MEASURING COMPUTER PROGRAM COMPREHENSION

John P. Boysen
Iowa State University Computation Center
and
Roy F. Keller
Department of Computer Science
Iowa State University
Ames, Iowa 50011

1.1 INTRODUCTION

While improved programming methodologies, better computer languages and more sophisticated programming aids have helped alleviate some problems associated with software development, a software crisis continues to exist. The software crisis continues partly because many of the suggested improvements in software development have emphasized the role of the computer, rather than the programmer, in the development process. Researchers are beginning to realize that the ultimate resolution of the software crisis will come only when we understand the human processes involved in software development.

Computer program comprehension has been one of the human processes which has been studied by researchers. Program comprehension is an important area of research for several reasons. First, as the program is developed it must be understood if it is to solve the intended problem. Second, comprehension is essential to debug the semantic aspects of the program. Finally, modification of the program requires an understanding of the program if the modifications are to be successful. Thus, improved comprehension can help to alleviate many of the difficulties encountered in the software development process.

Two basic approaches have been used to study program comprehension. Using the first approach, an objective measure of comprehension is proposed based on the author's suppositions about the sources of complexity. For example, McCabe (MCCA76) has suggested that the complexity of a program is directly related to the number of independent data paths in a program. Another measure of comprehension has been developed by Gordon and Halstead (GORD76). Using Halstead's software science measures (HALS77), Gordon developed a measure which purports to predict the time required to understand a program.

A second approach used in the study of program comprehension is to empirically investigate factors which might affect comprehension. For instance, Love (LOVE77) constructed versions of programs which differed in control flow complexity and paragraphing style. Subjects were presented with program versions and were asked to memorize them. The number of lines recalled was used as the measure of comprehensibility. Love found that paragraphing did not significantly affect the ability to recall the programs but complex control flow did. Also, graduate students were more adversely affected by complex control flow than undergraduate students.

While all the studies cited have helped advance the understanding of program comprehension, some methodological difficulties still exist. In many cases, proposed comprehension measures have only been validated using other author's opinions of comprehensibility rather than using an empirically-based measure. In other cases, the empirical measures have been too subjective or difficult to construct to be useful for validation purposes. Even the best developed measure, the number of lines recalled, has short-comings. First, such a measure cannot detect intrastatement factors of complexity since the unit of measurement is the statement. Furthermore, the goal of memorizing a program may differ from the goal of understanding the same program, and Weinberg (WEIN74) has shown that the goal of an experiment can have a marked effect on its outcome. What is still needed in comprehension research is an empirical measure of comprehension which is objective, easy to employ and directly related to the task of comprehension.

In the next section, a methodology is proposed to measure the comprehension of statements and programs. It is applied to study expression complexity in section three and selection statement complexity in section four. Implications for teaching programming are described in section five and the paper is concluded in section six.

1.2 A REACTION TIME METHODOLOGY

In the search for an acceptable empirical measure of comprehension, many techniques have been tried including hand-simulation, multiple choice tests and rankings of comprehensibility by subjects. Unfortunately, all of these techniques were either too subjective, too difficult to construct or were not adequate measures of

comprehensibility. However, one measure which has been successfully used in psycholiguistics to measure comprehension has been reaction time. For example, Gough (GOUG65) used a reaction time methodology to study the effects of truth, affirmation and voice in the understanding of English sentences. Subjects were presented with a sentence and picture and asked to indicate as quickly as possible whether the sentence correctly described the picture. The time to respond with the answer was used as a measure of how difficult the sentence was to comprehend. In Gough's experiment, the sentences were varied according to truth (whether the sentence accurately described the picture), affirmation (whether the sentence was stated negatively or positively) and voice (whether the sentence was stated in the active or passive). Gough found that true sentences were processed significantly faster than false, active faster than passive and positive faster than negative.

A similar approach might be applicable to the study of computer program comprehension. Programs, or segments of programs, could be presented to subjects and the time they took to understand the program could be a measure of the comprehensibility of the program. Such a methodology offers several advantages. First, it is easily measured, especially if a terminal is used to display the program and record the reaction time. Second, it is obviously objective. Finally, the subject is asked to comprehend the program as opposed to some other, possibly related, activity like memorization. In the next section the reaction time methodology is employed to study factors affecting statement complexity.

1.3 EXPRESSION ANALYSIS

1.3.1 Apparatus and Procedure

The study was conducted using a PLATO terminal. The PLATO IV terminal consists of an 8-inch square plasma screen, touch panel and keyset. Output text and graphics can be displayed on the plasma screen which, while using different technology, resembles the more familiar CRT display. Input to the terminal is via the keyset or touch panel. All reaction times were recorded using the touch panel as the input device.

The experiment began by giving the subject a practice reaction test on the terminal to help the subject become familiar with the operation of the touch panel. The actual experiment began following the practice test. Subjects were presented with a display such as

```
X := 6
X > 4
```

and were asked to indicate, as quickly as possible, whether the expression 'X > 4' was true or false. The subject indicated the answer by either touching a box on the screen which contained a 'T' or a box containing an 'F'. For the IF and CASE statements, the subject touched one of ten boxes on the screen, each box containing one of the ten digits. The expression to be evaluated was initially presented on the screen and the first assignment statement was displayed above the expression two seconds later. In subsequent trials, only the assignment statement was changed. The reaction time was measured from the time when the assignment statement was presented until the subject touched the screen.

The subjects were presented with nine different expressions which are shown in Table 1. For each expression, the values for X were the digits '0' through '9'. The order of presentation of both digits and expressions was random for each subject. The logical expressions were chosen so that an equal number of true and false answers were given. In the case of EQ and NE, eight additional fives had to be presented to maintain the same number of true and false answers.

1.3.2 Subjects

The subjects were both graduate and undergraduate students at Iowa State University. The graduate students were volunteers from an operating systems course and an advanced programming language course, while the undergraduate subjects were volunteers from a beginning programming language course. Thirty-four undergraduate and fifteen graduate students participated. A summary of the subjects' background data is shown in Table 2.

1.3.3 Results

The statistical design used for the simple expression analysis was a split-plot factorial with two repeated measures: expressions (9) and digits (10). The statistical analysis for the expressions LT,GT,AND,OR and NOT is shown in Table 3. Similar results were obtained for EQ and NE, and for the IF and CASE expressions but are not included here for the sake of brevity. A more detailed description of the experiment is detailed in (BOYS79). The mean reaction times by expression and digit are shown in Figure 1. The error rate was 4.5 per cent.

The results of the statistical analyses can be summarized as follows:

1. The mean processing times for the expressions were significantly different. The expressions were ordered in increasing difficulty as follows:
   EQ,NE,GT,LT,AND,NOT,IF,OR,CASE

2. Experience had no significant effect on the processing of the expressions.

3. Expressions which were true were processed more quickly than those which were false. The only exception occurred with expressions involving negation (NE and NOT), where no significant difference between true and false expressions was found.

4. For all comparison operations, it was significantly more difficult to compare two numbers when they were equal than when they were not.

TABLE 1

Expressions used in First Experiment

Symbol used in Text	Expression
EQ	X=5[1]
NE	X≠5
GT	X>4
LT	X<5
AND	X>2 AND X<8
OR	X<3 OR X>7
NOT	NOT(X<5)
IF	IF X<5 THEN Y=1; ELSE Y=2;
CASE	CASE X>=0 AND X<2 DO; Y=4; END; X>=2 AND X<4 DO; Y=2; END; X>=4 AND X<6 DO; Y=3; END; X>=6 AND X<8 DO; Y=1; END; ELSE DO; Y=5; END; ENDCASE;

[1]The values for X were the digits 0-9. The order of the digits was randomized for each expression and subject.

TABLE 2

Background of Subjects

Measure	N	Mean	S.D.
Months Programming			
Undergraduate	34	26.85	20.37
Graduate	15	39.26	29.51
Size of Largest Program[1]			
Undergraduate	34	3.41	.66
Graduate	15	3.47	.64
Time of Session[2]			
Undergraduate	34	35.32	18.59
Graduate	15	27.87	6.53

[1]Subjects were asked to classify the size of the largest program as less than 10,100,1000 or 10,000 lines. Thus, the measure is in terms of log(lines).
[2]In minutes.

TABLE 3

Statistical Analysis for LT,GT,AND,OR and NOT

Source	DF	MS	F
Experience	1	1.060	.76
Subject(Experience)	47	1.400	
Expression1	4	3.325	14.82**
Experience*Expression	4	.530	2.36
Subject*Expression(Experience)	188	.224	
Digit	9	.368	4.43**
Experience*Digit	9	.063	.76
Subject*Digit(Experience)	423	.083	
Expression*Digit	36	.318	4.15**
Experience*Expression*Digit	36	.056	.73
Subject*Expression*Digit(Experience)	1692	.077	

1A Duncan test for the means(in seconds) was (means not significantly different are underlined):

GT	LT	AND	NOT	OR
.842	1.021	1.043	1.080	1.429

**p<.01

1.3.4 Discussion

The fact that even single operators differ in complexity is important to the development of an acceptable comprehensibility measure. For example, the software science measure proposed by Gordon is based on the count of operators and operands in a program. No provision has been made for the differences between types of operators. If the mix of operator types is not constant across programs, then Gordon's measure cannot accurately predict those aspects of program comprehensibility which are affected by statement complexity. Not only is the complexity of statements affected by the types of operators in the sentence, but the complexity is also affected by the data being processed -- a fact supported by psycholinguistic research (BANK76). While data dependence is of limited practical value to computer program comprehension, it does indicate that human processing of statements can differ quite markedly from what we expect.

The fact that operators are ordered in difficulty is more practically useful. Since OR was more complex than AND, the expression 'DO WHILE X<=4 OR X>=8' could better be written as 'REPEAT UNTIL X>4 AND X<8' assuming that both tests were evaluated at the same point in the program. To provide the programmer with the flexibility to state conditions in the clearest format requires diversity in the control constructs and operators available in a programming language. Such diversity is rarely present in most current languages.

1.4 PROGRAM VERSION ANALYSIS

In reaction time research, it is assumed that if one statement is more comprehensible than another, it will take a subject less time to comprehend it. If this reasoning is extended to the evaluation of two versions of a program, the version which takes less time to comprehend will be more comprehensible.

But what does it mean to "comprehend" a program? Shneiderman (SHNE77) defines comprehension as

> the recognition of the overall function of the program, an understanding of intermediate level processes including program organization and comprehension of the function of each statement in a program.

If recognizing the overall function of the program demonstrates that a subject understands the program, than a possible measure of comprehensibility would be the time it took the subject to discover and state the program's function. If two versions of a program exhibited the same function, then the version which revealed its function more quickly would be judged more comprehensible. In the next section, this methodology is applied to the investigation of the effect of selection statement complexity on comprehension.

1.4.1 Procedure

The same experimental setting and subjects used

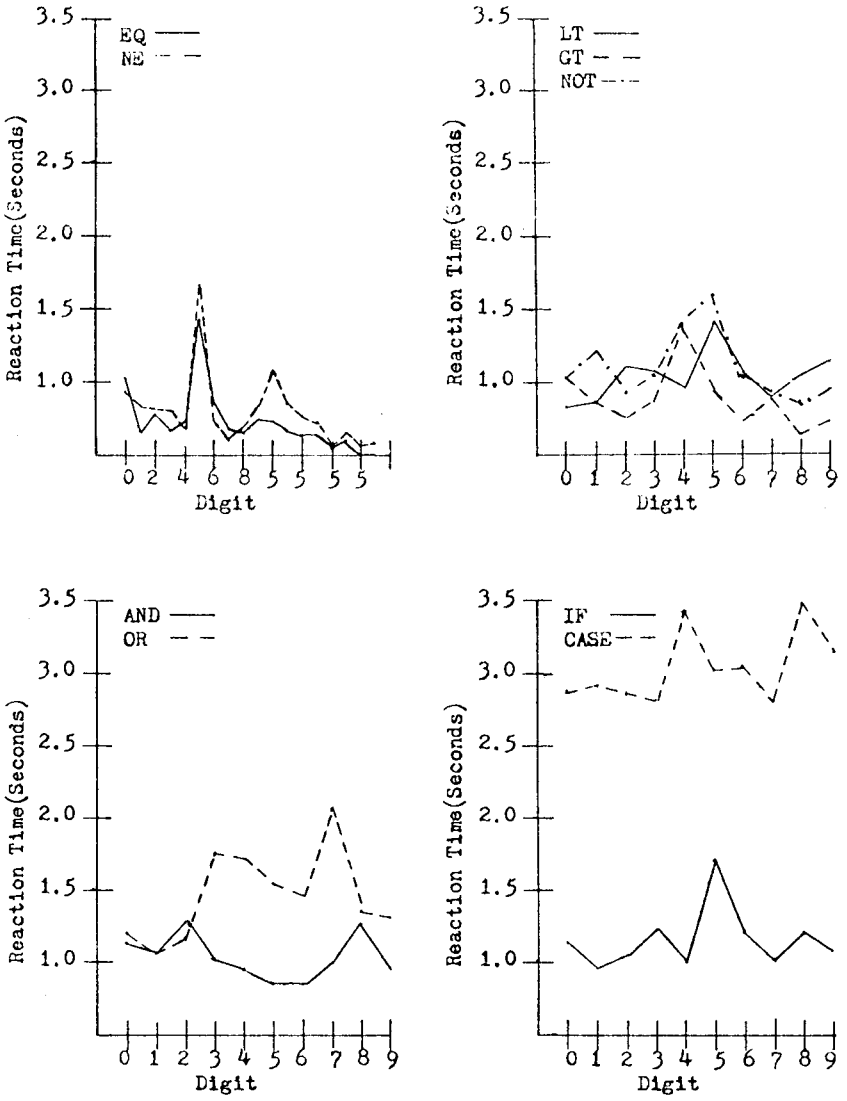

Figure 1: Reaction Times by Digit for Expression Analysis

TABLE 4

Statistical Analysis of Undergraduate Program Data

Source	DF	MS	F
Blocks	5	.029	.43
Subjects(Blocks)	28	.069	
Program	2	.917	17.20**
Version	2	.213	4.00*
Program*Version	4	.034	.63
Residual	28	.053	

*p<.05
**p<.01

TABLE 5

Statistical Analysis of Graduate Program Data

Source	DF	MS	F
Blocks	5	.117	.58
Subjects(Blocks)	9	.200	
Program	2	.142	3.38
Version	2	.051	1.21
Program*Version	4	.152	3.61*
Residual	18	.042	

*p<.05

in the expression experiment were used in the program version experiment. Each subject was initially presented with instructions on the conduct of the experiment via the terminal. After completing a practice problem, the subjects were presented with the first of three programs. The program was displayed on the screen and the subject was asked to discover the function of the program as quickly as possible. The functions included the computation of a minimum, maximum and the merging of two sorted arrays. As soon as the function was discovered, the subject touched the screen and the program was erased. The subject then wrote down the function of the program on paper. The time from the presentation of the program until the subject touched the screen was used as the reaction time. When the subject touched the screen again the next program was presented.

Since the data would be biased if subjects saw two versions of the same program, the program versions were presented so that each subject saw each program and each version but not all nine combinations. Version 0 of each program used nested IF statements while version 2 used a CASE statement similar to an INCASE (KELL77). Version 1 consisted of sequential IF statements for program 0 and program 1 and an IF-ELSE-IF construction for program 2. These programs are shown in Appendix A.

1.4.2 Results

The data for the undergraduate and graduate students were analyzed separately. For each group, a randomized block, partially confounded factorial design with repeated measures was used (KIRK68). Subjects were randomly assigned to one of six groups, each group seeing three program versions.

The statistical analyses for the undergraduate and graduate subjects are shown in Tables 4 and 5, respectively. Since it seems likely that the

TABLE 6

Summary of Program Version Measures

Program	Version 01	Version 1	Version 2
0			
Mean Reaction Time2	(54,72.3)	(59.9,51.9)	(46.7,28)
Correct Statements3	(7/12,4/5)	(7/11,5/6)	(10/11,4/4)
Ec4	732	320	1095
C5	4	3	3
1			
Mean Reaction Time	(91.3,83.4)	(44.7,68.2)	(72,79.4)
Correct Statements	(7/10,5/5)	(5/12,4/5)	(4/12,4/5)
Ec	1644	935	1540
C	3	3	3
2			
Mean Reaction Time	(230,42.5)	(122,134.2)	(93,87.9)
Correct Statements	(9/12,5/5)	(10/11,4/4)	(11/11,6/6)
Ec	13539	13204	16382
C	5	5	5

1Parenthesized entries are listed as: (undergraduate,graduate)
2In seconds
3Entries are listed as: number correct/total
4Gordon's measure of understanding
5Cyclomatic number

subjects who misstated the functions could bias the analysis, only those reaction times for which the function statement was correct were included in the analysis shown above. However, similar conclusions were obtained when all the data were included in the analysis.

As expected, the programs differed in comprehensibility although this effect only approached significance ($p<.06$) for the graduate subjects. Across programs, the versions differed significantly for the undergraduates while the effect of versions depended on the program for the graduate students.

A summary of the mean reaction times and number of correct statements is shown in Table 6. Based on the mean reaction times, the CASE version was the most comprehensible version for program 0, while the sequential IF version was most comprehensible for program 1. In program 2, the undergraduates found the CASE to be most comprehensible while the graduates understood the nested IF version the quickest. The analysis of the number of correctly answered functions resulted in slightly different findings. The fewest function misstatements were made for the CASE version of program 0 and program 2, and the most function misstatements for program 1.

1.4.3 Discussion

The superiority of the CASE statement for improving comprehension was clearly demonstrated for program 0. Not only was the function conveyed more quickly, but only one of the subjects was unable to discover it. The CASE was also superior in program 2 for the undergraduates, but the effect of experience apparently made the nested IF version the easiest to understand for the graduate students.

But why was the CASE version not superior in program 1? A possible explanation comes from the sequence-taxon theory of Sime, Green and Guest (SIME77). According to their theory, the process or program composition is the conversion of taxon information (the conditions under which an action is to be performed) into a combination of taxon and sequence information (the order in which the actions are to be performed). Conversely, comprehension is the conversion of the taxon and sequence information back into the taxonomic structure of the problem statement. Consequently, the version which most closely conveys the taxonomic information of the problem statement should be the easiest to understand. In the present experiment, the CASE statement -- which is highly taxonomic -- was superior in two of the three programs for the undergraduates. The possible reason why the CASE was not superior for program 1 was because the subjects had to infer some of the taxonomic information (i.e. X>=0 when X>Y, and

X>=0 when X<=Y). Had the conditions of the CASE included this redundant, but essential, information, the CASE statement might have also been superior for version 1.

To see how the other measures of comprehension fared, the cyclomatic measure and Gordon's measure were computed for the programs and are displayed in Table 6. As can be seen, the cyclomatic measure was not sufficiently sensitive to distinguish between versions. In contrast, Gordon's measure did correctly predict the ordering of versions in program 1, but overemphasized the difficulty of the CASE version in the other programs. If the sequence-taxon theory is correct, the redundant information which enhances comprehension would inflate Gordon's measure because of the additional operators and operands required to express the information. As a result, Gordon's measure would indicate that the CASE was less comprehensible than the other versions when it was actually more comprehensible.

1.5 IMPLICATIONS FOR TEACHING PROGRAMMING

The results of these experiments present practical implications for the teaching of programming. First, students need to be made aware that the manner in which expressions are stated in a program can affect its readability. They should strive to state an expression as simply as possible, aware that operators like OR can make the expression more difficult to understand. Obviously, many factors will have to be considered including the language being used and the possible alternate forms of the expression. But a realization that expression simplicity is an important factor in comprehensible software can help the student focus on areas in the program which need to be improved.

More importantly, the research conducted thus far indicates that the conditions under which an action is performed need to be stated clearly and explicitly. Often, this will require redundant coding of conditions as was demonstrated in this experiment. While redundant coding may be abhorrent to computer scientists who feel that terseness is a virtue, it should be emphasized that terseness may only be a virtue for the machine and not the programmer.

1.6 CONCLUSION

In this paper, a reaction time methodology was applied to the study of some factors affecting program comprehension. In general, the results indicate that operators differ in complexity and that redundant information in a program can enhance its comprehensibility. But more importantly, the study of comprehension has provided support for a theory of the programming process which is based on human subject data. It is appropriate that the object of research be the human rather than the machine since the important improvements in software will only occur when software tools are developed to accommodate those who use them.

BIBLIOGRAPHY

(BANKS76) Banks, William P., Fujii, Milton; and Kayra-Stuart, Fortunee. "Semantic Congruity Effects in Comparative Judgments of Magnitude of Digits." Journal of Experimental Psychology: Human Perception and Performance 2 (August 1976):435-447.

(BOYS79) Boysen, John P. "Factors affecting computer program comprehension." Ph.D. dissertation, Iowa State University, 1979.

(GORD75) Gordon, R.D. "A Measure of Mental Effort Related to Program Clarity." Ph.D. dissertation, Purdue University, 1975.

(GOUG65) Gough, Phillip B. "Grammatical Transformations and Speed of Understanding." Journal of Verbal Learning and Verbal Behavior 4 (April 1965):107-11.

(HALS77) Halstead, M.H. Elements of Software Science. New York: Elsevier North-Holland Inc., 1977.

(KELL77) Keller, Roy F. "On Control Constructs for Constructing Programs." SIGPLAN Notices 12 (September 1977): 36-44.

(KIRK68) Kirk, Roger E. Experimental Design: Procedures for the Behavioral Sciences. Belmont: Wadsworth Publishing Company Inc., 1968.

(LOVE77) Love, Tom. "An Experimental Investigation of the Effect of Program Structure on Program Understanding." ACM SIGPLAN Notices: Language Design for Reliable Software 12 (March 1977):105-13.

(MCCA76) McCabe, Thomas J. "A Complexity Measure." IEEE Transaction on Software Engineering SE-2 (December 1976): 308-20.

(SHNE77) Shneiderman, Ben. "Measuring Computer Program Quality and Comprehension." International Journal of Man-Machine Studies 9 (July 1977): 465-78.

(SIME77) Sime, M.E.; Green, T.R.G.; and Guest, D.J. "Scope Marking in Computer Conditions - A Psychological Evaluation." International Journal of Man-Machine Studies 9 (January 1977): 107-18.

(WEIN74) Weinberg, Gerald M.; and Schulman, Edward L. "Goals and Performance in Computer Programming." Human Factors 16 (February 1974): 70-7.

Appendix A

PROGRAM VERSIONS USED IN EXPERIMENT

```
------------------------Version 0---------------------------
DCL (W,X,Y,Z) FIXED DEC;
IF X>=Y
  THEN IF Y>=Z
         THEN W=Z;
         ELSE W=Y;
  ELSE IF X>=Z
         THEN W=Z;
         ELSE W=X;
PUT LIST(W);
------------------------VERSION 1--------------------------
DCL (W,X,Y,Z) FIXED DEC;
W=X;
IF Y<W
  THEN W=Y;
IF Z<W
  THEN W=Z;
PUT LIST(W);
------------------------VERSION 2--------------------------
DCL (W,X,Y,Z) FIXED DEC;
CASE
  X<Y AND X<Z DO; W=X; END;
  Y<X AND Y<Z DO; W=Y; END;
  ELSE        DO; W=Z; END;
ENDCASE
PUT LIST(W);
```

Figure 2: Program 0: Three Versions to Compute a Minimum

```
-----------------------VERSION 0--------------------------
DCL (X,Y) FIXED DEC;
      Y=0;
NEXT: GET LIST (X);
      IF X>=0
        THEN IF X>Y
               THEN Y=X;
               ELSE GOTO NEXT;
        ELSE GOTO FINISH;
      GOTO NEXT;
FINISH: PUT LIST(Y);
-----------------------VERSION 1--------------------------
DCL (X,Y) FIXED DEC;
      Y=0;
NEXT: GET LIST(X);
      IF X>Y
        THEN Y=X;
      IF X>=0
        THEN GOTO NEXT;
      PUT LIST(Y);
-----------------------VERSION 2--------------------------
DCL (X,Y) FIXED DEC;
      Y=0;
NEXT: GET LIST(X);
      CASE
        X>Y DO; Y=X; GOTO NEXT; END;
        X<0 DO; GOTO FINISH; END;
        ELSE DO; GOTO NEXT;  END;
      ENDCASE;
FINISH: PUT LIST(Y);
```

Figure 3: Program 1: Three Versions to Compute a Maximum

```
------------------------VERSION 0--------------------------
DCL (A(N),B(M),C(N+M)) FIXED DEC,
    (I,J,K) FIXED DEC;
 * ARRAYS A AND B ARE ASSUMED TO BE SORTED IN
   ASCENDING ORDER                                        */1

I=1; J=1; K=1;
DO WHILE(K<=N+M);
  IF I<=N
    THEN IF J<=M
           THEN IF A(I)<B(J)
                  THEN DO; C(K)=A(I); I=I+1; END;
                  ELSE DO; C(K)=B(J); J=J+1; END;
           ELSE DO; C(K)=A(I); I=I+1; END;
    ELSE DO; C(K)=B(J); J=J+1; END;
  K=K+1;
END;
------------------------VERSION 1--------------------------
I=1;J=1;K=1;
DO WHILE (K<=N+M);
  IF I>N
    THEN DO; C(K)=B(J); J=J+1; END;
    ELSE IF J>M
           THEN DO; C(K)=A(I); I=I+1; END;
           ELSE IF A(I)<B(J)
                  THEN DO; C(K)=A(I); I=I+1; END;
                  ELSE DO; C(K)=B(J); J=J+1; END;
  K=K+1;
END;
------------------------VERSION 2--------------------------
I=1;J=1;K=1;
DO WHILE (K<=N+M);
  CASE
    I<=N AND J<=M DO; IF A(I)<B(J)
                        THEN DO; C(K)=A(I); I=I+1; END;
                        ELSE DO; C(K)=B(J); J=J+1; END;
                  END;
    I>N           DO; C(K)=B(J); J=J+1; END;
    J>M           DO; C(K)=A(I); I=I+1; END;
  ENDCASE;
  K=K+1;
END;
```

1The declarations and comments were present for all versions of program 2.

Figure 4: Program 2: Three Versions to Merge Two Sorted Arrays