FETE:
A FORTRAN EXECUTION TIME ESTIMATOR

BY

DANIEL H. H. IGNALLS

STAN-CS-71-204
FEBRUARY, 1971

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

FETE

A FORTRAN EXECUTION TIME ESTIMATOR

Daniel H. H. Ingalls

Introduction

If you want to live cheaply, you must make a list of how much money is spent on each thing every day. This enumeration will quickly reveal the principal areas of waste. The same method works for saving computer time. Originally, one had to put his own timers and counters into a program to determine the distribution of time spent in each part. Recently several automated systems have appeared which either *insert counters* automatically or interrupt the program during its execution to produce the tallies. FETE is a system of the former type which has two outstanding characteristics: it is very easy to implement and it is very easy to use. By demonstrating such convenience, it should establish execution timing as a standard tool in program development.

FETE is a three-step process. The first step accepts any FORTRAN (IV) program such as the example in Figure 1 and produces an edited file with counters. The second step executes the modified program, but retains the source file. After execution, the third step re-reads the modified source and correlates it with the final counter values to provide the listing shown in Figure 2. Here the executable statements have been collected and appear beside the exact number of executions and approximate computation time. The number of TRUE branches of logical IFs is tallied on the right, and subtotals appear at the end of each routine timed.

The Value of Execution Time Profiles

The second section of this paper will show how such a format for execution time feedback may be easily achieved. This first section jumps ahead to treat the implications of this tool for computer programming in general. The style may follow that of a patent medicine dealer describing his special brand of panacea, but the enthusiasm comes mainly from watching sceptical programmers using FETE for the first time.

Execution-time profiles are of value in three main areas of programming: improving old programs, writing new programs and educating programmers. In improvement of old programs it most often happens that the programmer initially does not know what the program does. Even when improving one's own program, much of the original scheme has probably faded from memory (and we all know how much the *comments* will help). The results of the appendix show that from a typical program, approximately 3 percent of the code constitutes 50 percent of the execution time. In some sense, then, we may conclude that if a naive programmer sets out to improve a program, he will work 30 times more effectively if he has a FETE (or similar) listing in front of him. Two words describe the programmers I have watched looking at their FETE runs: focussed attention. The human mind's most powerful tool is selective attention, but the selection requires an awareness about the environment which in this situation is furnished by a source-level presentation of execution time distribution.

Since FETE became operational, I have changed my own approach to programming. My three steps to creating a program used to be:

1 Think how I want to do it.

2 Write it up in the best way.

3 Debug it.

The numbers at the left are not to indicate order but are an estimate of how long the steps take. My new recipe is more like the following:

1 Think how I want to do it.
  Write it up in the quickest way.
1 Debug it.
0 Get a FETE listing.
1 Rewrite and debug the important parts.

The writing time is less because you assume that none of the program needs to be efficient (remember that only 3 percent does). The debugging time is less because the code you have to debug is really simple. The time to rewrite the important sections is low because although you try to write very efficient code, there is very little which needs this attention. The result is a program written in 2/3 the time, and which is much easier to understand because it is simply written. On top of that, it probably runs faster, because the inner loops have been specially written. The first run of FETE upon itself led to a twofold increase in speed!

The instructional value of execution-time awareness must be great. For one thing, the programmer will learn to recognize inefficient algorithms. Moreover, the reinforcement from FETE enhances the aesthetic enjoyment of writing a good program. The nicest reward which came from finishing FETE was being able to run it on itself, in part because it was fun to improve, and part because it was clear when the job was finished. Many people point out that good programs come from good algorithms, yet the implication is often that only programmers such as the critic are capable of choosing good algorithms. My feeling is that much mediocre programming comes about only because the programmer is lost in his program and can't see what is important. He would choose better methods if he had better perspective, and that is exactly what FETE and similar systems can provide.

The current approach to higher level languages aims at liberating the programmer from petty (hardware and archaic software) considerations. This is a laudable goal, but one must not include computation as *a* petty consideration. APL is a good example of a liberating language, but it also masks the huge amount *of* processing behind much of its vocabulary. The risk of conciseness is that a bad algorithm may fit on one line, and never be noticed. Incorporation of execution-time tallies into the new languages offers a solution to this problem, by maintaining the awareness of the programmer at the same level as the power of the language. Those contemplating new compilers would do well to include execution time profiles as an option for users.

Implementing Source-Level Execution Profiles

As summarized in the introduction, FETE is a three-step procedure. Since the second step runs as a normal FORTRAN job it entails no effort other than job-control organization. The bulk of this section is devoted to describing the details of the first and third phases of FETE.

Figure 3 shows the modified source produced from the program of Figure 1 during FETE's first step. The annotations (a) through (*l*) refer to Figure 3. The first insertion (a) is a labelled common declaration for the counter array. The dimension 2000 is adequate for most programs up to 6000 statements in length. The common declaration is inserted in all subprograms immediately following any SUBROUTINE, FUNCTION, or IMPLICIT statements. The names KOUNT1, KOUNT3, etc., are unlikely to conflict with users' names as they spelled with a zero, not an O . Initialization of the counters (b) occurs immediately before the first "noticeably" executable statement (a non-arithmetic executable statement). FETE makes no attempt to recognize statement functions because of the difficulty of inserting counters for them, and hence must assume that the first arithmetic statements might have been function definitions. The first counter must then be inserted (c) to tally the executions of any preceding arithmetic statements. From here on, counters need only be inserted where control branches and where logical ifs occur. For instance, we need counters immediately after a DO statement (d) because there is an implied loop entry at that point. Now note what became of statement 10. FETE removes each statement label (except those which terminate DO-loops), and attaches it to an inserted counter (e). In this way, each time control branches into the main

line of code, the extra executions will be recorded. If a CONTINUE statement is stripped of its label in this way, it will be deleted from the source, and a flag set in the counter so that it *may* be recreated for the final listing.

When FETE encounters a logical IF, it first strips off the target statement and replaces it by a counter. The resulting IF statement is then inserted (f) above the original. Thus even if the original IF would cause a branch out of line, the fact that the branch was taken will be recorded by the counter. Usually the editing of IFs can be done on one line, as is the case in our example; however, when the IF clause is too long (typically less than 5 percent of the time), appropriate continuation cards are generated for the IF-counter. Most of the time, FETE does not insert counters after IF statements. Almost all target statements of IFs are either arithmetic or GO TOs. In the former case, the main-line execution count will be unchanged; in the latter, it must be decreased by the value of the IF counter (i.e., the number of branches out of line). The analysis routine which reads the counters can determine which was the case by examining the sequence-column flags (q.v.). In indeterminate cases, such as a CALL with multiple returns, or a READ with ERR return, FETE inserts a counter after the IF to be safe.

Note (g) of Figure 2 indicates a labelled statement which has not been modified in the manner of the other labelled statements. The terminal statement of a DO-loop presents a special problem to execution tallying. On the one hand we need a labelled counter before the statement in question for the tallies and so that transfers to the label will work properly; yet that would end the DO-loop above the statement originally labelled, and exclude it from the loop. Fortunately, though, we have enough extra information to solve the dilemma. The following simplified code segment illustrates the situation:

```
      ...
      K(n) = K(n)+1
      DO 10 I = I1,I2
      K(n+1) = K(n+1)+1
      ...
10    P(I) = F
      K(n+2) = K(n+2)+1
      ...
```

One thing we know for sure: K(n+1) would have the correct tally for statement 10 if there were no branches out of the DO-loop. In fact if we could subtract from K(n+1) the number of branches out of the DO-loop, then we would have the answer. Now we note that the only way for K(n) to be stepped without K(n+2) increasing also is if there is a branch out of the loop. Thus we obtain our result that P(I) = F must have been executed K(n+1)-K(n)+K(n+2) times. The interested reader may deduce the result for a statement which terminates two nested DO-loops with the same end-label.

When FETE encounters a STOP (or CALL EXIT or RETURN in the main program) it inserts a call (h) to the analysis routine (KOUNT1) which goes back to correlate the modified source with the counter contents. Provision is also made for termination in an IF statement such as

```
IF (NCARD.EQ.LAST) STOP
```

Here the IF clause will be repeated three times; once with a counter, once with the CALL, and a last time with the STOP.

FETE handles SUBROUTINES and FUNCTIONS in the same manner as the MAIN, except that no counter initialization is inserted and a RETURN is not treated as a STOP. We move on now to deal with the sequence-column flags before summarizing the task of the analysis routine.

The sequence column fields of Figure 3 are denoted i,j,k,ℓ . Field j is a two digit code for the statement type (1 = arithmetic, 2 = DO, 3 = IF, 4 = GO TO, etc.). Since logical IFs are flagged in the i-field, their j-field is used to give the classification of the target statement. The k-field is a two-digit index of the depth of DO-nesting. Actually, this value does not increase with every DO encountered, but only when the DO refers to an end-label not yet used in previous DOs. The convention economizes on stack space, and yet gives enough information to the analysis routine. The ℓ-field gives the "cost" of

each statement, and is responsible for the 'dirty' in FETE's designation as a quick-and-dirty system. FETE determines cost by a linear scan of each executable statement which looks for operators, parentheses, etc., charging a reasonable fee for each. Another base cost is derived from the statement type, and the operator cost is then added on. In statements such as WRITE or FUNCTION, a further charge is levied for each comma encountered to reflect the extra argument overhead. At each left-parenthesis a check is made to see if the preceding identifier was a FORTRAN internal function name, and if so the appropriate cost is added on from a table.

Most of the cost of a CALL is put into the corresponding SUBROUTINE statement. The justification is a human engineering consideration, to suggest to a programmer the possibility of writing his subroutine in line to save time. To evaluate that suggestion, the programmer really wants to see the total cost of the subroutine linkage in one number, rather than in five calls scattered throughout his program. The same convention is especially appropriate for FUNCTION statements, because FETE's lack of a symbol table precludes detection of the implied calls, yet the tallies in the function code will be correct.

Future versions of FETE will use a more elegant cost assessment, but this crude scheme has been remarkably successful. The source editing is performed in one pass without scratch files, and takes roughly 1/5 as long as the FORTRAN compilation.

The analysis routine, which comprises FETE's third phase, is linked in during the FORTRAN step, so that it may be called just before the program would have come to a STOP. This phase rereads the modified source and correlates the executable statements with the counter values and prints the FETE listing in one last pass.

The i-field of the sequence-column flags was originally intended as a coded column of useful facts for the analysis routine. However, as that routine took shape, it became clear that these numbers worked as op-codes for an analysis-machine. This is one of several instances where I have found new insight into a problem by considering its data-to-program relationship to be a form of program-to-machine relationship. I have chosen to lay this on the reader by roughly outlining the order code of the analysis machine in Table 1 and inviting him to simulate the analysis of this sample program.

As the analysis routine proceeds through the file, it maintains subtotals and totals of executions and cost and prints these for the programmer to use for judging relative importance of different parts of the listing. Percentage cost is not given for two reasons. First is the necessity for an extra pass through the source file (or a smaller file with static costs only). Second is the observation that people using FETE simply scan the cost column visually for the number of digits, a process for which FETE's large integers are ideally suited. A simple statistic which I included out of curiosity is the running total of the executions and costs squared. From these and the normal totals, the r.m.s. values may be compared with the mean values to give an idea of how "peaky" the execution and cost are. All of these statistics are currently printed out in a table for instrumentation curiosity, and some results gathered from 17 sample programs appear in the appendix.

The FETE approach to determining actual timing is a very coarse one, but has proved to be 90 percent effective in giving programmers what they want. Other workers have developed compilers incorporating the whole execution-timing process, and that is obviously the proper approach. With the symbol table available, the timing of Input/Output statements can be assessed, the code-generator can give exact timings for the other statements and the insertion of counters is efficient, both in placement and in code generated. Furthermore, the compiler's run-time routines can usually pick up the pieces after a program dies or runs out of time, and the FETE enumeration of executions would be informative in such cases.

FETE has proved to be very useful at Stanford. A version to work with WATFIV allows inexpensive timing for use on the level of student programming assignments. The morality of enhancing FORTRAN may be suspect, but hopefully the optimistic results described here will inspire availability of execution time profiles in all languages before any damage is done.

APPENDIX -- Program Localization

A phenomenon of considerable interest is the manner in which programs tend to spend all their time in a very small portion of their total code. The first version of FETE included instrumentation for investigating the effect, and this appendix describes the results.

Let us suppose that we have an N-statement program of which only k statements are significant, and these are equal in cost. The fraction of statements required to make up 50 percent of the execution time of this program would be k/2N . Letting c(j) be the cost of the j-th statement, we can define a mean cost M and a root-mean-square cost, as

$$M = \frac{1}{N} \sum_{j=1}^{N} c(j) \qquad R = \sqrt{\frac{1}{N} \sum_{j=1}^{N} c^2(j)}$$

where the summations are over all statements of the program. For our hypothetical program, we may let $c = T/k$ for k of the statements and $c = 0$ for the other N-k statements, so that $M = T/N$ and $R = T/\sqrt{kN}$. I now tentatively define the Ingalls factor $I = M^2/2R^2$, which should give the number of statements making up 50 percent of execution time. Such a definition for I is motivated by the observation that r.m.s./mean measures how 'peaky' a function is over the interval considered.

Figure A1 shows a plot of the tentative I-factor against the actual 50 percent factor for 17 randomly selected programs. These varied in size from 100 to 3500 cards and in content from numerical integration to a meta-compiler. Both the linearity and the uniformity of the points over the sample programs are striking. Departure of the slope from unity is due to the invalid assumption that all significant statements have equal weight. Redefining the I-fact as $I = 0.4\ M^2/R^2$, we have an empirically good measure of program localization. Moreover, the plot shows a value of 3 percent to be typical. Without a more detailed study of program graphs, this statistic does not imply much about how programs should be partitioned. However, the 3 percent figure does demonstrate the enormous potential of source-level timings for focusing attention on inner loops.

FETE grew out of a research project in programming languages led by Donald Knuth and supported by IBM Corporation, Xerox Corporation and ARPA. I am also indebted to Dick Sweet at Stanford for the FORTRAN statement classifier used in FETE.

Bibliography

Cerf, V. G. "Measurement of Recursive Programs." Ph.D. Thesis, School of Engineering and Applied Science, University of California, Los Angeles, California, Report 70-43, May 1970, 106 pp.

Conrow, K. and Smith, R. "NEATER2: A PL/I Source Statement Reformatter." CACM, Vol. 13 No. 11 (1970), pp. 669-675.

Darden, Stephen C. and Heller, Steven B. "Streamline your software development." Computer Decisions 2 (October 1970), 29-33.

Russell, E. C., Jr. "Automatic Program Analysis." Ph.D. Thesis, School of Engineering and Applied Science, University of California, Los Angeles, California, Report 69-12, March 1969, 168 pp.

Satterthwaite, E. "Source Language Debugging Tools." Ph.D. Thesis, Stanford University, in preparation.

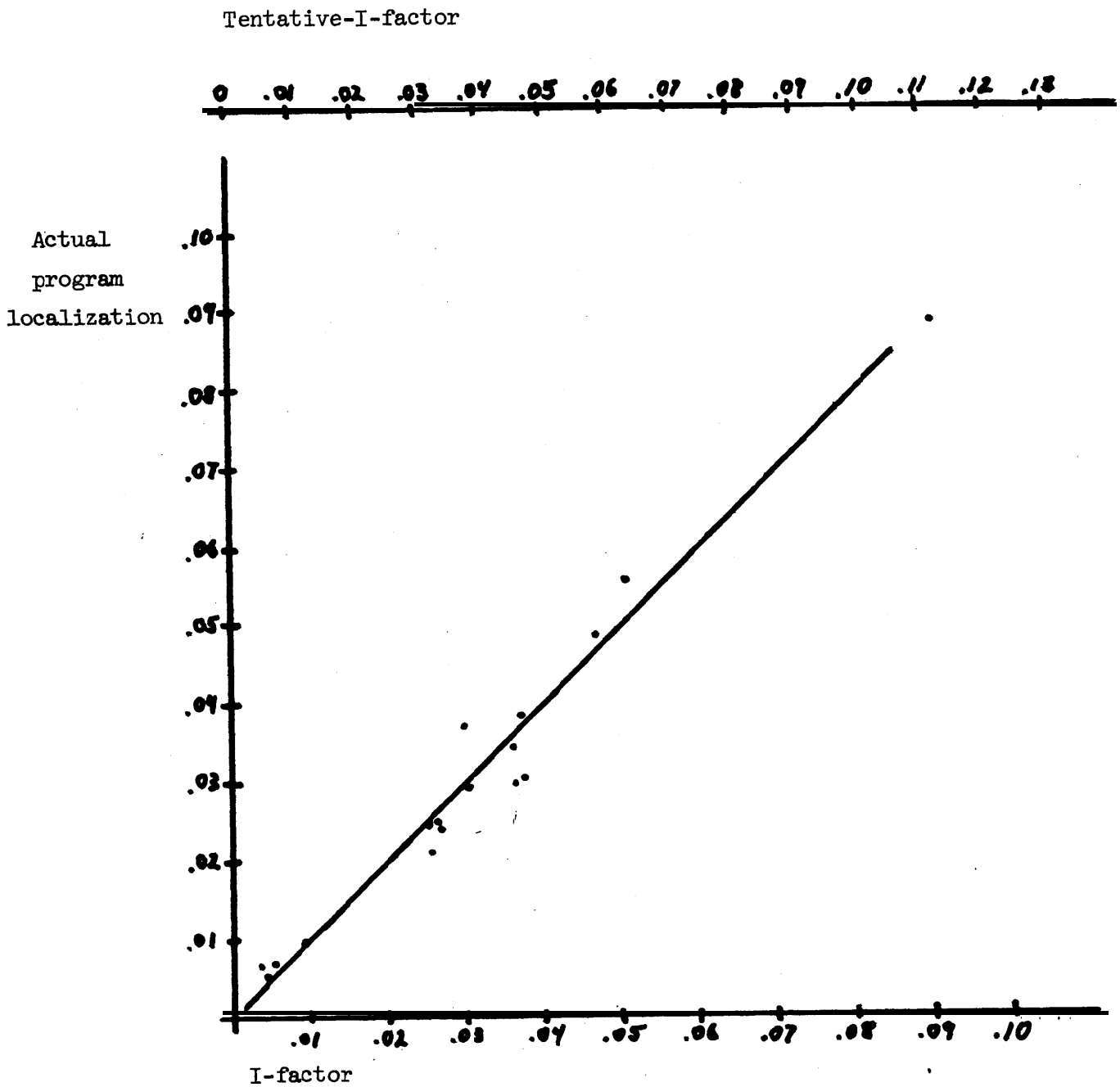

Figure A1: Comparison of I-factor with actual program localization for 11 user programs

```
C       PRINT OUT FIRST 100 PRIMES
        INTEGER PRIMES(100)
        PRIMES(1) = 2
        PRIMES(2) = 3
        N = 3
        DO 30 INDEX=3,100
C       GET NEXT (ODD) CANDIDATE
10      N = N + 2
C       RUN THROUGH POSSIBLE (PRIME) DIVISORS
        K = 2
20      IQUOTN = N/PRIMES(K)
        IF(PRIMES(K)*IQUOTN.EQ.N) GO TO 10
        IF(IQUOTN.LE.PRIMES(K)) GO TO 30
        K = K + 1
        GO TO 20
30      PRIMES(INDEX) = N
        WRITE(6,40) PRIMES
40      FORMAT('1 THE FIRST 100 PRIMES ARE:',13(/8I10))
        STOP
        END
```

Figure 1. Listing of sample program to be timed.

```
      EXECUTABLE STATEMENTS                                  EXECUTIONS      COST        TRUE

      PRIMES(1) = 2                                               1           2
      PRIMES(2) = 3                                               1           2
      N = 3                                                       1           1
      DO 30 INDEX=3,100                                           1           2
10    N = N + 2                                                 269         538
      K = 2                                                     269         269
20  IQUOTN  =  N/PRIMES(K)                                      911        8199
      IF(PRIMES(K)*IQUOTN.EQ.N) GO TO 10                        911        7459         171
      IF(IQUOTN.LE.PRIMES(K)) GO TO 30                          740        2318          98
      K = K + 1                                                 642        1284
      GO TO 20                                                  642         642
30    PRIMES(INDEX) = N                                          98         294
      WRITE(6,40) PRIMES                                          1         506
      STOP                                                        1           0
      END
                  SUBTOTALS FOR THIS ROUTINE - - -             4757       21516

*****  16 EXECUTABLE,    2 NON-EX,    3 COMMENTS; TOTALS:      4757       21516
```

Figure 2. Sample program timed. Only executable statements are displayed

```
                                                                        I  J  K  L
                                                                      ----------
a)          COMMON /KOUNT2/ KOUNT5(2000),KOUNT3                       0
            INTEGER PRIMES(100)                                       027  1  0
            PRIMES(1) = 2                                             1  1  1  2
            PRIMES(2) = 3                                             1  1  1  2
            N = 3                                                     1  1  1  1
b)          DO 83294 KOUNT3=1,2000                                    0
      83294 KOUNT5(KOUNT3)=0                                          0
c)          KOUNT5(   1)=KOUNT5(   1)+1                               5
            DO 30 INDEX=3,100                                         1  2  2  2
d)          KOUNT5(   2)=KOUNT5(   2)+1                               5
e)    10    KOUNT5(   3)=KOUNT5(   3)+1                               6  1  2  2
            N = N + 2                                                 1  1  2  2
            K = 2                                                     1  1  2  1
      20    KOUNT5(   4)=KOUNT5(   4)+1                               6  1  2  9
            IQUOTN = N/PRIMES(K)                                      1  1  2  9
f)          IF(PRIMES(K)*IQUOTN.EQ.N)              KOUNT5(   5)=KOUNT5(   5)+15
            IF(PRIMES(K)*IQUOTN.EQ.N) GO TO 10                        3  4  2  8
            IF(IQUOTN.LE.PRIMES(K))                KOUNT5(   6)=KOUNT5(   6)+15
            IF(IQUOTN.LE.PRIMES(K)) GO TO 30                          3  4  2  3
            K = K + 1                                                 1  1  2  2
            GO TO 20                                                  1  4  2  1
g)    30    PRIMES(INDEX) = N                                         2  1  2  3
            KOUNT5(   7)=KOUNT5(   7)+1                               5
            WRITE(6,40) PRIMES                                        118 1506
      40    FORMAT('1 THE FIRST 100 PRIMES ARE:',13(/8I10))           033 1506
h)          CALL KOUNT1                                               0
            STOP                                                      1  7  1  0
            END                                                       721  0  3
```

Figure 3. Sample program editted by FETE. Lower case letters refer to text.

i-field	Operation	Comment
0	If j not blank then tally static. Set ISEXEC = NO.	Not executable or not from original source.
1	Dynamic count is KOUNT5(IK); Tally static, dynamic, and by cost; Set ISEXEC = YES; Print with counts; if k = 2 , push 0 onto DO-stack if new DO-label, then add KOUNT5(IK+1)-KOUNT5(IK) to top of DO-stack; if k = 21 (END), then print subtotals and set ISFRST = YES.	Executable statement.
2	Dynamic count is KOUNT5(IK+1) + top of DO-stack; pop DO-stack; proceed otherwise as when i = 1 .	End of a DO-loop.
3	IF count is KOUNT5(IK-1); TRUE count is KOUNT5(IK); if j = 1 then move KOUNT5(IK-1) into KOUNT5(IK); if j = 4 then move KOUNT5(IK-1)-KOUNT5(IK) into KOUNT5(IK); Proceed otherwise as when i = 1 .	Logical IF.
4	If ISEXEC print with counts.	Continuation card.
5	If not ISFRST, IK = IK+1 ; set ISFRST = NO.	Inserted counter.
6	Save label and append to next line with i ¬ = 4 ; If j = 12 , create CONTINUE statement as next line; proceed as when i = 5 .	Labelled counter.
7	Print END followed by subtotals and totals; Number source comments is 1000*k+1; Print table of statistics; RETURN.	Last statement of program.

Table 1: order code of the analysis machine. Initial conditions are ISFRST = YES and IK = 1. Tallying is described in text.