A STUDY OF ERRORS CAUSED BY TRANSCRIPTION MISTAKES IN FORTRAN PROGRAMS

by

L. D. Fosdick
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-146-79

August, 1979

INTERIM TECHNICAL REPORT

U.S. ARMY RESEARCH OFFICE

CONTRACT NO. DAAG29-78-G-0046

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

We acknowledge U.S. Army Research support under contract no. DAAG29-78-G-0046 and National Science Foundation support under grant no. MCS77-02194

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CU-CS-146-79

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): "A Study of Errors Caused by Transcription Mistakes in Fortran Programs"

5. TYPE OF REPORT & PERIOD COVERED

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): Lloyd D. Fosdick

8. CONTRACT OR GRANT NUMBER(s): DAAG29-78-G-0046, MCS77-02194 (NFS)

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Dept. of Compter Science, University of Colorado at Boulder, Boulder, Colorado 80309

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709

12. REPORT DATE: August 1979

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: NA

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

software reliability, keypunch errors, error detection

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Transcription mistakes which are not caught in proof-reading must be caught by observation of phenomena, such as syntax errors or wrong results, caused by them. Here we explore the nature and frequency of simple phenomena caused by typing mistakes such as striking the wrong key on a keyboard. This is done mainly with a simulation of the mistakes but some analytic work on this problem is also described. Finally, the efficacy of compilers in detecting the phenomena caused by typing mistakes is described.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

A STUDY OF ERRORS CAUSED BY TRANSCRIPTION MISTAKES IN FORTRAN PROGRAMS[†]

LLOYD D. FOSDICK[††]

Introduction

Transcription mistakes are a common kind of mistake made in the construction of programs. Often they occur when a program is transcribed from a handwritten form into a machine readable form, but they also occur when a program is transcribed from the mind of the author onto paper, or from a flow diagram into a sequence of statements, and indeed whenever transcription is performed. It is clear that these mistakes are inevitable: no matter how much care is taken in the preparation of a program, no matter how rigorously good principles of design are followed, and no matter how much effort is invested in proving a program, the chance of program errors caused by transcription mistakes cannot be reduced to zero because human systems are not perfect. Indeed there is a kind of uncertainty principle operating in this domain because the act of verifying a program itself involves transcription and is therefore vulnerable to these mistakes.

Depending on the care taken, some transcription mistakes will be discovered by proof-reading and those which remain will be discovered, if at all, by the phenomena they cause. When a transcription mistake causes a syntax error it will be discovered easily; or, when a transcription mistake causes an unusual construction to appear, as when

---

[†]Supported in part by U.S. Army Grant DAAG29-78-G-0046 and NSF Grant MCS 77-02194

[††]Dept. of Computer Science, Univ. of Colorado, Boulder, Colo. 80309

the FORTRAN statement

$$X = X + 1.0$$

is erroneously transcribed as

$$Y = X + 1.0$$

and Y is not a program variable, it too can be found easily. But when π is written to fifteen significant figures as

3.1415 93653 58979,

the fact that the seventh digit should be 2 instead of 3 will not be discovered with comparable ease. Thus we arrive at the question which occupies us here: What is the nature of the errors caused by transcription mistakes and what portion of them can be detected easily, that is at a cost comparable to the cost of compilation.

The simplest and perhaps the most common kind of transcription mistake, is made with individual characters. The substitution of one character for another is an example. Another kind is the confusion of identifiers, where one is substituted for another. In a sense this kind of mistake is more complex since it takes place at the word level rather than the character level and, probably more importantly, it involves memory. Still another kind is the omission of expressions, statements, or even sequences of statements. Such mistakes are easily caused by a lapse in attention and it is not uncommon that the omitted text is preceded by a segment similar or identical to the end of the omitted text. From the point of view of the mental processes involved this kind of mistake seems almost as simple as the single character mistake. However, our interests here do not require that we know why a mistake was made or whether one is more complex than another. Our interest is in the effects of a mistake.

In this paper the focus is on single character mistakes in FORTRAN programs. The effects of several kinds of single character mistakes on programs with different characteristics are considered and we look briefly at the ability of some widely used compilers to detect the errors caused by these mistakes. A Monte Carlo scheme is used to generate an ensemble of programs containing errors from simulated transcription mistakes. These errors are then analyzed and classified according to the ease with which they may be detected. The difficulty of the problem addressed here almost precludes deriving useful results by formal analysis. However, in the next section a simple analysis of this problem to predict the frequency of syntax errors is described and, as we shall see, it yields results which are in good agreement with those obtained from Monte Carlo sampling.

The idea of inserting simulated mistakes in programs has been discussed by others. Weinberg and Gresset [1] used it to study the error detecting capability of a FORTRAN compiler. It has been advocated by Gilb [2] as a technique for measuring the number of undetected errors in a program - adopting the ideas used by biologists for measuring fish populations, etc. Recently Lipton and Sayward [3] have suggested it as a mechanism for guiding the selection of test data. The work reported here, while bearing some relation to this other work, is different in its objectives from the work of Gilb and that of Lipton et.al., and is wider in scope than the work of Weinberg and Gresset.

Prediction of Syntax Errors by Analysis.

Since short assignment statements, appear to be the most common kind of statement appearing in FORTRAN programs [4] we direct our

attention at them. Let α stand for any letter of the alphabet, and ω for any letter or any of the of the ten decimal digits. All legal three-character assignment statements will have the form

$$\alpha = \omega.$$

Now count the number of ways in which exactly one of these characters can be replaced by another FORTRAN character in such a way that a syntactically correct statement results. For all such statements but one this number is 60: for the one exception, namely the statement E = D, this number is 61 because of the possibility E = D ⇒ END. There are altogether 47 characters in the FORTRAN character set [5] hence, ignoring the exception, the probability that substitution of exactly one of the three characters by another will yield a syntactically correct statement is 60/138 = 0.43.

We can extend this straightforward analysis to longer statements but the number of cases that need to be considered grows rapidly and the computation becomes very tedious. A brief look at four-character assignment statements is sufficient to illustrate this. Let σ stand for + or -, δ for any decimal digit, and α and ω as before. There are nine forms to be considered: $\alpha = \alpha\alpha$, $\alpha = \alpha\delta$, $\alpha = \delta\delta$, $\alpha = \sigma\alpha$, $\alpha = \sigma\delta$, $\alpha = .\delta$, $\alpha = \delta.$, $\alpha\alpha = \omega$, $\alpha\delta = \omega$. With each form we assign a weight, w, which is the number of instances of that form; for example, $w(\alpha = \alpha\alpha) = 26^3$, $w(\alpha = \alpha\delta) = 26^2 \times 10$, and so forth. We distinguish between the forms $\alpha = \alpha\alpha$ and $\alpha = \alpha\delta$ and do not lump them together as $\alpha = \alpha\omega$ because the third character can be changed to a digit or a decimal point in the second form yielding a syntactically correct statement (viz. A = A9 ⇒ A = 99, or A = A9 = A ⇒ .9) but this is not true for the form $\alpha = \alpha\alpha$. Similar considerations force distinction

of the nine forms listed above. For each of these cases we compute the probability, p, that a single character substitution will yield a syntactically correct statement just as for the three-character case; for example, $p(\alpha = \alpha\alpha) = 0.48$, $p(\alpha = \alpha\delta) = 0.54$. Finally, we compute the average probability $\bar{p}$ that a single character substitution will yield a syntactically correct statement: this is given by the usual formula

$$\bar{p} = \frac{\sum_i w(i)\, p(i)}{\sum_i w(i)},$$

where the sums extend over the nine cases. The result is $\bar{p} = 0.51$. When this analysis is extended to five-character assignment statements 57 forms are distinguished and similarly analyzed: for this class of statements the average probability that a single character substitution will yield a syntactically correct statement is $\bar{p} = 0.56$. This analysis has not been extended to longer assignment statements because the number of forms which need to be distinguished makes the problem almost intractable.

On the basis of this approach we can estimate that a single character substitution in a FORTRAN program has a slightly better than 50% chance of yielding a program that is syntactically correct. This estimate is crude for a number of reasons which are evident from the approach we have taken. However, we shall see that it agrees rather well with the random sampling or Monte Carlo approach described below.

Monte Carlo Experiments to Simulate Transcription Mistakes.

Four common transcription mistakes made in typing are simulated in these experiments: substitution - the substitution of one character for another; deletion - the omission of a character; insertion - the insertion of a character; transposition - the interchange of adjacent

characters. All of these, except transposition, are single character mistakes. Two of them, substitution and insertion, require the introduction of a new character into the text and so the question of how this new character is to be selected arises. In simulating substitution mistakes we randomly selected a character from among the correct character's nearest-neighbors on the keyboard of an IBM 026 keypunch. This rule was used to govern the selection because evidence from experiments with typists shows that a nearest-neighbor is the most likely character to be erroneously substituted [6]. However, in order to explore the effect of another selection rule, a series of experiments were made in which every character in the FORTRAN character set was made an equally likely candidate for substitution. As will be seen, use of this alternate rule had a noticeable effect on the results. Another obvious choice, but not one considered here, is the character on the same key but in the alternate shift mode - simulating failure to shift from alphabetic to numeric or vice versa. For insertion mistakes the alternate selection rule, all characters equally likely, was the only rule used.

The character position in the program text where the mistake is simulated was selected at random, giving each position equal probability of selection, ignoring COMMENT statements and blank positions. When the position was selected one instance of each kind of mistake was simulated. This selection process was repeated fifty times, so for each program text fifty samples of it were created with one instance of a particular kind of mistake - thus two hundred and fifty samples of a particular text altogether: fifty of substitution with nearest-neighbor character substituted, fifty of substitution with any

character substituted, fifty of deletion, fifty of insertion with any character substituted, and fifty of transposition. There is a correlation among samples arising from the fact that, for a given position selection, each of the five kinds of mistake appeared at the same place. This correlation permits a better comparison of the effects of the different kinds of mistake.

The particular mistakes chosen for consideration here are no doubt familiar to the reader who may draw on personal experience to decide their relative likelihood. However, it is worth noting that substitution and deletion errors together appear to be far more common than insertion and transposition errors. In a study [7] of mistakes made in keying cash amounts in a bank central office the following frequencies were observed: substitution, 62.4%; deletion, 20.7%; insertion, 6.0%; transposition, 1.5%; other, 9.4%. With specific reference to these mistakes in FORTRAN text, James and Partridge [8] made the following observations: substitution, 24%; deletion, 58%; insertion, 18%; transposition, 0%. These observations are consistent with the observation that substitution and deletion are simpler actions than insertion and transposition.

Four program texts, taken from ACM Transactions on Mathematical Software, were used as subjects:

1. Algorithm 495 - Solution of an Overdetermined System of Linear Equations in the Chebyshev Norm [9];
2. Algorithm 498 - Airy Functions Using Chebyshev Series Approximations [10];
3. Algorithm 505 - A List Insertion Sort for Keys with Arbitrary Key Distribution [11];

4. Algorithm 513 - Analysis of In-Situ Transposition [12].

There are significant differences between them. In Algorithm 495 a two-dimensional array and two one dimensional arrays are prominent and there are no constants except a few small integers. In Algorithm 498 there is no two-dimensional array but there are some small one-dimensional arrays used as tables for real constants: the large number of real constants, nearly 200, it contains is a distinguishing characteristic of this algorithm, and it is the only one to contain WRITE and FORMAT statements. Algorithm 505 has variables and constants of type integer only and it has a one dimensional array of 57 integer constants initialized in a DATA statement. Algorithm 513 has a one-dimensional array of type real, all other variables are of type integer and it has just a few constants all of type integer. There are some differences in size: the number of lines, excluding COMMENT lines, in these algorithms is 208, 249, 71, and 81, respectively; the number of statements, excluding COMMENT statements, is 207, 168, 58, and 81, respectively; the number of characters, excluding blanks and COMMENT statements, is 2917, 6820, 1364, and 979, respectively.

After the samples were generated each was examined by eye to determine the kind of error caused by the simulated mistake. Four kinds of error were distinguished.

1. Syntax error: A violation of the language rules determinable by scanning the altered statement out of context.
2. Semantic error: A violation of the language rules determinable at compile or load time and not included in 1.
3. Anomalous use of a variable: Exactly one appearance of a variable name in a program unit, or use of a local variable only in a referencing context, or use of a local variable only in a defining context.

4. Other: Anything not covered by 1, 2, or 3.

The language rules referred to here are those for ANS FORTRAN 66 [5]. For a language like ALGOL or PASCAL the errors in the first category could be defined with respect to the formal grammar used to define them, but since FORTRAN 66 is defined only informally we are forced to use an informal definition of syntax and semantic errors here. However, this should not cause any serious misunderstanding. The nature of the mistakes we are considering is such that they are likely to cause an anomolous use of a variable to appear and most of these are recognized in category 3, however, they are included in this category only if they are determinable without path tracing - i.e., without recognizing the order in which statements are executed. It will be noted that no path tracing is required to recognize the fact that a variable name appears only once in a program unit,and provided it is not used in a procedure call it is possible to determine whether a local variable is used only in a referencing context or only in a defining context. These terms, reference and define, refer to fetching a value from memory and assigning a value, respectively: x is in a referencing context in y = x + 1 and y is in a defining context. Any anomalies which would require path tracing to detect them fall in category 4. A FORTRAN expert will recognize that category 3 includes certain errors that might have been placed in category 2 because it is a violation of the language to use a variable in a referencing context before it has appeared in a defining context and a variable used only in a referencing context is surely such a violation. Nevertheless it seemed more sensible to include these in category 3. One point about this classification needs to be emphasized. The anomalies or errors included in the first three

categories are easy to detect and we should expect that a good compiler will detect all of them. Those in category 4 on the other hand are substantially more difficult to detect and while some might be detectable by techniques of static analysis, others would not. In any case we would expect to use data flow analysis, testing, or some other technique to recognize them.

The results of this classification of the 1000 samples are displayed in Fig. 1, where the abbreviations used are defined as follows: SN, substitution mistake - nearest-neighbor substituted; SR, substitution mistake - any character substituted; DL, deletion mistake; IN, insertion mistake - any character inserted; TR, transposition mistake. The large number of real constants in Algorithm 498 explains why the SN, DE, and TR mistakes yield a high proportion of samples in category 4: real constants tend to be converted into real constants. The SN mistakes cause a lower percentage of syntax errors than SR mistakes because SN mistakes are more likely to substitute another character of the same type. The actual probabilities are: pr{ < letter > ⇒ < letter > } = 85% (56%), pr { < digit > ⇒ < digit > } = 74% (20%), pr{ < sp. character > ⇒ < sp. character > } = 57% (18%) where the number inside parentheses is the value for an arbitrary character substitution. When the results in Fig. 1 for all four algorithms are combined the distribution of errors over the four categories is: syntax error, 52%; semantic error, 18%; anomaly, 16%: other 14%. It is interesting to note that the result obtained here for the frequency of syntax errors agrees well with the result we obtained earlier by analysis.

Out of the one thousand samples, one hundred and forty fell in the fourth category representing errors relatively difficult to detect, and of these fifteen were one of the following types:

referencing an undefined variable, two definitions of a variable without an intervening reference, a null statement (e.g. x = x). It is reasonable to assume that these fifteen could be detected by data flow analysis or simple matching (for the null statements). Thus it appears that more than 10% of the errors caused by mistakes would remain until execution time for their detection, making generous assumptions about detection by static analysis.

It is natural in considering these results to wonder about the effectiveness of compilers in detecting these errors. Accordingly the samples produced by the SN mistakes were submitted for compilation to four different compilers: MNF, the University of Minnesota FORTRAN compiler; FTN, the CDC FORTRAN compiler; FORTH, the IBM FORTRAN H-level compiler; and WATFIV, the University of Waterloo compiler. In Fig. 2 the errors detected by these compilers are illustrated, with the number of errors in the first three categories shown for reference (marked EDE). It is evident that most of these compilers do little more than catch the syntax errors and some of the semantic errors. No results were obtained for WATFIV on algorithm 498 because of difficulties caused by the long DATA statements it contained.

Conclusion

These results have three applications. They contribute towards providing quantitative measures of the reliability of programs, they provide a base for the comparison of similar phenomena in other languages, and they provide a target at which the builders of FORTRAN compilers can aim.

Our ability to provide some quantitative measure for the reliability of a program is notably weak. In practice ad hoc techniques

based upon what appears to be reasonable are all that we have for judging a program to be reliable or, to put it another way, for estimating the number of errors it might contain. The results presented here provide us with some assistance with this problem. It is reasonable to assume that the density of transcription mistakes which remain in a program after proof-reading is sufficiently low to treat them as independent, non-interfering phenonema: if there is any doubt as to the validity of this assumption one would have little difficulty in testing it. Therefore by simply multiplying the frequency of mistakes in the text after proof-reading by the numbers obtained here we have an estimate of the number of mistakes which escaped detection after compilation and static analysis. The first factor can be measured independently and will certainly depend on the quality of the typists and proof-readers, but to show what might be expected from such a calculation we use some data that are available. James and Partridge [8] in a study of two hundred FORTRAN programs, composed of 20,121 statements altogether, found approximately three mistakes per thousand statements of which 90% were of the single character type: since we can assume James and Partridge did not find them all and since it is unclear precisely when, after proof-reading, they made their observations, we conclude that the number of mistakes in the programs after proof-reading was at least three per thousand statements. Other data on keying errors support this. In a study of mistakes made in keying statistical data by Deming, Tepping, and Geoffrey [13] they found that the "maximum error rate is one wrong card in one hundred cards punched." In a study concerned with the design of keyboards Klemmer [14] observed that "Experienced operators average 56,000 to 83,000 keystrokes per

day with 1,600 to 4,300 strokes per residual error" (a "residual error" is one remaining after detection and correction of the text by the typist): this translates to 0.25 to 0.6 residual mistakes per 1,000 characters. Klemmer's data is consistent with that obtained from the Oxford University Press for operators of typesetting keyboards: superior operators have an average error rate of about 0.5 residual mistakes per 1,000 characters. If we assume an average of about 14 (non-blank) characters per statement, as is the case for Algorithm 495, then we might expect between 3.5 and 8.4 mistakes per one thousand statements. It is a matter of conjecture as to how many of these might be caught in proof-reading. If we assume the worst, that is none caught in proof-reading, and we take the results obtained from the work reported here which show that about 85% of the errors caused by typing mistakes could be caught during compilation and static analysis, we obtain the result that after compilation and static analysis we could expect between 0.5 and 1.3 mistakes per one thousand statements. This result would be reduced in proportion with the number of mistakes caught by proof-reading; but on the other hand we have seen that existing compilers have a much poorer error detection rate than 85% tending to increase this result in actual practice. The density of mistakes remaining in a program when it is put into use, that is to say after testing, can then be estimated once we have a quantitative measure of test effectiveness.

There is an intuitive feeling people have to the effect that mistakes in programs written in Algol-like languages are less likely than in programs written in FORTRAN. Now so far as the kind of mistakes that we are treating here is concerned this difference, if it exists, will

be primarily due to the fact that an Algol-like language will make the mistake easier to detect: it is not so likely that the language difference would reduce the frequency of typing mistakes - indeed the larger alphabet of Algol-like languages could serve to increase the frequency of typing mistakes. An investigation carried out on programs written in other languages like the one carried out here on programs written in FORTRAN could resolve this issue and might provide some clues to language features which enhance, or inhibit, error detection.

Finally, we have seen from the results presented here that there appears to be considerable room for improvement in existing compilers. The main area needing improvement is anomaly detection, though it must be admitted that this is a difficult area to deal with because increasing the reporting of anomalies tends to increase the false alarm rate. Investigating the error detecting capability of existing FORTRAN compilers has not been an important theme of this work, however, the few results we have obtained in this direction suggest that further work in this area could serve as a stimulus to compiler writers and as a warning to the careless programmer who likes to leave it to the compiler to find the mistakes.

Part of this work was done while I was a visitor with the Numerical Algorithms Group in Oxford. I thank them for their hospitality and I also thank C. W. Gear who kindly ran my samples on a WATFIV compiler, and J. M. Boyle who did the same on an IBM FORTRAN compiler. Finally, I thank Dan Ruegg, Mario Escobar, and Carol Drey of the University of Colorado who assisted in gathering the data reported here.

References

1. Weinberg, G. M. and Gresset, G. L. An experiment in automatic verification of programs. Comm. ACM 6, 10 (Oct. 1963), 610-613.

2. Gilb, T. Software Metrics, Winthrop (1977).

3. Lipton, R. J. and Sayward, F. G. Hints on Test Data Selection: Help for the practicing programmer. Computer (April 1978), 34-41.

4. Knuth, D. E. An empirical study of FORTRAN programs. Software P. & E. 1,2 (1971), 105-133.

5. ANS FORTRAN (1966). American National Standards Institute, Inc., 1430 Broadway, New York, N.Y. 10018.

6. Shaffer, L. H. and Hardwick, J. Errors and error detection in typing. Quart. J. Exp. Psych. 21 (1969), 209-213.

7. Carlson, G. Predicting clerical error. Datamation (Feb. 1963), 34-36.

8. James, E. B. and Partridge, D. P. Tolerance to inaccuracy in computer programs. Computer J. 19, 3 (Aug. 1976), 207-212.

9. Barrodale, I. and Phillips, C. Algorithm 495 - Solution of an overdetermined system of linear equations in the Chebyshev norm. ACM Trans. on Math. Software 1, 3 (Sept. 1975), 264-270.

10. Prince, P. J. Algorithm 498 - Airy functions using Chebyshev series approximations. ACM Trans. on Math. Software 1, 4 (Dec. 1975), 372-379.

11. Janko, W. Algorithm 505 - A list insertion sort for keys with arbitrary key distribution. ACM Trans. on Math. Software 2, 2 (June 1976), 204-206.

12. Cate, E. G. and Twigg, D. W. Algorithm 513 - Analysis of in-situ transposition. ACM Trans. on Math. Software 3, 1 (Mar. 1977), 104-110.

13. Deming, W. E.; Tepping, B. J.; and Geoffrey, L.: Errors in card punching. J. American Statistical Association 37,220 (Dec. 1942), 525-536.

14. Klemmer, E. T. Keyboard entry. Appl. Ergonomics 2, 1 (1971), 2-6.

Figure captions.

Figure 1: The effect of five classes of typing mistakes (SN, substitution - nearest neighbor; SR substitution - any character; DE, deletion; IN, insertion; TR, transposition) on four algorithms. For each case the first interval on the bar graph denotes "syntax" errors, the second interval denotes "semantic" errors, the third interval denotes "anomalous use of a variable," and the fourth interval (shaded) denotes "other" errors. The fourth interval is shaded to clearly distinguish the errors in this class which are difficult to detect from those in the other three classes which are relatively easy to detect.

Figure 2: The effectiveness of four FORTRAN compilers in detecting errors caused by substitution (nearest neighbor) typing blunders. The percent of errors detected is shown. Also shown (EDE) is the percent of errors which are easy to detect, namely those in the three classes: 1, syntax; 2, semantic; 3, anomalous use of a variable.

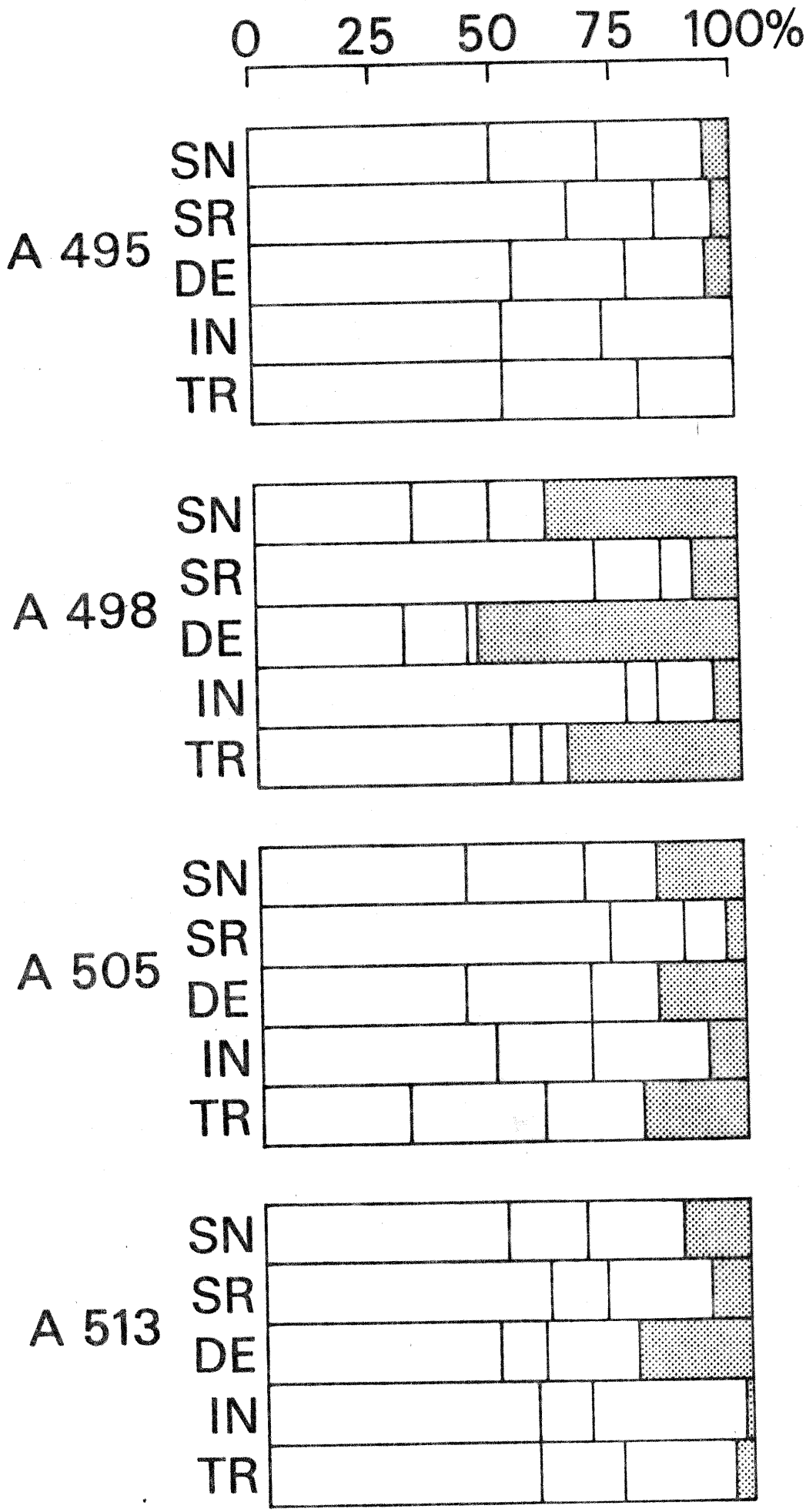
0
25
50
75
100%
A 495
SN
SR
DE
IN
TR
A 498
SN
SR
DE
IN
TR
A 505
SN
SR
DE
IN
TR
A 513
SN
SR
DE
IN
TR

Figure 1

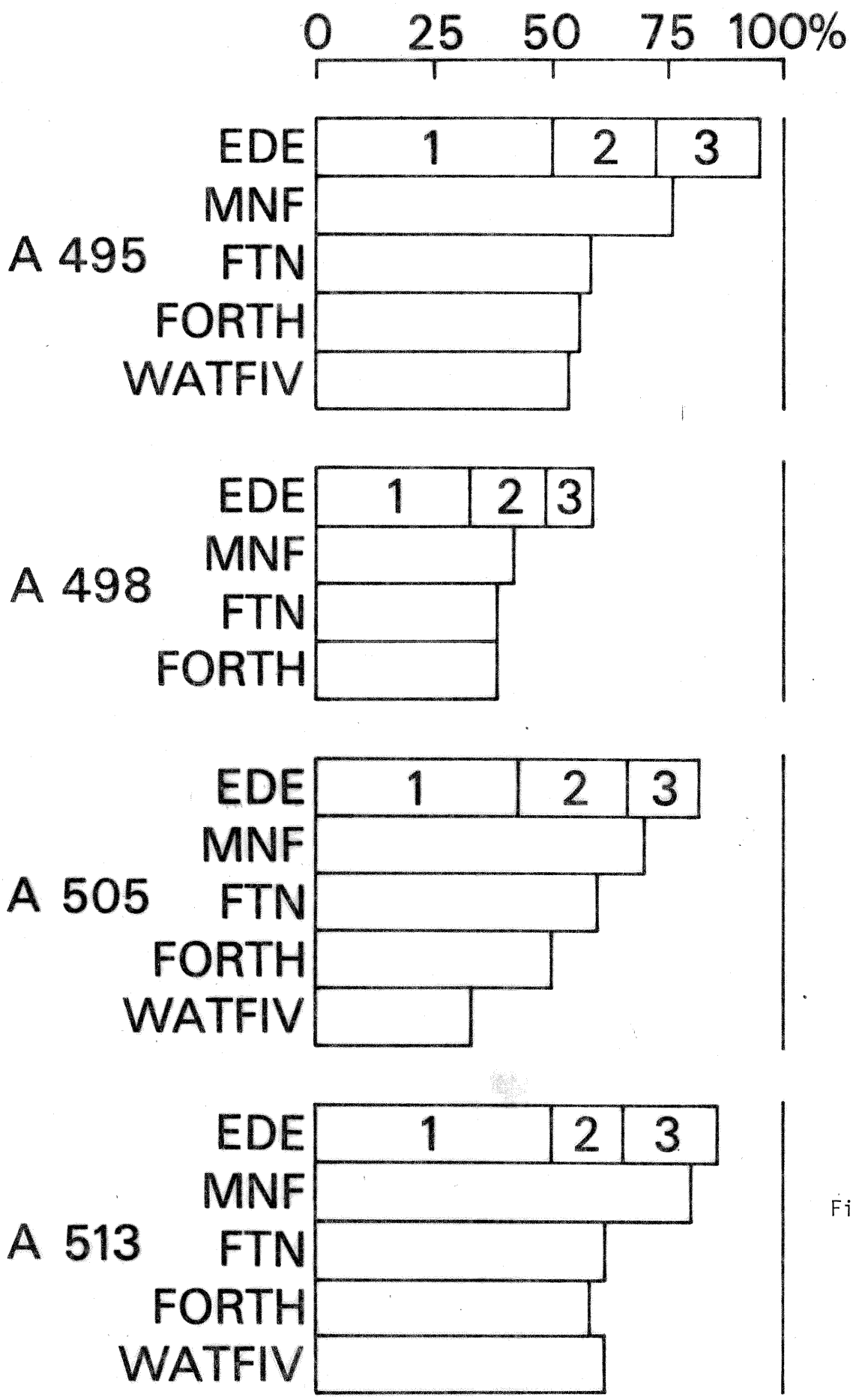
0 25 50 75 100%
A 495
EDE
MNF
FTN
FORTH
WATFIV
1
2
3
A 498
EDE
MNF
FTN
FORTH
1
2
3
A 505
EDE
MNF
FTN
FORTH
WATFIV
1
2
3
A 513
EDE
MNF
FTN
FORTH
WATFIV
1
2
3

Figure 2