the target for a compiler or preprocessor. Indeed, numerous preprocessors attempt to offer higher-level control structures or data structures. How might procedural or data abstraction concepts be molded to fit the COBOL context? Can computer scientists offer an interesting theory of preprocessors to parallel the theory of compilers? Does large-system design in COBOL have features that are distinct from FORTRAN or Ada?

This list is only a starting point. COBOL and fourth-generation languages present many provocative challenges to computer scientists. Also, electronic spreadsheets such as VisiCalc and Lotus 1-2-3 are exciting innovations that have yet to be properly acknowledged in the computer science community.

Acknowledgments

In addition to the people I interviewed, I was greatly helped by knowledgeable reviews of early drafts by Robert Dewar, John Gannon, Rick Linger, Daniel McCracken, Terrence Pratt, Edward Reid, and Jean Sammet. My first consideration of this topic was stimulated by Hank Tropp and Jean Sammet's invitation to give a talk at the 1979 National Computer Conference's Pioneer Day celebration of the 20th anniversary of COBOL.

REFERENCES

Dijkstra, Edsger W. May 1982. How do we tell truths that might hurt? *ACM SIGPLAN Notices 17,* 5, pp. 13–15.

Jensen, Kathleen, and Niklaus Wirth. 1975. *Pascal User Manual and Report.* Second Edition, New York, Springer-Verlag.

MacLennan, Bruce J. 1983. *Principles of Programming Languages: Evaluation and Implementation.* New York, Holt, Rinehart and Winston.

McCracken, Daniel D. 1963. *A Guide to COBOL Programming.* New York, Wiley.

Pratt, Terrence W. 1984. *Programming Languages: Design and Implementation.* Second Edition, Englewood Cliffs, N.J., Prentice-Hall.

Reid, E. October 1983. Fighting the disease: More comments after Dijkstra and Tompkins. *ACM SIGPLAN Notices 18,* 10, pp. 16–21.

Sammet, Jean E. 1969. *Programming Languages: History and Fundamentals.* Englewood Cliffs, N.J., Prentice-Hall.

Sammet, Jean E. 1981. "The Early History of COBOL." In Wexelblat, R. (ed.), *History of Programming Languages,* New York, Academic Press, pp. 199–243.

Saxon, James A. 1963. *COBOL.* Englewood Cliffs, N.J., Prentice-Hall.

Shneiderman, Ben. 1980. *Software Psychology: Human Factors in Computer and Information Systems.* Boston, Little, Brown.

Tompkins, H. E. April 1983. In defense of teaching structured COBOL as computer science (or, Notes on being sage struck). *ACM SIGPLAN Notices 18,* 4, pp. 86–94.

Tucker, Allen B. 1977. *Programming Languages.* Reading, Mass, Addison-Wesley.

Wirth, Niklaus. 1971. The programming language Pascal. *Acta Informatica 1,* 1, pp. 35–63.

Wirth, Niklaus. 1973. *Systematic Programming: An Introduction.* Englewood Cliffs, N.J., Prentice-Hall.

Wirth, Niklaus. 1976. *Algorithms + Data Structures = Programs.* Englewood Cliffs, N.J., Prentice-Hall.