1981

Problems with COBOL–Some Empirical Evidence

D. M. Volpano

Herbert E. Dunsmore
Purdue University, dunsmore@cs.purdue.edu

Report Number:
81-371

Volpano, D. M. and Dunsmore, Herbert E., "Problems with COBOL–Some Empirical Evidence" (1981). *Department of Computer Science Technical Reports.* Paper 300.
https://docs.lib.purdue.edu/cstech/300

Problems With COBOL - Some Empirical Evidence

CSD TR-371
August 1, 1981

D.M. Volpano
H.E. Dunsmore

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

This study investigated programming activity in COBOL. Attempts were made to identify problem areas so that improvements can be made in COBOL compilers and in the manner in which COBOL is taught. Identification of problem areas was achieved through examining program changes made by student programmers during the development of four different COBOL programs. The data, which was collected from a COBOL course at Purdue University, consisted of all versions of all programs submitted for compilation by each student. Thus, the data represented a complete history of each subject's program development process beginning with the initial version compiled and ending with the final version submitted for grading. All program changes made between two successive versions were classified into four categories: COBOL-related, algorithmic, cosmetic and report-generation-related. This classification scheme indicates that a significant number of changes are related to report generation which suggests a need for support in this area. Secondly, all COBOL-related changes were delineated into 104 error categories. This delineation suggests that there are several problem areas in COBOL. Finally, the four categories of program changes were observed with respect to various points in the program development process. Most COBOL-related changes occur before the midpoint of the program development process whereas most cosmetic changes occur late in the process.

Keywords and Phrases: error-proneness, COBOL, programming languages, language features

1. Introduction

There is no doubt that COBOL is an important programming language. As part of the early triumvirate (with FORTRAN and ALGOL) COBOL is still important in business school programs and is the most widespread and intensively-used language in application programming [Phil73,Lemo79]. In industry, it has weathered the storm of PL/1 and even seems to be holding on in a world rapidly filling with PASCAL-trained programmers. There are those who believe that ADA will ultimately make COBOL obsolete but the slow pace of ADA's introduction suggests that if such occurs, it will be far in the future. Thus, for the present it appears that "real-world" programmers will continue to construct "real-world" programs in COBOL.

Despite COBOL's widespread use, it suffers from "human engineering problems" [Nels72]. The language has some features that are difficult to use safely (e.g. the CORRESPONDING option). Although COBOL has been touted as "easily-readable", few have ever claimed that it is "easily-writable". Furthermore, COBOL has received little academic attention [Samm78]. Little research has been done to attempt to identify problems with this language.

At Purdue University, we have been conducting a research project investigating COBOL. We are interested in those features of the language that may be troublesome for programmers. Our goal has been to identify such features so that (1) they might be emphasized when teaching COBOL, (2) existing compilers might be altered to provide better diagnostics, and (3) ultimately some language features might be changed to make them more usable. In the following sections of this paper, we describe a previous study of COBOL, report the methodology we employed, and discuss our results.

2. Previous Research

There have been attempts to investigate those constructs in ALGOL, BASIC, COBOL, FORTRAN and PL/1 which are difficult to use. Youngs [Youn74] analyzed 69 programs written in these languages and delineated errors into 8 functionally-defined categories : allocation, assignment, iteration, I/O formatting, other I/O, parameter/subscript list, conditionals and vertical delimiter. He found that these categories accounted for approximately 83 percent of all errors committed.

Youngs' study suggests that COBOL suffers in terms of allocation for two reasons. The allocation of space (for identifiers, tables, etc.) in COBOL is complex. Consider the following declarations for a 5 X 10 array in both FORTRAN and COBOL:

FORTRAN

```
DIMENSION ITEMS (5,10)
```

COBOL

```
01  ITEMS-TABLE.
    05  ITEMS-1 OCCURS 5 TIMES.
        10  ITEMS-2     PICTURE 999 OCCURS 10 TIMES.
```

Note that although the syntax used in COBOL to allocate space for tables is relatively complex, it provides a greater degree of flexibility. For example, one can access an entire "row" of the table declared above in COBOL using ITEMS-1 (*I*) for *I*=1,2,...,5. Secondly, COBOL suffers in terms of allocation because there is a lack of complete implicit and default specifications. For example, in the above FORTRAN declaration, ITEMS is implicitly declared to be an array of type integer. COBOL does not provide such an implicit type specification.

Another study conducted by Litecky and Davis studied errors and error-proneness in COBOL [Lite76]. "Error-proneness" is defined as the error frequency for a particular language element divided by the number of usages of that element. Errors from 1,400 runs from 73 students in a beginning COBOL course were classified according to a scheme established from a pilot study. The hierarchical classification scheme distinguished 132 types of errors. The highest level consisted of 32 major error classes such as hyphenation and punctuation. A relatively high frequency was found for many different types of COBOL errors. For example, a missing period and a misspelled structural word accounted for 6.6 and 2.6 percent of all COBOL errors respectively. However, only four error types were declared to be error-prone:

[1] Period added after the file name specified in a file description (FD). For example

```
FD  INPUT-FILE-NAME.
    LABEL RECORDS ARE STANDARD.
```

The period inserted after "INPUT-FILE-NAME" is syntactically incorrect.

[2] The use of commas as word delimiters. The following is an example of a comma used to delimit the identifiers B and C

```
ADD A TO B,C
```

The proper delimiter is a space rather than a comma.

[3] A missing period after a record name at a level 01. For example

```
FD  INPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  INPUT-RECORD
    05  SOME-FIELD      PIC 99.
```

COBOL syntax requires a period after the group level item "INPUT-RECORD".

[4] Operand(s) of an arithmetic statement are not computational in nature. For example, the arithmetic statement ADD A TO B is invalid in the context

```
05  A          PIC 999.
05  B          PIC ZZ9.
```

since B is alphanumeric.

Litecky and Davis also studied the content of specific high-frequency errors and the accuracy of compiler-generated error diagnostics. They found that 80 percent of the spelling errors in COBOL could be classified into only 4 error classes and therefore could be corrected by existing algorithms. The 4 error classes are

[1] One letter wrong

[2] One letter missing

[3] An extra character inserted

[4] Two adjacent characters transposed

The diagnosis of COBOL errors by the compiler (Control Data Corporation COBOL compiler for the 6600) was compared with the diagnosis of a "conversant" human judge. The major finding was that less than one in five errors were accurately diagnosed by the compiler.

The idea behind their research is good but we believe that their study has three major shortcomings:

[1] The COBOL errors identified are very low-level. For example, errors such as a missing hyphen in a FILE-CONTROL clause are very elementary relative to those errors that will cause problems for experienced programmers using advanced features. Thus, COBOL errors which most likely occur at the professional level have not been adequately identified.

[2] The behavior of high-frequency errors and error-proneness has not been observed over time. Thus, some error types that are claimed to be error-prone may not be a problem as programmers become more experienced in COBOL. For example, in our study we found that the frequency for the error type "period added after FD filename" decreases quickly.

[3] Only one compiler was considered in the study of error diagnosis accuracy. Therefore any results pertaining to error diagnosis accuracy cannot be generalized.

3. Procedure

Our research attempts to identify problem areas in COBOL by studying program changes made by programmers who developed several different programs. A program change is defined as a textual change between successive versions of a program [Duns80]. Each of the following textual changes to a program represents one program change:

One or more changes to a single statement. Even multiple character changes to a statement represent mental activity with only a single abstract instruction.

One or more statements inserted between existing statements. The contiguous group of statements inserted probably corresponds to the concrete statements that represent a single abstract instruction.

A change to a single statement followed by the insertion of new statements.

The following textual changes to a program are not counted as program changes:

The deletion of one or more statements. Deleted statements must usually be replaced by other statements elsewhere. The inserted statements are counted. Counting deletions as well would give double weight to such a change.

The insertion of standard output statements. These are occasionally inserted in a "wholesale" fashion during debugging.

Examining program changes for several different programs developed by the same set of subjects enabled us to observe the frequency of various error types with respect to time.

Our research involved three major areas:

[1] All program changes were classified as *algorithmic*, *COBOL-related*, *cosmetic*, or *report-generation-related*. *Algorithmic* program changes are those needed to correctly implement an algorithm. For example, changing

```
IF KEY = DEPT-NO
```

to

```
IF KEY = DEPT-NO AND NOT = PREV-DEPT-NO
```

is considered an algorithmic change since the original statement is syntactically correct. The change is made to correctly implement the chosen algorithm. *COBOL-related* changes are those necessary due to restrictions imposed by COBOL. For example, a missing hyphen in a keyword (e.g. LINE-COUNTER) necessitates a COBOL-related change. *Cosmetic* changes include the insertion of blank lines and comments as well as reformatting without alteration of existing statements. *Report-generation-related* changes include those changes necessary to generate a report. Such changes often involve maintaining page numbers, manipulating carriage control and determining page breaks (The Report Writer feature was not used in any of the programming assignments). Some program changes can be placed into two categories. For example, changing

```
IF LINE-COUNTER > 55
```

to

```
IF LINE-KOUNTER > 60
```

is considered to be both a COBOL-related and report-generation-related change; COBOL-related because LINE-COUNTER is a COBOL reserved word and report-related because 60 lines are now desired rather than 55. As an example of the intersection between the categories of algorithmic and COBOL-related, consider changing

```
IF EMP-NO <> PREV-EMP-NO
```

to

```
IF EMP-NO NOT = PREV-EMP-NO AND OLD-EMP
```

This change is considered algorithmic because an additional condition must be satisfied and is considered COBOL-related because "<>" cannot be used to denote inequality in COBOL.

[2] All COBOL-related program changes were further delineated into 104 error categories such as editing, literals, punctuation etc..

[3] Each of the four categories of program changes were examined with respect to when they occur in the program development process.

The data for our research was obtained from students in an upper-level COBOL course at Purdue University in the summer of 1980. The students, who had some experience programming in FORTRAN or PASCAL, were required to write five programs (COBOL1, COBOL2, ..., COBOL5) as part of the course requirements. The first program, COBOL1, was disregarded for our purposes because it did not demand significant programming effort and represented most students' initial experience with COBOL. The second program, COBOL2, involved writing a file in readable form. The last three programs, which were approximately 700-800 lines of code each, involved master file updating. COBOL5 required changing COBOL4 to include random access. Some COBOL features that would most likely appear at the professional level, namely sorting and random access, were employed in COBOL4 and COBOL5 only. Thus, we could not observe how the frequency of program changes in these categories behave over time.

All versions of a particular program submitted for compilation were captured for each programmer. The average number of versions submitted per programmer ranged from 12 for COBOL2 to 53 for COBOL4. Instead of examining all versions from approximately 40 programmers for each programming assignment, a random sample of 10 programmers was used. A sample size of 10 seemed to be appropriate since 2 random samples, of 10 programmers each, yielded similar results for COBOL2. For each of these sample groups, *Table 1* shows the frequency of changes for each category and the percentage that frequency is of the total number of changes.

	COBOL		Algorithmic		Cosmetic		Report Generation	
	f	%	f	%	f	%	f	%
Group 1	176	23.9	189	25.9	193	20.7	371	50.4
Group 2	151	20.8	184	25.4	102	12.3	407	56.2

Table 1

To examine program changes between two successive versions, a system utility called "SRCCOM" was used. SRCCOM provided a file of all textual changes between two versions. This file was then examined manually.

4. Results

Our results correspond to the three major areas involved in our research. For each programming assignment, *Table 2* shows the frequency of changes for each category and the percentage that frequency is of the total number of changes for that assignment. The sum of the percentages is greater than 100% because of the overlap discussed earlier. Note that algorithmic changes account for only 25 percent of the changes for COBOL2 but account for over 60 percent of all changes for the last three assignments. Half of all changes on COBOL2 are report-generation-related but for COBOL3, 4, and 5 these remain relatively stable constituting approximately 25 percent of all changes. Finally, notice that those changes necessitated by problems with COBOL remain relatively stable at about 20 percent. That is, one of every five changes is due, at least in part, to problems with the programming language.

	COBOL		Algorithmic		Cosmetic		Report Generation	
	f	%	f	%	f	%	f	%
COBOL2	176	23.9	189	25.9	193	20.7	371	50.4
COBOL3	197	21.6	611	67.0	177	16.3	203	22.3
COBOL4	199	19.1	701	67.3	211	16.8	276	26.5
COBOL5	36	21.4	101	60.1	108	39.1	44	26.1

Table2

Appendix 1 represents the delineation of COBOL-related program changes into the 104 error categories. For each error category, *Appendix 1* shows the frequency of changes made due to this type of error for each programming assignment. A blank entry in *Appendix 1* indicates that the COBOL feature for the error category in question was not employed in this programming assignment. For example, only COBOL5 required random access and therefore there are blank entries for this category for COBOL2, COBOL3 and COBOL4.

For programming assignments COBOL2-COBOL5, *Figure 1* shows that COBOL-related changes are typically made early in the program development process whereas *Figure 2* shows that cosmetic changes are more frequent at the end. *Figures 3* and *4* show that algorithmic and report-generation- related changes occur throughout the development process.

5. Discussion

As indicated in *Table 2*, there is a significant number of report-generation-related changes for each programming assignment. This suggests that programmers could most likely use some support in generating reports. One type of support already being used (not in this study) is the Report Writer feature. A study has shown that programmers find Report Writer makes the maintenance and generation of reports much easier [Aude81]. However, our research does not suggest that Report Writer is a panacea, primarily because some changes involved features that exist even in Report Writer. For example, changes which involved editing were considered report-generation-related but clearly such changes may be necessary even if Report Writer were used.

Our research suggests that the following error categories appear to be problem areas in COBOL. However, it does not suggest that these categories are error-prone. Recall that error-proneness is a function of the total number of usages of a particular language element. Since we did not attempt to determine the total number of usages for each of the language features in question, we cannot make any conclusions pertaining to error-proneness. The frequency of program changes for some of these categories remains relatively stable over time and therefore these categories appear to be problem areas. Other categories show potential for being problem areas due to the relatively high frequency of changes observed.

[1] Data-name qualification.

The program changes that we categorized as "data-name qualification" involved qualifying non-unique data names. There were considerably more instances where qualification was omitted entirely than there were instances where it was inadequately specified. For example, in the context

```
01  A.
   05  B.
      10  C     PIC 99.

01  D.
   05  B.
      10  C     PIC XXX.
```

frequently the data name C was not qualified when referenced in the procedure division. Proper qualification of C in this context is C OF D or (C OF B OF D) or C OF A or (C OF B OF A). The function of non-unique data names in COBOL is twofold; they provide increased flexibility and are necessary for the proper use of the CORRESPONDING option. Despite increased flexibility, non-unique data names require qualification, and qualification actually makes programming in COBOL more cumbersome. For example, consider the arithmetic statement

```
MULTIPLY QTY-ON-HAND OF INPUT-QTY
BY UNIT-PRICE OF PARTS-RECORD
GIVING TOTAL-COST OF OUTPUT-RECORD.
```

Data-name qualification appears only to complicate programming and make such arithmetic expressions less readable. Since COBOL is an inherently verbose language, it would most likely not suffer if all data names were required to be unique.

[2] CORRESPONDING option.

This feature may be used to reference all fields with common data names within two different groups [Shel77]. Most program changes made due to the CORRESPONDING option were attempts to reference the fields intended within two different groups. For example, in the context

```
01  A.
    05  C.
        15  D     PIC X.

01  B.
    05  C.
        10  E.
            15  D     PIC X.
```

we observed many programmers using a statement such as

```
MOVE CORRESPONDING A TO B
```

to move the contents of D in group A to D in group B. However, the intended move will not occur in this context since D in group B is at a different level than D in group A. The CORRESPONDING option has pitfalls that have caused experienced programmers to minimize its use. The main problem is that it tends to create trouble when a program is changed, as virtually all programs are if they are used for any length of time. One portion of a program that generally changes is the format of records. Experience has shown that record format changes very frequently cause the CORRESPONDING verb to give undesired results [McCr76]. The effort of using unique data names and explicitly referencing elementary data items has the advantage of providing easier maintenance of the program and reduced chance of error. Thus, it appears that COBOL would not suffer without the CORRESPONDING option.

[3] Edited numeric data items as operands in arithmetic expressions.

The restriction that edited numeric data cannot be used in arithmetic statements often causes a programmer to declare another data name to be computational in nature. For example, in the context

```
05  LINE-KOUNTER     PIC ZZ9
```

the statement ADD 1 TO LINE-KOUNTER is invalid since LINE-KOUNTER is alphanumeric. COBOL compilers could be written to generate code that would coerce edited numeric data in much the same way as integers are coerced in real expressions in FORTRAN. However, the introduction of coercions into a programming language should be done with considerable discretion [Tenn81]. For example, consider the declaration

```
05  FIELD     PIC X.
```

The bit configuration of FIELD, which occupies one byte, can represent a digit or some other character such as a letter. Since edited numeric data items are a subset of the set of all alphanumeric data items, it would be possible to extend coercion to the set of all alphanumeric data items. Such an extension would allow FIELD to occur as an operand in an arithmetic expression. However since FIELD can represent a letter, coercion would allow computation of the arithmetic expression to continue and possibly produce bizarre results. Programmers normally do not welcome error messages but a message that helps in locating a bug is far more useful than meaningless output.

[4] Literal continuation.

The program changes related to literal continuation involved correcting a misplaced single quote or providing a hyphen in column seven. The frequency of changes made due to invalid literal continuation decreases rapidly after the second programming assignment, COBOL2 (see *Appendix 1*). This rapid decline is due to the abandonment of the technique used to continue literals. Programmers avoided this technique by adopting other means for declaring lengthy literals. For example, some programmers placed the entire literal on a new line whereas others partitioned the literal into smaller segments.

[5] IF-ELSE pairing convention.

Perhaps the most famous example of ambiguity in a programming language is the dangling ELSE. Consider the conditional

```
IF $c_1$
   $s_1$
   IF $c_2$
      $s_2$
ELSE
   $s_3$.
```

example 1

It is not clear to which IF the ELSE corresponds. Without changing the formal definition of the syntax of COBOL, the ambiguity can be resolved in one of two ways. The first approach involves introducing the keywords BEGIN and END (see *example 3*). The second approach, which is used in COBOL, is to adopt a convention. The one used in COBOL is that in a nested IF statement, the first ELSE clause corresponds to the innermost IF statement [Shel77]. Consider *example 1*. If s_3 is to be executed when c_1 is false then "ELSE NEXT SENTENCE" must be inserted before the existing ELSE since the ELSE which currently exists corresponds to the innermost IF (see *example 6*).

[6] Dependency upon the period to terminate a conditional.

Since COBOL is very sentence-oriented, the placement of a period after a statement is natural. However, the use of periods after statements within conditionals will yield undesirable results. As a result of the dependency upon the period in an IF statement, programmers often spend much time debugging programs only to discover the existence of an extraneous period in an IF statement. The following conditional illustrates the problem of period dependency.

```
IF c1
    READ file-name
      AT END s3
    s1
    s2.
```

example 2

The programmer is forced to put a period at the end of the imperative clause s_3 so that s_1 and s_2 are not executed upon an end-of-file condition only. However, placing a period after s_3 causes s_1 and s_2 to be executed independently of c_1 because the period terminates both the AT END clause and the conditional. Clearly, an "ENDIF" or perhaps an "ENDAT" construct would eliminate the dependency upon the period. However, until such a construct is added to the language, COBOL instructors should emphasize such potential pitfalls.

There have been attempts to simplify COBOL programming by making COBOL extensible; i.e., allowing the syntax and semantics of COBOL to be changed. One of the earliest and most commonly proposed schemes for language extension is the macro definition [Grie71,Aho 72]. Already in use, are two macro preprocessors MetaCOBOL and COBRA which enhance COBOL [ADR.76,Hami73]. The processor need not precede compilation. Triance et al. built a macro facility into a COBOL compiler [Tria80]. This compiler is believed to be the first compiler with a builtin macro facility capable of recognizing macro calls with arguments. An example of a macro call specified in a COBOL program is "CSR". This call initiates execution of a macro which simply replaces the call by "COMPUTATIONAL SYNCHRONIZED RIGHT". The COBOL macro facility could conceivably be extended to provide support in the area of nested conditionals. For example, a programmer could override the convention adopted for IF-ELSE pairing by using the keywords BEGIN and END. For example, assuming s_3 is to be executed if c_1 is false, *example 1* could be rewritten as

```
IF $c_1$
    BEGIN $s_1$
    IF $c_2$
        BEGIN $s_2$ END
    END
ELSE BEGIN $s_3$ END.
```

example 3

Suppose there were an "ENDIF construct, then assuming s_1 and s_2 are to be executed if c_1 is true and independently of the imperative clause, *example 2* could be rewritten as

```
IF $c_1$
    READ file-name
      AT END $s_3$.
    $s_1$.
    $s_2$.
ENDIF
```

example 4

To utilize a macro facility, a programmer could specify a macro call such as "CONDITIONAL". For example

```
CONDITIONAL
IF $c_1$
    BEGIN
    $s_1$
```

```
        IF $c_2$
            BEGIN $s_2$ END
        END
    ELSE BEGIN $s_3$ END
    ENDCONDITIONAL
```

example 5

The entire conditional specified between "CONDITIONAL" and "ENDCONDITIONAL" would be treated as a by-value parameter subject to interpretation by a particular macro. This macro would generate standard ANSI COBOL code for the conditional specified. For example, the code generated for *example 5* would be

```
IF $c_1$
    $s_1$
    IF $c_2$
        $s_2$
    ELSE
        NEXT SENTENCE
ELSE $s_3$.
```

example 6

6. Summary

Since COBOL is a widely used language, there is a need to identify its problem areas so that improvements can be made in COBOL compilers and in the manner in which COBOL is taught. Such improvements could yield a reduction in the number of errors committed by COBOL programmers.

Attempts have been made to identify error-inducing features in COBOL [Litc76,Youn74]. However, the error frequencies for certain COBOL features have not been observed with repect to time. Our research attempted to identify error-inducing features (problem areas) by observing the frequency of errors for various features over time. Thus the features we have identified as problem areas are likely to be error-inducing for experienced as well as novice COBOL programmers. Our study suggests there are at least six problem areas in COBOL:

[1] Data-name qualification

[2] CORRESPONDING option

[3] Edited numeric data items in arithmetic expressions

[4] Literal continuation

[5] IF-ELSE pairing convention

[6] Dependency upon the period to terminate a conditional

Furthermore, we have suggested approaches that may tend to eliminate some of these problem areas. For example, we feel that non-unique data names and the CORRESPONDING option could be eliminated. Edited numeric data items occuring in arithmetic expressions could be coerced. A macro facility could be used to alter the syntax of conditionals in COBOL so that errors related to conditionals can be reduced.

Undoubtedly additional problem areas exist in COBOL. For example, we could not observe the error frequency for features such as the COBOL sort facility and random access since these features were not used more than once by our subjects. Hence, there is a need for further research to observe the error frequency over time for more advanced features. Upon identifying those error-inducing features, additional improvements can be made with respect to COBOL compilers and the teaching of COBOL.

7. Acknowledgements

We wish to thank Andrew Wang for his effort in the time-consuming task of collecting the data for our study. We would also like to extend our appreciation to Bill Ward and Tom Putnam for aiding in the manipulation of data and lastly to the instructor of the COBOL course Steve Booth and his students for participating in the study. This research was supported by the U.S. Army, contract no. DAAG29-79-C-0173, Purdue University Computing Center and the Purdue University Department of Computer Science.

8. References

[ADR.76] ADR. "MetaCOBOL concepts and facilities", *Applied Data Res.*, Princeton, N.J., 1976.

[Aho 72] Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation and Compiling Volume 1 : Parsing Prentice Hall, 1972.

[Aude81] Aude, T.J., "The productivity and readability of COBOL's Report Writer., *Project Report, CS 590D*, Department of Computer Sciences, Purdue University, May 1981.

[Duns80] Dunsmore, H.E. and Gannon J.D., "Analysis of the effects of programming factors on programming effort", *The Journal of Systems and Software* . 1, pp. 141-153, 1980.

[Grie71] Gries, D., Compiler Construction for Digital Computers, John Wiley & Sons Inc, 1971.

[Hami73] Hamilton, J.G.A., Finlayson, E.D., and Heywood-Jones, A.H., "Computer aided program production", *Proc. Datafair, Nottingham, England*, pp. 191-196, 1973.

[Lemo79] Lemos, R.S., "An implementation of structured walk-throughs in teaching COBOL programming", *Comm ACM* 22, 6, pp. 335-340, June 1979.

[Lite76] Litecky, C.R. and Davis, G.B., "A study of errors, error-proneness and error diagnosis in COBOL", *Comm ACM* 19, 1, pp. 33-37, Jan 1976.

[McCr76] McCracken, D.D., A Simplified Guide To Structured COBOL Programming, John Wiley & Sons Inc. 1976.

[Nels72] Nelson, D.A., "COBOL : Some limitations on the implementor and the user", *CODASYL Symposium on COBOL Compiler Techniques*, Philadelphia, Pa., pp. 1-35, May 22, 1972.

[Phil73] Philipakkis, A.S., "Programming language usage", *Datamation*, 19, 10, pp. 109-114, Oct. 1973.

[Samm78] Sammet, J., "The early history of COBOL. ACM

SIGPLAN History of Programming Languages Conference, Los Angeles, June 1978, *SIGPLAN Notices* 13, 8 pp. 121-160, August 1978.

[Shel77] Shelly, G.B. and Cashman, T.J., Introduction to Computer Programming Structured COBOL, Anaheim Publishing Co., 1977.

[Tenn81] Tennent, R.D., Principles of Programming Languages, Prentice Hall International, 1981.

[Tria80] Triance, J.M. and Yow, J.F.S., "MCOBOL - a prototype macro facility for COBOL", *Comm ACM*, 23, 8 pp. 432-439, Aug. 1980.

[Youn74] Youngs, E.A. "Human errors in programming", *Int. J. Man-Machine Studies*, 6, pp. 361-376, 1974.

```
CS Dept.  VAX 11/780  UNIX   Thu Apr  1 12:37:44 1982      dmv     -   Dennis Volpano, MATH-437, 4




     d
     d
     d
     d
     d
 ddddd  mmm mm  v     v
d    d  m  m  m v     v
d    d  m  m  m v     v
d    d  m  m  m  v   v
d    d  m  m  m   v v
 ddddd  m  m  m    v
```

Appendix 1

	Program			
	COBOL2	COBOL3	COBOL4	COBOL5
Error Categories	f	f	f	f
1. Structural Keywords				
A. Misspelling				
1. ENVIRONMENT	1	0	2	0
2. DATA DIVISION	2	1	0	0
B. Missing Keywords				
1. Data Division				
a. FILE SECTION	0	2	1	0
b. WORKING-STORAGE SECTION	1	0	0	0
c. PICTURE	0	3	2	0
2. Procedure Division				
a. STOP RUN	0	1	0	0
b. OPEN	0	1	0	0
c. CLOSE	1	1	2	1
d. INPUT/OUTPUT of OPEN	0	0	3	0
2. Sentence Structure				
A. Invalid PERFORM	0	0	3	0
B. Data-name qualification				
1. Omitted	0	5	37	0
2. Insufficient	0	0	3	0
C. Invalid assignment	0	2	0	0
D. Misspelling				
1. ASCENDING			1	0
2. CORRESPONDING			2	0
3. Editing				
A. Zero suppression				
1. Truncation of higher order digits	4	0	1	0
B. Editing symbols				
1. ./V	9	1	4	0
2. -/S	5	0	0	0
3. S used for zero suppression	1	0	0	0
4. B used instead of SPACES	1	0	0	0
5. Editing symbol in PICTURE not intended to edit	1	3	0	0
C. Use of edited item in				

arithmetic	7	2	5	0
4. CORRESPONDING verb				
A. Improper use			28	0
5. Format				
A. Margins				
1. Left of column 8	0	1	4	1
2. Right of column 8	0	9	12	1
3. Left of column 12	4	5	7	0
4. Right of column 72	3	4	1	0
6. Reserved words used as identifiers				
A. PAGE	1	0	0	0
B. PAGE-COUNTER	4	0	0	0
C. LINE-COUNTER	0	1	0	0
7. Data description				
A. Format				
1. Missing keyword ALL in VALUE clause	1	0	0	0
B. Class				
1. Alphanumeric/Numeric	2	1	3	0
C. Spacing				
1. Space between type and length (e.g. PIC X (18))	0	5	0	1
D. Level number missing	0	0	1	0
8. Punctuation				
A. Period added				
1. Within CLOSE statement	1	0	0	0
2. After FD file-name	2	0	0	0
3. After VALUE keyword	1	0	0	0
4. Within OPEN statement	1	0	1	0
5. Before end of file description	0	3	0	0
B. Period missing after				
1. SOURCE-COMPUTER	1	0	0	0
2. OBJECT-COMPUTER	1	0	0	0
3. Group level item	18	7	3	1
4. PICTURE clause	4	3	3	2
5. VALUE clause	4	2	2	0
6. Program name	0	1	1	1
7. FILE SECTION	0	1	0	1
8. Paragraph name	0	0	4	0

9. Hyphenation

A. Missing in				
1. SPECIAL-NAMES	1	0	0	0
2. SOURCE-COMPUTER	1	0	0	0
3. OBJECT-COMPUTER	1	0	0	0
4. HIGH-VALUES	0	2	0	0
5. FILE-LIMIT				13
B. Added in				
1. WORKING-STORAGE-SECTION	0	1	0	0
10. Literals				
A. Literal continuation				
1. Misplaced hyphen or single quote	11	0	1	0
B. Alphanumeric/Numeric (e.g. PIC 99 VALUE '20')	5	3	0	0
C. Alphanumeric literal				
1. Missing quotes	8	4	0	0
2. Length exceeds size of PICTURE	0	1	2	0
3. Invalid delimiter	0	4	0	0
11. Invalid use of figurative constants				
A. SPACES	20	0	0	0
12. Conditionals				
A. Invalid compound				
1. AND/OR missing	1	0	0	0
2. Not parenthesized properly	12	0	0	1
3. Improper use of AND/OR	0	6	0	0
B. Invalid relational operator				
1. NOT=/IS UNEQUAL/<>	5	0	0	0
2. Space not preceding/following relational operator	2	1	0	0
3. IF A < THAN B	0	6	0	0
C. Missing period	0	6	2	2
D. Unmatched ELSE	3	2	5	1
E. Period placed too early				
1. Nested conditional	8	8	6	2
2. Not nested	0	4	2	1
F. Abbreviations				
1. Subject and relation omitted in compound conditional which involves a class test (e.g. IF A = B AND NUMERIC)	0	0	5	0
13. Write statement				

A. Write WORKING-STORAGE record	10	3	0	0
B. Write statements with *and* without ADVANCING option	3	0	4	0
14. Read statement				
A. AT END clause omitted	1	0	1	0
B. Conditional within imperative clause	3	4	0	0
C. READ is not last statement within conditional	0	43	0	0
D. READ file-name-1 INTO file-name-2	0	4	0	0
E. READ file-name TO record-name	0	4	0	0
15. Level 88 items				
A. PICTURE clause at level 88	0	0	2	0
B. Quantity MOVEd to a level 88 item	0	3	0	0
C. Level 88 item MOVEd	0	0	1	0
D. Data name with PICTURE clause used as switch	0	0	1	0
16. Redefinition				
A. At a level other than 01 did not have the same number of bytes as the item being redefined	0	2	1	0
17. Tables				
A. Subscripting				
1. No space separating data name and left parentheses of subscript		11	5	0
2. Subscript missing		0	5	4
3. Subscripted data name used as subscript		1	0	0
4. Data name without OCCURS clause is subscripted		0	1	0
B. OCCURS clause				
1. At a level 01		2	1	0
2. PIC X(40) OCCURS 40 TIMES/ PIC X OCCURS 40 TIMES		3	1	0
C. Indexing				
1. Use of an index other than the index defined for that table		0	6	0
D. SEARCH verb				
1. SEARCH the incorrect data name		1	1	0
E. Level structure				

1. Improper level number	0	1	0
18. SORT verb			
A. FD/SD		1	0
B. READ/RETURN		2	0
C. WRITE/RELEASE		1	0
D. INPUT/OUTPUT procedure is not a section		1	0
E. PERFORM paragraph-name SECTION		2	0
F. INPUT/OUTPUT PROCEDURE IS paragraph-name SECTION		1	0
G. Invalid sort key		2	0
H. SELECT clause for sort file missing		3	1
19. Random access			
A. Environment division			
1. Invalid SELECT clause			2

FIG. 3

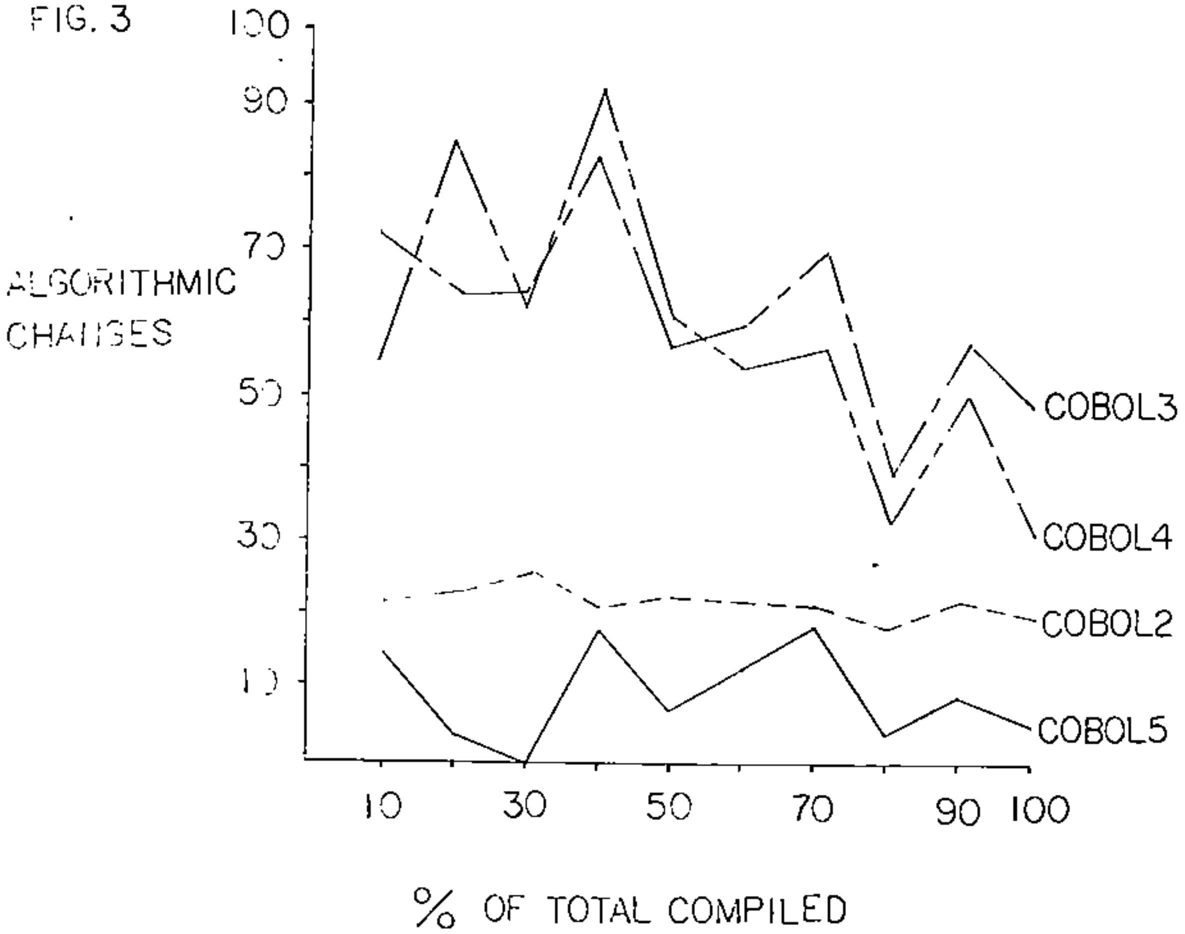

100
90
70
50
30
10
ALGORITHMIC CHANGES
COBOL3
COBOL4
COBOL2
COBOL5
10
30
50
70
90
100
% OF TOTAL COMPILED

FIG. 1

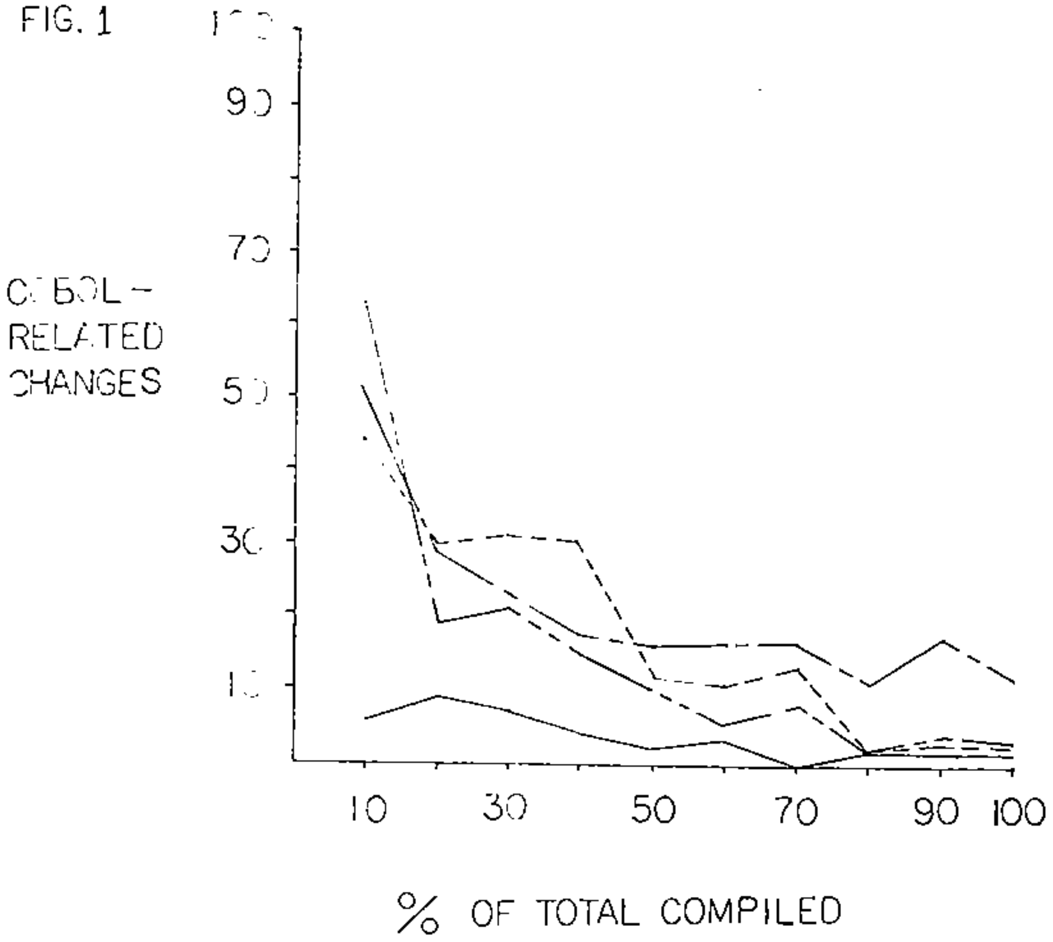

100
90
70
50
30
10
COBOL - RELATED CHANGES
10
30
50
70
90
100
% OF TOTAL COMPILED

FIG. 2

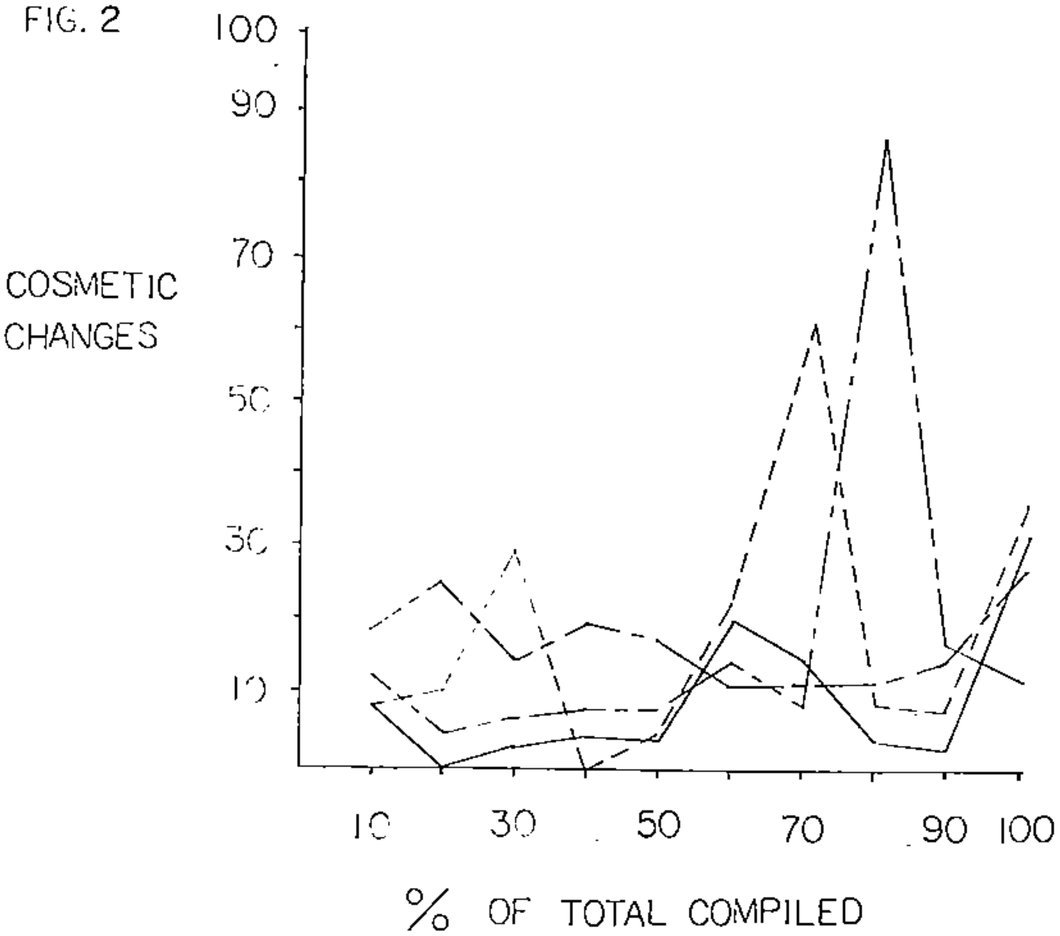
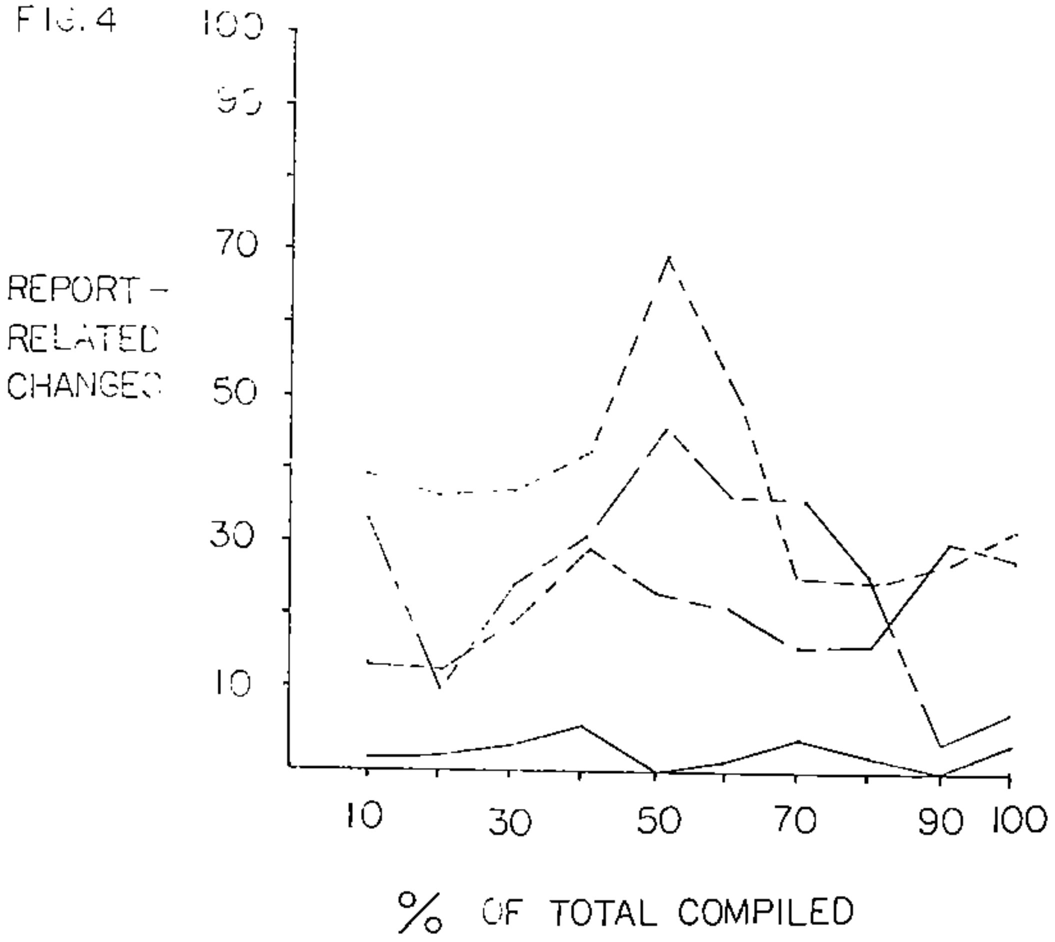

100
90
70
50
30
10
COSMETIC CHANGES
10
30
50
70
90
100
% OF TOTAL COMPILED

FIG. 4


100
90
70
50
30
10
REPORT-RELATED CHANGES
10
30
50
70
90
100
% OF TOTAL COMPILED