EXPLORATORY EXPERIMENTS IN PROGRAMMER BEHAVIOR

Ben Shneiderman

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 17

EXPLORATORY EXPERIMENTS IN PROGRAMMER BEHAVIOR

BEN SHNEIDERMAN

REVISED: JUNE, 1975

Exploratory Experiments in Programmer Behavior

Ben Shneiderman
Computer Science Department
Indiana University
Bloomington, Indiana 47401

PART I: INTRODUCTION

The literature and research in programming focuses heavily on machine related issues such as parsing ease, execution efficiency, and character sets, but deals only superficially with human factors issues. The thesis of this paper is that we can and must separate the machine related issues from the human factors issues and that we should apply the relevant techniques to each area. The utility of such a decomposition should be obvious in light of the recent discussions of modular design.

The isolation of human factors questions from implementation details permits us to exercise our imagination in the creation of new languages and allows us to pursue a more thorough study of the programming process by the experimental techniques developed by cognitive psychologists. Although programming is a more complex problem solving task than most tasks studied by cognitive psychologists, controlled psychological tests can be extremely helpful in providing new insights.

The immediate goals of such experimentation would be to compare specific programming language features such as control structures,

argument passing techniques, input/output facilities, and declaration statements. Other immediate goals would be to develop reliable standards for stylistic issues such as commenting, indentation, meaningful variable names and the complexity of modules.

Longer range goals include the development of an understanding of the intellectual skills relevant to programming, the creation of problem and program complexity measures, the production of programmer aptitude and ability tests and the improvement of programming education.

The first step in such a program or research must be the development of the experimental methodology. Unfortunately the history of experimentation on programmer behavior is relatively short (see 1 for a discussion with references). We should reiterate that we are, in this paper, interested in controlled experimental comparisons, not protocol analysis of individual programmer's introspective comments and not case studies of programmers in non-controlled environments. Both of these methods are useful, but the focus of this paper is on controlled experimental techniques.

Categorization of environments

The term "programming" refers to a wide variety of behaviors and environments. An experiment must focus precisely on a small number of variables and attempt to eliminate bias by keeping all other variables constant. The previous experience of subjects plays a crucial role since the variation in performance of individuals is enormous. Since we were using university undergraduates, graduates, and faculty, we separated our subjects according to the number of programming courses they had taken:

Naive	No programming courses
Novice	Currently enrolled in a first course in programming
Intermediate	Currently enrolled in a second or third programming course
Advanced	Graduate students and faculty

This crude separation is not always sufficient since we found tremendous variation within each group, but this scale is useful in roughly describing subjects.

A second relevant issue is the size of the problem or program that is being dealt with. Our initial work was largely with FORTRAN and we found the following scale useful in our discussions:

Small	less than 100 program statements
Medium	less than 1000 program statements
Large	less than 10,000 program statements
Very large	more then 10,000 program statements

Because of the limitations of our experimental environment, our first experiments were limited to small and medium sized programs, but we eventually hope to study larger programs. We hope that the work on short programs will be generalizeable to larger programs.

In addition to learning programming, we identified four relevant tasks which needed study:

Comprehension	Give subject a program and measure how well he/she understands the program
Composition	Give subject a problem and require a program to be written

Debugging	Give subject a problem with an incorrect program for that problem and require the subjects to locate the bugs
Modification	Give subject a correct program and require a modification

These tasks are interrelated since comprehension is necessary for degugging, composition is necessary during modification, etc. There can be wide variation in the scope and difficulty of each of these tasks, but this classification is helpful.

Measurement techniques

The relevant measurement techniques vary with the task assigned. Program comprehension can be measured by "fill-in-the-blank" or multiple choice questions which ask for

- the value of a variable at a specific point in the program
- the sequence of values assumed by a variable
- the number of times a particular statement is executed
- the sequence of statements executed
- the output of the program
- a brief description of the function of the program
- the impact of an alternaton

"Fill-in-the-blank" questions are more difficult to grade but multiple choice questions are often unrealistic.

Subjective measures such as asking the subject to estimate, on a 1 to 10 point scale, how well he/she understood the program are dubious but easy to capture. Time to completion can be helpful, but may be misleading. Time to criterion, that is, how long did it take the subject to correctly answer a question, is more appropriate.

A final measurement technique for comprehension is the ability to memorize. Since memorization of complex material is accomplished by absorbing meaningful "chunks" subjects must comprehend before memorizing. Memorization is not necessarily an aid to comprehension, but success at memorization is a measure of comprehension [2,3].

Measurement of the composition task consists of assigning a problem and requiring subjects to create a program which performs the required tasks. The most obvious way to do this is to require the subject to write the program on blank paper or appropriate coding sheets and then grade the final results. Unfortunately the grading process is subject to variation and careful attention must be given to establishing precise standards. Duplicate grading by different graders can produce more accurate results.

If the test environment permits, subjects may be required to keypunch their programs or enter them on an interactive timesharing system. The latter approach enables the experimenter to collect a complete history and accurate timing data. The programs produced by the subjects can then be executed, tested and debugged. Grading can include such factors as number of bugs, number of runs, time to completion, number of statements, execution efficiency, etc.

The test environment for composition can be simplified if subjects are required to write program fragments, only. This approach enables the experimenter to focus on particular language features and greatly reduces the time necessary. Another simplification would be to ask the subject to select, instead of compose, the correct program from a group of two or more possibilities.

Debugging can be studied experimentally by providing subjects with a problem description and an error laden program and requiring

the subjects to locate the errors. Supplementary aids such as flowcharts, program output or traces may be given to assist the subject. Multiple choice questions are probably unrealistic since they strongly direct the subjects' thought processes. Grading responses is, again, a difficult task: subjects often have insightful answers which point up other failures in the program or eliminate the bug in unorthodox ways. Modification is similar to debugging.

Research topics

Our research began with some simple experimental designs in which we hoped to develop our methodology. We chose experiments which were easy to conduct and had few variables. This paper reports on two experiments: a memorization task based on related work by Simon [4,5] and a comparison of the logical and arithmetic IF statements in FORTRAN. The first experiment was designed to give use some insight into the cognitive processes which occur during the study of a program, while the second dealt with a specific language feature in FORTRAN.

In more recent work [6] we have studied the effect of modularity on comprehension and debugging, the utility of comments and meaningful variable names and the ability of naive subjects to learn simple database query languages. In another series of experiments [7] we studied the usefulness of detailed standard flowcharts on comprehension, composition, debugging and modification.

These experiments have given us new insights into the cognitive processes in programming and a cognitive model has been proposed to explain our findings [6]. Our future work will be directed at the verification of this model and to the specific goals mentioned earlier.

For an alternate discussion of research topics and methodology see the work of Weissman [8,9], Gannon and Horning [10], and Miller [11,12].

PART II: EXPERIMENTS

Memorization

This experiment compared the abilities of subjects to memorize two sequences of FORTRAN statements. One was a proper executable program while the other contained valid statements in scrambled order. The experiment, involving subjects of varying experience, measured short-term memory capacity in an attempt to correlate program structure and ease of memorization. Our results indicate that this correlation exists at a statistically significant level, leading useto the conclusion that the structure of a program facilitates comprehension and memorization. The test items were approximately twenty statements long, but we have not shown that this is the upper limit of module size for human comprehension. This question will be the subject of further experimentation.

Method

Subjects and Design: The subjects were selected from four experience groups. Group I contained forty-two people who had no previous experience with FORTRAN. These tests were given to them on the first day of their introductory FORTRAN class. Group II consisted of ten people, and the test was given at the end of their introductory FORTRAN course. Group III consisted of eighteen subjects who had had an introductory FORTRAN course and were in the process of taking more advanced computer science courses: assembly language, data structures,

or programming languages. Group IV consisted of nine members: graduate students and faculty members of the computer science department who were considered to have had extensive programming experience.

Subgroups of each group were tested at various occasions for testing convenience during the summer of 1974. All subjects received both the proper executable and the scrambled program test item. They were asked to do one test at a time and record the sequence.

Materials: Two FORTRAN program listings were used as test items Program A was a proper executable program (Figure 1). Program B was a set of statements of a randomly shuffled FORTRAN program (Figure 2). Program A consisted of 20 lines of code, and Program B had 17 lines. (These programs were taken from Organick [13], pp. 86-87.)

Procedure: Each group of subjects was run in subgroups for testing convenience. Sometimes a subgroup consisted of only one subject. Subjects at each session were evenly divided on a random basis into two groups. The first group did Program A first, and the second group did Program B first. Computer printed listings of each program one per page, were handed out at the beginning of each testing session. After each subject received Program A or Program B, the following instructions were given orally:

1. This is a memory test consisting of two parts.
2. Do not turn the page over until you are told to do so. (The subjects received the test face down.)
3. Print your name on the upper right-hand corner of the page

and write a 1 (or a 2 during the second part) next to your name.

4. You will have three minutes to memorize the material and four minutes to rewrite what you have memorized on a second sheet of paper.
5. To get full credit on the line you copied back, you must write the line exactly as it appeared in the listing, i.e., 10 is not the same as 10.0.

The number next to the subject's name, requested in instruction three, was used for the purpose of identifying the order in which the test items were taken. Immediately after the three-minute memorization period was up, each sub-ect was given four minutes to copy back what he/she had memorized. Part two of the test, the alternate test item, was given immediately after the four-minute period and the same instructions were followed. After the session was over, the test papers were collected and graded. The papers were graded by the number of correct lines. A line was considered correct if it was identical to the original and the relative position was approximately right.

Results

The results of this experiment can be summed up in Table 1.

Experimental Group	Number Correct	
	A	B
I	7.1	4.1
II	10.2	4.6
III	12.7	5.4
IV	17.3	6.4

Table I: Mean number and percentage of correct lines

```
      XSMALL = 0.0
      NSMALL = 0
      XLARGE = 0.0
      NLARGE = 0
      READ(5,82) TEST
   82 FORMAT(8F10.0)
      DO 10 I = 1,40
      READ(5,82) X
      IF(X .GT. TEST) GO TO 5
      XSMALL = XSMALL + X
      NSMALL = NSMALL + 1
      GO TO 10
    5 CONTINUE
      XLARGE = XLARGE + X
      NLARGE = NLARGE + 1
   10 CONTINUE
      WRITE(6,814) NSMALL, XSMALL, NLARGE, XLARGE
  814 FORMAT(I10, F15.3, I10, F15.3)
      STOP
      END
```

Figure 1. Program A

```
      READ(5,81) NEMP
      PAY = RATE*HOURS
      IF(HOURS .LE. 40.0) GO TO 5
    5 CONTINUE
   82 FORMAT(8F10.0)
      PAY = PAY + 0.5 * RATE * (HOURS - 40.0)
      WRITE(6,85) I, PAY
      END
      N = N + 1
   10 CONTINUE
      WRITE(6,81) N
      STOP
   81 FORMAT(8I10)
      N = 0
      DO 10 I = 1,NEMP
   85 FORMAT(I10, F15.2)
      RFAD(5,82) RATE, HOURS
```

Figure 2. Program B

Regardless of experience, subjects did equally well on Program B, but with Program A, the ability to memorize increased with programming experience. We predicted that the order of the test items would influence the subjects' performance, since

a. If the subject received Program A first, then he/she might try to spend most of his/her time organizing Program B rather than memorizing it.
b. If Program B were received first, he/she might not try to memorize Program A in an organized fashion.
c. It might take a while for the subjects to gear their intellectual activity to memorizing.

This experimental bias was observed, and it was dealt with by dividing the subjects in each session into two equal subgroups. One subgroup received Program A first, the other Program B first.

An analysis of variance indicated that our groups representing different levels of programming experience were significantly different at the 0.001 level. The interaction between groups and type of program was also significant at the 0.001 level.

Discussion

When a complex problem is encountered, subjects attempt to tackle it by dividing it into parts. In the case of memorizing test items, human intellectual powers are rather geared to conceptually "group" as much information as possible which, in turn, eases the burden of memorizing. This phenomenon is known to psychologists as "chunking". Chunking is a recoding process that human beings seem to do without conscious effort. This process involves grouping or organizing the

input information into "chunks", which are as easy to handle as individual units. For example, to an experienced programmer, Program A reads something like: initialize four variables in the first four lines. The fifth line sets a testing variable, then a group of instructions are executed forty times. This group of instructions consists of inputing a test value and the input value; two variables are set. At the end of forty iterations, the results of the comparisons are printed. Depending on the experience of the programmer, that group of instructions can further be recognized as a comparison with the testing value followed by one instruction which keeps a running sum and another which keeps track of the number of occurrences.

The ability to reorganize the test item is more prominent in experienced programmers. To the non-programmer, FORTRAN is totally foreign, and each line or even each token is memorized independently. The existence of this phenomenon is supported by our experiment, since the number and percentage of correct lines for Program A increases substantially with the level of experience, while the gain for Program B is only modest.

Studying Program A we find a few similar statements, such as the first four statements, which lessened the burden of memorization, even for non-programmers. All subjects seem to have remembered these four statements as a unit, a chunk. This explains, in part, why novice programmers did better on Program A, since Program B did not contain a similar simplification.

With Program B, chunking cannot be applied easily since the subjects have no basis on which to chunk the information, regardless of experience. The slightly better performance of experienced subjects

on Program B can be attributed to the fact that they are familiar with FORTRAN and remembered each FORTRAN statement as a unit rather than each token, as non-programmers would do.

The fact that the average number of correctly memorized statements for Program B is five or six brings to mind the well-known paper by George Miller, "The Magical Number Seven, Plus or Minus Two; Some Limits on Our Capacity for Processing Information" [14], which indicates that the short term memory of humans is seven units plus or minus two. A detailed psychological analysis of the dimensions of our problem of information transfer is complex, but if one accepts a line of FORTRAN code as a unit of information transfer, our result is well within the limit of Miller's seven, plus or minus two.

Returning to our results for Program A, we conclude that experienced subjects have developed an ability to encode the program in chunks whose size is larger than one statement. Experienced programmers can deal with sophisticated control structures such as the DO-WHILE, IF-THEN-ELSE, etc., as a single unit and thus can comprehend and memorize substantially longer sections of code. These conceptual control structures can be recognized by experienced programmers even when they are implemented in the limited syntactic forms provided by FORTRAN. Further research in this area could lead to practical measures of the limits of intellectual manageability and recommendations as to the optimum complexity of program modules for differing experience levels. Recognizing that there exists a close relationship between comprehension, memorization, and chunking, the memory experiment provides a practical means for conducting this research.

Replication

In response to several criticisms of the first memorization experiment, we performed a simplified version in early 1975. Critics of our first experiment suggested that there was a poteneial bias from the fact that we had used different programs of slightly different length.

A more complex 74 line FORTRAN program was created in proper executable and shuffled form. Sixteen non-programmers and sixteen experienced graduate students were used as subjects. Eight subjects in each group received the proper executable program and the remaining eight subjects in each group received the shuffled program. Subjects were given fifteen minutes to memorize and five minutes to write.

The non-programmer subjects averaged 13.13 correct lines in the proper executable program and 10.1 correct lines in the shuffled program. This difference was not statistically significant and supports the contention that non-programmers could not chunk the proper executable program. The experienced subjects memorized 24.8 lines of the proper executable program but only 18.9 lines of the shuffled program. This difference was statistically significant at the .025 level, thus supporting the chunking idea.

Although these statistics support our contention, we had hoped for stronger results. Apparently the subjects focused on the multiplicity of CONTINUE statements and other highly similar repetitive statements and achieved high scores by writing these statements rather than attempting to proceed sequentially. This enabled them to do relatively well on the shuffled form of the program.

We look forward to testing the capacity of our subjects and to determine what structures are more difficult to memorize. Hopefully, we will be able to isolate the structural patterns that the subjects perceive and study which programming languages provide the best facility for representing these structures clearly.

It has been suggested that a retest after a day or a week (without showing the program again) would allow us to determine what aspects of the program remained most prominent in the subject's mind.

The memorization technique might also enable comparative testing of the usefulness of mnemonic variables, programming languages features or other stylistic features.

Conditional Branching

The two conditional branching techniques in FORTRAN are the arithmetic IF statement and the logical IF statement. The logical IF statement is not included in the ANSI Standard Basic FORTRAN [15], but it is more frequently used than the ANSI standard arithmetic IF statement. An examination of contemporary FORTRAN textbooks reveals that although the arithmetic IF statement is frequently introduced before the logical IF statement, most authors place more emphasis on the logical IF statement. McCracken, in his recent book, A Simplified Guide to FORTRAN Programming [16], takes the extreme position that "The arithmetic IF has little use in well-constructed programs... Heavy use of the arithmetic IF leads to intricate programs that are very hard to read and understand". At many educational institutions, the use of arithmetic IF statements is either not taught or is discouraged. Is this unfair treatment to the arithmetic IF statement justified? Our experiment is an attempt to provide experimental results and guidance for educators and language designers.

Method

Subjects and Design: Forty-eight subjects were recruited in the summer of 1974 for this study. Twenty-four of them (Group I) were students who were in the process of taking an introductory FORTRAN class. Their instructor covered both arithmetic and logical IF statements, with equal emphasis. The remaining twenty-four subjects (Group II) were students who had completed an introductory FORTRAN course and all were in the process of taking a more advanced computer science course. Graduate students and faculty members were included in Group II. Within each group, two types of test items were given: arithmetic IF and logical IF. Within each type, three levels of difficulty of the test items were given. The difficulty ratings were based on the subjective judgement of the authors. The test items were given to the subjects of Group I during one of their regular classroom sessions. Subjects of Group II were tested in four sessions for testing convenience.

Materials: The test items were given in two parts. Part 1 consisted of three FORTRAN programs: LHARD (hard), LEASY (easy), and LMOD (moderately hard). Only logical IF statements were used in this part. Part 2 also consisted of three FORTRAN programs, but only arithmetic IF statements were used. Part 2 consisted of programs AHARD (hard), AEASY (easy), and AMOD (moderately hard). Associated with each program were questions that tested the subjects' understanding of the program. This was accomplished by asking the subjects to follow the execution sequence and to reproduce the output of the program. There were four fill-in type questions associated with programs LEASY, AEASY, LMOD, and AMOD. There were ten questions (six fill-in type and four multiple choice) associated with program LHARD.

Programs LEASY and AEASY consisted of nine lines each and were similar. The questions associated with each of them were simple and similar. They tested the subjects' ability to follow logical/arithmetic IFs in a simple branching sequence.

Programs LMOD and AMOD were similar programs consisting of nine and ten lines, respectively. Each program contained an "IF-loop". Program LMOD checked the termination of the loop at the exit, and Program AMOD checked at the entrance to the loop. The questions associated with each of these two programs tested the subjects' ability to follow the execution sequence and understand the role that the IF statement played in each program. Subjects were required to reproduce the output generated by a PRINT statement at the termination of the loop. All programs and questions were in computer output form. Space was provided for a subjective measure of difficulty of each question and for timing data for each program.

The test papers were collected at the end of each test session and graded on the number of wrong answers that each subject made. Additional materials used in the experiment included a general survey of the subjects' computer science background. A lesson was also given to some of the subjects of Group II a few weeks before the testing session to ensure their knowledge of both arithmetic and logical IF statements.

Procedure: The survey on the computer science background of the subjects was distributed and the subjects were asked to respond to the questions. Half the subjects in each testing group received a test booklet containing Part 1 first while the other half received Part 2 first. The following instructions were on the front of the test booklets:

> Please wait for the signal then you may start answering questions appearing on the following pages. It is important that you work sequentially and do not go back and change your answer. Use the spaces to the right of the questions to give any comment that you care to make. Please indicate how difficult you find each of the questions by marking the boxes to the left of each question. Use the digits 1 through 10 to indicate difficulty: 10 is most difficult; 1 is easiest.

Students worked at their own pace and turned in the test booklets when they were done. They were told to mark the clock time to the nearest minute at the end of each section.

Results and Discussion

The test papers were graded on the basis of the number of errors the subject made for each program. The average is shown in the following table:

Experience Groups	Part 1 LOGICAL IF			Part 2 ARITHMETIC IF		
	LHARD	LMOD	LEASY	AHARD	AMOD	AEASY
I	1.71	2.13	0.83	3.38	1.63	1.25
II	1.05	1.45	0.62	1.50	1.15	1.65

Table II: Average number of errors

Experience Groups	Part 1 LOGICAL IF			Part 2 ARITHMETIC IF		
	LHARD	LMOD	LEASY	AHARD	AMOD	AEASY
I	17.7	53.3	20.8	37.6	40.8	31.3
II	10.5	36.3	15.5	16.7	28.7	16.3

Table III: Average percentage of errors

An analysis of variance was performed on the error data. As expected, the experienced subjects did statistically significantly better at the 0.01 level than inexperienced subjects. The two-way interaction of the difference between the two groups and the two types of test items was marginally significant. This gives mild support to the conjecture that logical IF statements tend to be easier for beginners, while for an experienced programmer, logical IF statements and arithmetic IF statements are equally difficult.

The difficulty of the programs, in the authors' opinion, is in this increasing order: LEASY and AEASY, LMOD and AMOD, then LHARD and AHARD. The analysis of variance yields a significant level of interaction (0.05) of groups by difficulty. However, according to Table II the majority of the subjects, regardless of the group, made errors in questions associated with programs LMOD and AMOD (the IF-loop problems). Their mistakes tended to be the usual "off-by-one" type of error. To avoid this type of error, a "DO-loop" should be encouraged in preference to an "IF-loop". A further possibility in parts LMOD and AMOD is that subjects found it more difficult to understand loop termination tests at the bottom of the loop.

Unfortunately, some of the subjects did not give the information for timing of each program and/or hardness of each question. Out of the limited timing information we have, only eight out of nineteen subjects took more time to finish Part 1 in Group I. Of the sixteen responding in Group II, exactly half took more time on Part 1. These results are inconclusive and do not support or detract from our conclusions. The average time for those reporting is shown on the following table:

Experience	Part 1 (logical IF)	Part 2 (arithmetic IF)
I	9.60	11.35
II	11.88	10.50

Table IV: Average time

The difficulty rating of each question is a subjective measure. Of the eleven subjects in Group I who responded, only two rated Part 1 as more difficult than Part 2; of the seventeen subjects in Group II who responded, seven rated Part 1 as more difficult than Part 2. The fact that the majority of the novice programmers considered the questions with logical IF statements to be easier, but almost a half of the experienced programmers rated the problem with arithmetic IF statements to be easier, further supports the claim that the logical IF statement is easier for novice programmers and that the logical IF statement and arithmetic IF statement are equally difficult for the experienced programmers.

We conjecture that the experienced programmers recode the syntactic form in their minds and deal with a higher level semantic notion of what the program actually does. The novices are constrained to deal with the raw syntactic inputs and have greater difficulty with the complex details of the arithmetic IF statements. This effect should be even more dramatic with longer and more complicated programs.

Further Research

The experimental controls in this experiment could be substantially improved if the same program, coded using the arithmetic or the logical IF were presented to independent subgroups, rather than

having each subject act as his/her own control. Longer and more complex programs would shed further light on the usefulness of the two branching constructs for novice and experienced programmers. The influence of the logical and arithmetic IF statements will also be studied on the tasks of program composition, debugging, and modification.

In future experiments timing data will have to be collected in a way that does not interfere with the subject's test-taking. We are unsure as to the usefulness of subjective measures, but we will pursue this approach as well.

Interpretation

These experiments have enabled us to improve our methodology and have demonstrated the need for further study of the cognitive processes that occur when subjects examine computer programs. Both experiments suggested to us that experienced programmers recode the syntactic forms into an internal structure which represents the semantic structure of the program. This analysis is informally supported by the fact that in the memory experiment experienced subjects would reproduce a semantically correct program with syntactic mistakes. These mistakes were such items as altered statement labels or sequencing changes which did not affect the output of the program.

We hypothesize that the internal recoding process also accounts for the fact that experienced subjects found little difference in the ease of comprehension of the arithmetic and logical IF statements. [7].

Further study of the recoding process might lead to an understanding of what the chunks consist of and to suggestions for improved

language designs, stylistic guidelines, and recommendations for design strategies. The influence of comment cards, indentation, meaningful variable names, etc., must all be interpreted with respect to their effect on this recoding process.

Acknowledgement

We would like to express out sincere thanks to Richard Mayer for enlightening discussions concerning the data analysis and to Jim Carlisle for his detailed review which substantially improved the manuscript. The suggestions of the referees were helpful in broadening the scope of this paper and placing the experiments in the proper context.

Finally, this work could not be done without the diligence of the graduate students who carried out the experiments. Mao-Hsian Ho did the basic memory and conditional experiments while Ken Yasukawa replicated the memory experiment.

References

1. Shneiderman, Ben. Experimental testing in programming languages, stylistic considerations and design techniques. Proc. National Computer Conference, AFIPS Press, Montvale, NJ (1975).

2. Bransford, John D., and Franks, Jeffery J. The abstraction of linguistic ideas. Cognitive Psychology 2 (1971), 331-350.

3. Barclay, Richard J. The role of comprehension in remembering sentences. Cognitive Psychology 4 (1973), 229-254.

4. Chase, William G., and Simon, Herbert A. Perception in Chess. Cognitive Psychology 4 (1973), 55-81.

5. Simon, Herbert A., and Gilmartin, Kevin. A simulation of memory for chess positions. Cognitive Psychology 5 (1973), 29-46.

6. Shneiderman, Ben, and Mayer, Richard. Towards a cognitive model of programmer behavior (in progress).

7. Shneiderman, Ben; McKay, D.; and Heller, P. Experimental studies of flowcharts in programming (in progress).

8. Weissman, L. Psychological complexity of computer programs: an initial experiment. Technical Report CSRG-26, Computer Systems Research Group, University of Toronto, Toronto, Canada (1973).

9. ----------. A methodology for studying the psychological complexity of computer programs. Ph.D. Thesis, University of Toronto (1974). Available as Technical Report, Computer Science Research Group CSRG-37.

10. Gannon, J.D., and Horning, J.J. The impact of language design on the production of reliable software. Proc. 1975 International Conference on Reliable Software.

11. Miller, Lance. Programming by non-programmers. IBM Research Report RC 4280 (1973).

12. ----------. Naive programmer problems with specification of transfer-of-control. Proc. National Computer Conference, AFIPS Press, Montvale, NJ (1975).

13. Organick, E.I., and Meissner, L.P. FORTRAN IV, 2nd Edition, Addison-Wesley Publishing Co. (1974).

14. Miller, G.A. The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychological Review 63 (1956), 81-97.

15. U.S.A. Standard Basic FORTRAN, ANSI Standard X3.10, American National Standards Institute, New York (1966).

16. McCracken, D.D. A Simplified Guide to FORTRAN Programming, John Wiley & Sons, Inc. (1974).

Typed by Christopher Charles

Appendix 1: Logical IF Program.

LHARD

```
      PROGRAM A(INPUT,OUTPUT)                                  LINE    1
      READ 60,I,J,K                                            LINE    2
      IF(I.GT.J)GO TO 20                                       LINE    3
      IF(J.GT.K)GO TO 10                                       LINE    4
      IL=K                                                     LINE    5
      GO TO 40                                                 LINE    6
   10 IL=J                                                     LINE    7
      GO TO 40                                                 LINE    8
   20 IF(I.GT.K)GO TO 30                                       LINE    9
      IL=K                                                     LINE   10
      GO TO 40                                                 LINE   11
   30 IL=I                                                     LINE   12
   40 PRINT 50,IL                                              LINE   13
   50 FORMAT(3X,I3)                                            LINE   14
   60 FORMAT(3I3)                                              LINE   15
      END                                                      LINE   16
```

LEASY

```
      PROGRAM C(INPUT,OUTPUT)                                  LINE    1
      IA=9                                                     LINE    2
      IB=34                                                    LINE    3
      IF(IB .GT. IA) GO TO 10                                  LINE    4
      PRINT,IB                                                 LINE    5
      GO TO 20                                                 LINE    6
   10 PRINT, IA                                                LINE    7
   20 STOP                                                     LINE    8
      END                                                      LINE    9
```

LMOD

```
      PROGRAM E(INPUT,OUTPUT)                                  LINE    1
      IJ=5                                                     LINE    2
      IX=4                                                     LINE    3
   10 IJ=IJ-1                                                  LINE    4
      IX=IX+1                                                  LINE    5
      IF(IJ .GE. 0) GO TO 10                                   LINE    6
      PRINT, IX                                                LINE    7
      STOP                                                     LINE    8
      END                                                      LINE    9
```

Appendix 2: Arithmetic IF Program.

AHARD

```
      PROGRAM B(INPUT,OUTPUT)                                  LINE    1
      READ 80,I,J,K                                            LINE    2
      IF(I-J)40,10,10                                          LINE    3
   10 IF(J-K)30,20,20                                          LINE    4
   20 IS=K                                                     LINE    5
      GO TO 70                                                 LINE    6
   30 IS=J                                                     LINE    7
      GO TO 70                                                 LINE    8
   40 IF(I-K)60,50,50                                          LINE    9
   50 IS=K                                                     LINE   10
      GO TO 70                                                 LINE   11
   60 IS=I                                                     LINE   12
   70 PRINT 90,IS                                              LINE   13
   80 FORMAT(3I3)                                              LINE   14
   90 FORMAT(3X,I3)                                            LINE   15
      END                                                      LINE   16
```

AEASY

```
      PROGRAM D(INPUT,OUTPUT)                                  LINE    1
      IM=27                                                    LINE    2
      IL=21                                                    LINE    3
      IF(IM-IL)10,20,30                                        LINE    4
   10 PRINT, IM                                                LINE    5
   20 STOP                                                     LINE    6
   30 PRINT, IL                                                LINE    7
      STOP                                                     LINE    8
      END                                                      LINE    9
```

AMOD

```
      PROGRAM F(INPUT,OUTPUT)                                  LINE    1
      IL=7                                                     LINE    2
      IT=0                                                     LINE    3
    5 IF(IT-7) 10,20,30                                        LINE    4
   10 IT=IT+1                                                  LINE    5
      IL=IL-1                                                  LINE    6
   30 GO TO 5                                                  LINE    7
   20 PRINT, IT,IL                                             LINE    8
      STOP                                                     LINE    9
      END                                                      LINE   10
```

Appendix 3: Questions

LHARD

1. FOR I=2, J=5, AND K=3, WHAT IS THE OUTPUT?

2. FOR I=3, J=7, AND K=7, WHAT IS THE OUTPUT?

3. IN ONE RUN OF THIS PROGRAM, HOW MANY TIMES DOES LINE 13 GET EXECUTED?

4. WHAT IS THE MOST NUMBER OF TIMES THAT THIS PROGRAM MAKES AN 'IF' TEXT?

5. WHAT IS THE LEAST NUMBER OF TIMES THAT THIS PROGRAM MAKES AN 'IF' TEST?

6. CAN YOU DESCRIBE WHAT THIS PROGRAM DOES?

7. IF THREE INPUT VALUES ARE EQUAL, WHICH VALUE DOES THIS PROGRAM PRINT OUT?

A. I B. J C. K D. NONE OF THE ABOVE

8. IF I=K AND I ALSO IS GREATER THAN J, WHICH VALUE DOES IT PRINT OUT?

A. I B. J C. K D. NONE OF THE ABOVE

9. IF ALL .GT.'S WERE CHANGED TO .GE. IN THIS PROGRAM,

A. THE ORIGINAL PURPOSE OF THIS PROGRAM DOES NOT GET ALTERED.

B. THE NEW PROGRAM DOES THE OPPOSITE OF WHAT THE ORIGINAL PROGRAM DOES.

C. THE NEW PROGRAM OUTPUTS GARBAGE.

D. NONE OF THE ABOVE.

E. DO NOT KNOW

10. HOW COULD ONE MODIFY THIS PROGRAM SO THAT IT PRINTS OUT THE MINIMUM OF THREE INPUT NUMBERS?

A. REPLACE ALL .GT.'S WITH .LT.'S.

B. INTERCHANGE I,J; J,K; AND I,K IN LINES 3, 4, AND 9, RESPECTIVELY.

C. IN ADDITION TO B ABOVE, REPLACE K WITH I, J WITH K, K WITH I, AND I WITH K IN LINES 5, 7, 10, AND 12, RESPECTIVELY.

D. NO MODIFICATIONS NEEDED.

E. NONE OF THE ABOVE.

F. DON'T KNOW.

LEASY

1. WHAT IS THE OUTPUT OF THIS PROGRAM?

2. PLEASE WRITE DOWN THE SEQUENCE OF LINE NUMBERS THAT ARE EXECUTED.

3. IF .GT. IN LINE 4 WERE CHANGED TO .LT., WHAT WOULD BE THE OUTPUT?

4. AFTER THE CHANGE TO .LT., IA=34 AND IB=9, WHAT IS THE OUTPUT OF THE NEW PROGRAM?

LMOD

1. WHAT IS THE OUTPUT OF THIS PROGRAM.

2. PLEASE WRITE DOEN THE SEQUENCE OF LINE NUMBERS THAT ARE EXECUTED.

3. IF .GE. IN LINE 6 WERE CHANGED TO .LT., WHAT WOULD BE THE NEW EXECUTION SEQUENCE?

4. WHAT WOULD BE THE NEW OUTPUT?

AHARD

1. FOR I=2, J=5, AND K=3, WHAT IS THE OUTPUT?

2. FOR I=3, J=7, AND K=7, WHAT IS THE OUTPUT?

3. IN ONE RUN OF THIS PROGRAM, HOW MANY TIMES DOES LINE 13 GET EXECUTED?

4. WHAT IS THE MOST NUMBER OF TIMES THAT THIS PROGRAM MAKES AN 'IF' TEST?

5. WHAT IS THE LEAST NUMBER OF TIMES THAT THIS PROGRAM MAKES AN 'IF' TEST?

6. CAN YOU DESCRIBE WHAT THIS PROGRAM DOES?

PLEASE CIRCLE THE BEST ANSWER FOR THE FOLLOWING QUESTIONS:

7. IF THREE INPUT VALUES ARE EQUAL, WHICH VALUE DOES THIS PROGRAM PRINT OUT?

A. I B. J C. K D. NONE OF THE ABOVE

8. IF I=K AND I ALSO IS GREATER THAN J, WHAT IS THE OUTPUT?

A. I B. J C. K D. NONE OF THE ABOVE

9. HOW COULD ONE MODIFY THIS PROGRAM SO THAT IT PRINTS OUT THE MINIMUM OF THREE INPUT NUMBERS?

A. INTERCHANGE I, J, J,K, AND I,K IN LINES 3, 4, AND 9, RESPECTIVELY.

B. IN ADDITION TO B ABOVE, REPLACE K WITH I, J WITH K, K WITH I, AND I WITH K IN LINES 5, 7, 10, AND 12.

C. NO MODIFICATIONS NEEDED.

D. NONE OF THE ABOVE.

E. DON'T KNOW.

AEASY

1. WHAT IS THE OUTPUT OF THIS PROGRAM?

2. PLEASE WRITE DOWN THE SEQUENCE OF LINE NUMBERS THAT ARE EXECUTED.

3. IF THE MINUS SIGN IN LINE 4 WERE CHANGED TO A PLUS, WHAT WOULD BE THE OUTPUT?

4. AFTER THE CHANGE TO PLUS, IF IM=21 AND IL=27, WHAT IS THE OUTPUT OF THE NEW PROGRAM?

AMOD

1. WHAT IS THE OUTPUT OF THIS PROGRAM?

2. HOW MANY TIMES DOES LINE 6 GET EXECUTED?

3. IF THE '-' IN LINE 4 WERE CHANGED TO '+', WHAT WOULD BE THE NEW EXECUTION SEQUENCE?

4. WHAT WOULD BE THE NEW OUTPUT?