AFOSR - TR - 71 - 2791

Report No. 2187
Job No. 11431

INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE

FINAL REPORT, SECTION 3
Task 3: PROGRAMMING LANGUAGES AS A TOOL FOR COGNITIVE RESEARCH

30 June 1971

ARPA ORDER NO. 890, Amendment No. 5

Sponsored by the Advanced Research Projects Agency, Department of Defense, under Air Force Office of Scientific Research Contract F44620-67-C-0033

Prepared for:

Air Force Office of Scientific Research
1400 Wilson Boulevard
Arlington, Virginia 22209

136

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3 REPORT TITLE

INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE TASK 3: PROGRAMMING LANGUAGES AS A TOOL FOR COGNITIVE RESEARCH

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Final

5 AUTHOR(S) (First name, middle initial, last name)

Wallace Feurzeig and George Lukas

6 REPORT DATE

30 June 1971

7a. TOTAL NO. OF PAGES

134

7b. NO. OF REFS

5 + (9 Appendix)

8a. CONTRACT OR GRANT NO F44620-67-C-0033

b. PROJECT NO 890

c. 61101D

d. 681313

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

AFOSR - TR - 71-2791

10 DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11 SUPPLEMENTARY NOTES

TECH, OTHER

12. SPONSORING MILITARY ACTIVITY

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (NL)
1400 WILSON BLVD
ARLINGTON, VIRGINIA 22209

13 ABSTRACT

Through study and analysis of data from previous teaching, several linguistic and conceptual difficulties in the way of acquiring the skills of problem-solving were identified. The LOGO programming language was taught to a group of teachers to explore its use as the basis for a course on mathematical problem-solving. LOGO-based courses in problem-solving were given to two groups of students with well-established difficulties in formal and mathematical work. Based on these teaching experiments, LOGO teaching sequences for an introductory problem-solving course were developed. An experiment was carried out to evaluate the validity of standard test measurements of achievement level. Programs were developed for monitoring, recording, and displaying students' problem-solving interactions with LOGO. A remote LOGO-controlled vehicle was developed to assist students in conceptualizing formal problem-solving tasks in a concrete context.

DD FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification

Report No. 2187 Bolt Beranek and Newman Inc.

INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE

FINAL REPORT, SECTION 3
Task 3: PROGRAMMING LANGUAGES AS A TOOL FOR COGNITIVE RESEARCH

30 June 1971

by

Wallace Feurzeig
and
George Lukas

ARPA Order No. 890, Amendment No. 5
Sponsored by the Advanced Research Projects Agency,
Department of Defense, under Air Force Office of
Scientific Research Contract F44620-67-C-0033

Prepared for

Air Force Office of Scientific Research
1400 Wilson Boulevard
Arlingtcn, Virginia 22209

TABLE OF CONTENTS

FINAL TECHNICAL REPORT

ARPA Order No. 890, Amendment No. 5
Program Code No. 9D20
Contractor: Bolt Beranek and Newman Inc.
Effective Date of Contract: 1 November 1966
Contract Expiration Date: 30 June 1971
Amount of Contract: $804,896.00
Contract No. F44620-67-C-0033
Principal Investigators: John A. Swets
Mario C. Grignetti
Wallace Feurzeig
M. Ross Quillian
Telephone No. 617-491-1850
Title: INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE

TASK 3: PROGRAMMING LANGUAGES AS A TOOL FOR COGNITIVE RESEARCH

1. Technical Problem

This task is an investigation of the use of programming languages as a means of studying and overcoming difficulties in solving formal problems.

2. General Methodology

Our method of investigation is by teaching experiments of the following kind. Trainee-subjects are taught the use of an appropriate programming language, LOGO, as a tool for problem-solving work. Their specific difficulties in learning and applying LOGO in various problem-solving tasks is studied and evaluated.

3. Technical Results

Through study and analysis of data from previous teaching, several linguistic and conceptual difficulties in the way of acquiring the skills of problem-solving were identified. The LOGO programming language was taught to a group of teachers to explore its use as the basis for a course on mathematical problem-solving. LOGO-based courses in problem-solving were given to two groups of students with well-established difficulties in formal and mathematical work. Based on these teaching experiments, LOGO teaching sequences for an introductory problem-solving course were developed. An experiment was carried out to evaluate the validity of standard test measurements of achievement level. Programs were developed for monitoring, recording, and displaying

students' problem-solving interactions with LOGO. A remote LOGO-controlled vehicle was developed to assist students in conceptualizing formal problem-solving tasks in a concrete context.

4. Department of Defense Implications

One area of direct application is that of teaching basic academic subjects and skills in military dependent schools. Problem-solving skills are important, not only in direct application to formal work in mathematics and military science, but also in less formal areas of problem-solving such as are encountered in military operational planning and decision-making.

5. Implications for Further Research

We expect the use of programming languages such as LOGO will make important contributions to both the theory and practice of education. Possible directions for further work are: (1) the use of programming languages as the operational framework for experimental studies on cognitive development in children, (2) the development of programming as a core subject for the mathematics curriculum, and (3) the LOGO program-controlled robot as a new framework for studying interactive man-machine systems. With appropriate sensors and effectors, such systems may provide useful operational applications.

1. PREFACE

At its inception in 1966, this contract was devoted solely to the one area of second-language learning. Later amendments have added three more tasks: Models of Man-Computer Interaction; Programming Languages as a Tool for Cognitive Research; and Studies of Human Memory and Language Processing. The present contract was scheduled for termination on 31 December 1970, but the final reporting date was changed to 30 June 1971, to allow completion of data analysis in the various tasks.

Due to the amount of information to be presented in the Final Report, we have bound it in four Sections, one for each task. In addition to a copy of this page, each Section contains an appropriate subset of the documentation data required for the report: a contract-information page, a summary sheet for the particular task at hand, and a DD form 1473 for document control.

2. INTRODUCTION

This report describes research investigating the teaching of programming languages as a means of studying problem-solving. The work utilized a new programming language, LOGO, expressly designed for teaching mathematical thinking and problem-solving. In this section we discuss the connection between programming and problem-solving and we give a brief description of our principal tool, LOGO.

The research was carried out in the context of three teaching experiments involving subjects over a range of age, aptitude, and achievement levels. The main result of the teaching was the development of an introductory course in problem-solving. The teaching experiments and the course are described in Sections 3 and 4. The work also generated two new tools for studying problem-solving interactions. These are described in Section 5.

Several persons participated in these efforts. Wallace Feurzeig designed and coordinated the program. The three teaching experiments were conducted by Seymour Papert and Cynthia Solomon; Wallace Feurzeig; and George Lukas. Walter B. Weiner performed the system programming required to incorporate "dribble files" for monitoring and displaying student interactions. Michael Paterson and Paul Wexelblat designed and constructed a computer-controlled vehicle for problem-solving study. Seymour Papert and Richard Grant assisted in the design and planning of the earlier phases of the work; George Lukas made major contributions to methodological developments during the final phase. The report was prepared by Wallace Feurzeig and George Lukas.

2.1 Problem-Solving and Programming Languages

An important open question in the theory and practice of education is whether the notions and skills of formal reasoning and problem-solving can be taught. These skills are important, not only for their own sake, in direct application to formal work, but even more for their side effects. It is plausible that persons who have the skills and habits of organizing their approach to mathematical and formal problems will be better able to deal with more complicated and realistic situations.

New approaches to teaching mathematical problem-solving skills have been explored by a number of investigators. These include the various "discovery" methods and several experimentally-oriented curricula employing mathematics laboratory materials of many kinds. Such approaches generally have the object of making students welf-conscious about the process of solving problems. The most explicitly elaborated program was described by George Polya. Polya seeks to inculcate an understanding of mathematical ways of thinking by making students familiar with the kinds of steps performed in the course of solving mathematical problems. His major contribution was to provide an explicit and systematic checklist of procedures a student can apply when faced with the kind of problem that has no obvious solution. Students are directed toward solving problems in a deliberate and systematic fashion, through following heuristic guidelines for conceiving, executing, and testing plausible plans of attack.

Teaching the art of solving problems nevertheless remains an art. The new approaches have had very limited success. For example, Polya's heuristics -- find a similar but simpler problem; formulate a plan of attack and try it, divide the problem into

subproblems, etc. -- cannot be carried out with students who do not already possess considerable mathematical experience and sophistication. Indeed, for many students the concept of a procedure for solving problems is vague because the very idea of procedure is itself vague. Further, Polya does not tell us what happens when students attempt to follow his excellent precepts. Careful studies of the specific difficulties actually experienced by students in the course of trying to solve mathematical or other intellectual problems are difficult to design and expensive to carry out. The problems include finding an appropriate problem context, and observing the steps in the reasoning of a subject, his manipulation of material, his reaction to conflict and counter-suggestion, etc. Nevertheless, significant advances in teaching problem-solving will very likely depend on improving our understanding of, and our ability to diagnose, student difficulties.

Our thesis is that teaching the use of a suitable programming language will provide a substantially improved means of studying, diagnosing, and helping to overcome students' difficulties in solving problems. Such a programming language must be easily accessible to persons inexperienced in formal thinking, and must provide a natural way of expressing problem-solving procedures of many kinds, including the simple tasks suitable for beginning students. Moreover, it must be particularly useful in elucidating the set of issues which cause the greatest difficulties for beginning students. We have created such a programming language, LOGO, described in brief in Section 2.2.

Using LOGO, the process of formulating problems as computer programs is useful in helping students and teachers in several ways including the following.

(1) The use of LOGO facilitates the acquisition of rigorous thinking and expression. Students impose the need for precise statement on themselves through attempting to make the computer understand and carry out their commands. The literal-mindedness of the computer clearly shows the necessity for precise formal description, not only of the problem itself, but of the student's own steps -- successful and unsuccessful -- towards a solution.

(2) The partial, tentative steps towards a solution are programs and thus are concrete, reactive objects. Any program used provides feedback to the student. Thus, we have a natural and effective experimental approach toward solving problems.

(3) LOGO programming provides highly motivated models for all the principal heuristic concepts.

It lends itself naturally to discussion of the relation of formal procedures to intuitive understanding of problems. It provides a wealth of examples for heuristic precepts such as "formulate a plan", "separate the difficulties", "find a related problem", etc. Thus, it provides a natural context for realizing Polya's approach to teaching.

It provides a sense of formal methods and their purpose. It gives the student a chance to learn to distinguish situations where rigor is necessary from those where looser thinking is appropriate.

In particular, it provides models for the contrast between the global planning of an attack on a problem and the formal detail of an elaborated solution. In the context of programming, the concept of subproblem or subgoal emerges crisply.

The concrete form of the program and the interactive aspect of the machine allow "debugging" of errors to be identified as a definite, constructive, and plannable activity. The programming concept of a "bug" as a definite, concrete, existent entity to be hunted, caught, and tamed or killed is a valuable heuristic idea.

(4) By enlarging the scope of applications, LOGO allows every problem to be embedded in a large population of related problems of all degrees of difficulty, for example:

Through LOGO programming, mathematical induction can be presented and generalized by its relation to recursion.

The extension of an operation to a larger domain becomes an everyday activity.

Generalizing this, generalization becomes an activity undertaken routinely by students.

Functions become familiar things one constructs to serve real purposes. Students use these functions as building blocks for constructing more complex functions which often are elements of still more powerful structures, useful in dealing with more difficult problems.

(5) Solving a mathematical problem is a process of construction. The activity of programming a computer is uniquely well suited to transmitting this idea. The image we would like to convey could, roughly speaking, be described thus: A solution to a problem is to be built according to a preconceived, but modifiable, plan,

out of parts which might also be used in building other solutions to the same or other problems. A partial, or incorrect, solution is a useful object; it can be extended or fixed, and then incorporated into a large structure. This image is mirrored in the activity of writing LOGO programs.

(6) The use of computers and LOGO is relevant to what is perhaps the most difficult aspect of mathematics for a teacher: helping the student strive for self-consciousness and literacy about the process of solving problems. High school students can seldom say anything about how they work towards the solution of a problem. They lack the habit of discussing such things and they lack the language necessary to do so. A programming language provides a vocabulary and a set of experiences for discussing mathematical concepts and problems. LOGO programs are more "discussable" than traditional mathematical activities: one can talk about their structure, one can talk about their development, their relation to one another, and to the original problem.

(7) Finally, a by-product of using LOGO is the automatic generation of printed protocols showing a record of the in vivo interaction between the student and the computer. His work is thus available for diagnostic study at a level of detail sufficient for making plausible hypotheses about his underlying thinking and ostensible difficulties.

An understanding, or even a clear appreciation, of thsse points is impossible without a brief description of the LOGO language. The presentation that immediately follows introduces the elements of LOGO. The use of LOGO programming in problem-solving is discussed subsequently.

2.2 A Brief Description of the LOGO Programming Language

The LOGO programming language was specifically designed for teaching mathematical thinking and problem-solving. The structure of LOGO programs and the flavor of the language are illustrated next.

LOGO is a language for expressing formal procedures. A LOGO procedure is written in an idiom similar to recipes in cooking. It has a name; it usually has ingredients (these are called its inputs); and it has a sequence of instructions telling how to operate upon its inputs (and upon the things made from them along the way) to produce a desired effect or to make some new thing (this is called its output).

To illustrate, we define a procedure for doubling a number. We begin by choosing a word for the name of the procedure -- DOUBLE in this case. Next we choose names for the inputs -- in this case there is a single input -- NUMBER. So, the title of the procedure is TO DOUBLE /NUMBER/ (like to boil an egg). Note the slash marks around NUMBER -- slashes are used to demarcate names of *things*; names for *procedures* like DOUBLE and for *already-built-in instructions* are written without any marks around them.

When we give LOGO the command PRINT DOUBLE OF 5 we want the teletype to respond 1Ø; when we say PRINT DOUBLE OF 9999 we want the response 19998. So now we set down the instructions for performing DOUBLE. Actually, one instruction suffices.

```
OUTPUT SUM OF /NUMBER/ AND /NUMBER/
```

This instruction is composed of two elementary (i.e., already-built-in) instructions -- OUTPUT and SUM.

OUTPUT has the meaning "the answer is". Thus, OUTPUT SUM OF /NUMBER/ AND /NUMBER/ means that the answer is SUM OF /NUMBER/

AND /NUMBER/. SUM is an operation which needs two inputs (these must be integers). Its output is their sum. Thus, SUM OF 3 AND 2 has the output 5. The LOGO instruction: PRINT SUM OF 3 AND 2 causes the teletype to print 5.

The entire procedure definition is:

```
TO DOUBLE /NUMBER/
1 OUTPUT SUM OF /NUMBER/ AND /NUMBER/
END
```

where the integer 1 is used to label the instruction line (in this case there is only one line, but procedures often have several lines of instructions), and END marks the end of the definition. When this completed definition is typed in, LOGO acknowledges by responding: DOUBLE DEFINED. From that point on, the procedure DOUBLE can be used as if it had always been part of LOGO, just like PRINT and SUM. The new procedure is used by typing:

```
PRINT DOUBLE OF 2
```
The machine responds with the answer
```
4
```

(We underscore the student's typing in these and the following examples to distinguish them from LOGO's responses.)

Procedures can be chained. Thus:

```
PRINT DOUBLE OF DOUBLE OF 2
8
```

Procedures can also be embedded in the definition of new procedures. For example:

```
TO QUAD /NUMBER/
1 OUTPUT DOUBLE OF DOUBLE OF /NUMBER/
END

PRINT QUAD OF 123
492
PRINT DOUBLE OF QUAD OF 7
56
```

There are a relatively small number of elementary operations and commands in LOGO. An operation which is analogous to the operation SUM for integers is the operation WORD for alphanumeric words. Thus, PRINT WORD OF "SUN" AND "STAR" will cause the LOGO word SUNSTAR to be printed. The operations SUM and WORD are used to put things together. LOGO also has operations for taking things apart. These are FIRST, LAST, BUTFIRST, and BUTLAST.

```
PRINT FIRST OF "BOX"
B

PRINT BUTFIRST OF "BOX"
OX

PRINT LAST OF "BOX"
X

PRINT BUTLAST OF "BOX"
BO
```

BUTFIRST means *all but the first* letter of the word and BUTLAST means *all but the last* letter.

Some elementary LOGO operations have no inputs. An example is the operation RANDOM whose output is a one-digit random number.

```
PRINT RANDOM
7
PRINT RANDOM
4
```

Two basic acts in procedures are *making* new LOGO things and *testing* them to see whether they satisfy some condition, such as a stop rule. To make a new LOGO thing, we type the command MAKE. LOGO responds by asking first for the name and then for the thing, i.e., for a LOGO expression for the new thing. Thus, if we want to make a list of the even digits, and call this "EVENS":

```
MAKE
   NAME:   "EVENS"
   THING:  "Ø 2 4 6 8"
```

If we then type PRINT /EVENS/, LOGO responds:

```
Ø 2 4 6 8
```

PRINT "EVENS", would have caused LOGO to print EVENS. Quotation marks refer to a LOGO thing *directly*. Slash marks refer to a thing by its *name*.

To test whether a LOGO thing satisfies some condition, we introduce the notion of predicates, i.e., operations which have two possible outputs, "TRUE" and "FALSE". The identity operation IS is one of the elementary LOGO predicates. IS takes two inputs and has the output "TRUE", if these inputs express the same thing. Otherwise it has the output "FALSE". Thus,

```
PRINT IS 2 SUM OF 1 AND 1
TRUE

PRINT IS 2 1
FALSE
```

The command TEST, and the associated commands IF TRUE and IF FALSE, are used with a predicate as in the following program:

```
TEST IS 2 2

IF TRUE PRINT "GOOD"
GOOD
```

The use of RANDOM, MAKE, and TEST in introducing recursion is illustrated in the following procedures for printing lists of random numbers.

```
TO NUMBER
1 PRINT RANDOM
END
```

This procedure is used by typing:

```
NUMBER
```

The machine responds with a number

```
8

NUMBER
5
```

etc.

The repetitive act of typing NUMBER is easily mechanized by writing a new procedure to do just this.

```
TO SLEW
1 NUMBER
2 SLEW
END
```

We have incorporated into SLEW the instruction to perform another procedure, NUMBER, and then the instruction to SLEW, i.e., to do the same again. So when we type SLEW, we obtain an endless sequence.

```
SLEW
7
3
Ø
9
:
```

As well as using another procedure, NUMBER, SLEW also uses itself -- it is a simple example of a recursively defined procedure. To modify SLEW so as to produce a definite number of random digits, we introduce an input /NTIMES/: the number of times we still have to SLEW.

```
TO SLEW /NTIMES/
1 TEST IS /NTIMES/ Ø
2 IF TRUE STOP
3 PRINT RANDOM
4 MAKE
    NAME:  "NEWNUMBER"
    THING:  DIFFERENCE OF /NTIMES/ AND 1
5 SLEW /NEWNUMBER/
END
```

(The elementary operation DIFFERENCE denotes integer subtraction. Thus DIFFERENCE OF 3 AND 1 is 2.)

The use of this new SLEW procedure is illustrated by:

```
SLEW 2
Ø
3
```

```
SLEW 1
2

SLEW 3
2
5
6

etc.
```

To show how LOGO performs SLEW, let's ask it to do SLEW 2 and trace through its subsequent operation, instruction by instruction. When we type in SLEW 2, LOGO takes the definition of the procedure SLEW and uses it as follows:

```
Round 1  TO SLEW "2"

Title Line:  /NTIMES/ is "2"
Line 1:      "2" is not "Ø"
Line 2:      Therefore this instruction is ignored
Line 3:      LOGO prints the output of RANDOM, say the
             digit 4
Line 4:      /NEWNUMBER/ is "1"  (that is, 2 - 1)
Line 5:      LOGO invokes SLEW OF "1"

Round 2  TO SLEW "1"

Title Line:  /NTIMES/ is "1"
Line 1:      "1" is not "Ø"
Line 2:      Ignored
Line 3:      LOGO prints the output of RANDOM, this time
             perhaps the digit 5
Line 4:      /NEWNUMBER/ is "Ø"  (that is, 1 - 1)
Line 5:      LOGO invokes SLEW OF "Ø"

Round 3  TO SLEW "Ø"

Title Line:  /NTIMES/ is "Ø"
Line 1:      "Ø" is "Ø"
Line 2:      Therefore LOGO stops
```

Using LOGO, recursive procedures can be written and systematically extended in a rich variety of mathematical contexts. An example of a deeper recursive procedure, closely related to the principle of "mathematical induction", is the factorial function:

```
FACTORIAL(1) = 1
FACTORIAL(N) = N X FACTORIAL(N-1), N >1
```

In LOGO we write a corresponding procedure as follows:

```
TO FACTORIAL /N/
1 TEST IS /N/ 1
2 IF TRUE OUTPUT 1
3 MAKE                                    (The operation PRODUCT
    NAME:  "N-1"                          denotes integer multi-
    THING:  DIFFERENCE OF /N/ AND 1       plication.)
4 OUTPUT PRODUCT OF /N/ AND FACTORIAL OF /N-1/
END

PRINT FACTORIAL OF 7
5040

PRINT FACTORIAL OF DOUBLE OF 3
720
```

A syntactically similar non-numerical procedure, for reversing the order of the letters in a word (i.e., writing a word backwards), is:

```
TO REVERSE /W/
1 TEST IS COUNT OF /W/ 1          (COUNT OF /W/ is the
2 IF TRUE OUTPUT /W/              number of letters in /W/.)
3 MAKE
    NAME:  "NEWWORD"
    THING"  BUTLAST OF /W/
4 OUTPUT WORD OF LAST OF /W/ AND REVERSE OF
  /NEWWORD/
END

PRINT REVERSE OF "ELEPHANT"
TNAHPELE

PRINT REVERSE OF FACTORIAL OF 7
0405
```

The basic capabilities of LOGO described above can be developed and extended in a natural way. In Section 4 we show how LOGO is used in several teaching sequences where these capabilities are used to build up complex program structures in various problem-solving contexts.

3. THREE LOGO TEACHING EXPERIMENTS

The LOGO course in problem-solving was developed and tested by means of a sequence of three teaching experiments. In the first, the participants included both school teachers and developmental psychologists with educational interests. The object of this phase of the teaching was to test our ideas about the use of LOGO in studying problem-solving and to develop specific LOGO materials for further use.

In the subsequent teaching experiments we further developed and tested these ideas and materials. This work involved two groups of students with distinctly different motivational and conceptual difficulties. The first was composed of eighth grade "problem students" who had developed strong resistance to working on virtually any kind of organized intellectual tasks. The other group comprised college students with a history of poor performance in mathematical work. The three experiments are described in the sections following.

3.1 Teaching Teachers - Summer Workshop, 1969

We developed with professional subject-trainees the idea of using LOGO as a tool for introducing constructive methods of problem solving. In this investigation we built on earlier work involving LOGO in studying problem-solving concepts such as planning, modeling, and testing. We also sought to obtain some experience with the problems of training teachers to learn and use LOGO in this way.

The course was given as an intensive summer workshop in July-August, 1969. The participants were two elementary school

teachers with limited mathematical background, two junior high school mathematics teachers who had majored in college mathematics, three Canadian professors of education and psychology who were personally interested in learning research based on the use of LOGO and were planning to implement LOGO in a French version to be used in Piagetian experiments, and two staff members of Bolt Beranek and Newman (BBN) with backgrounds in mathematics curriculum research and teaching. Except for one of the junior high school teachers and one of the BBN staff members, the participants had no previous familiarity with programming.

The plan was to immediately plunge the workshop participants into using LOGO. Thus, the following recursive procedure for administering an addition quiz was introduced to them on the first day of the course.

```
TO ADDQUIZ
1Ø PRINT "TYPE A NUMBER"
2Ø MAKE
     NAME:  "NUM1"
     THING:  REQUEST
3Ø PRINT "TYPE ANOTHER NUMBER"
4Ø MAKE
     NAME:  "NUM2"
     THING:  REQUEST
5Ø PRINT "WHAT IS THE SUM OF YOUR TWO NUMBERS?"
6Ø MAKE
     NAME:  "ANSWER"
     THING:  REQUEST
7Ø MAKE
     NAME:  "RIGHT ANSWER"
     THING:  SUM OF /NUM1/ AND /NUM2/
8Ø TEST IS /ANSWER/ /RIGHT ANSWER/
9Ø IF TRUE PRINT "YES, THAT'S RIGHT."
1ØØ IF FALSE PRINT "NO, TRY AGAIN."
11Ø ADDQUIZ
END
```

The trainees were introduced to LOGO operations, commands, names, and features gradually, as needed for their programming assignments. In the first two weeks, while writing and debugging programs, they did, in fact, learn virtually all of the LOGO vocabulary without any special emphasis on this. This "Berlitz" technique of requiring the use of the language in a working context ab initio, introduced some confusion and sense of pressure during the first few days. The participants benefited in the long run, though, from having to confront more realistic problem-solving situations. After the first weeks, they were confident about approaching and handling tasks of moderate to large scope.

Heuristics for Planning a Procedure

Even for the simplest programs, planning precedes implementation. To assist in this stage of problem-solving, the class was introduced to various heuristics for planning a procedure. An example of such a heuristic is

(1) Find easy cases,
(2) Reduce the hard cases to these easy ones.

It was emphasized that these plans do not always work, but that having a collection of plans enables one to "do something" when faced with a problem.

The use of the foregoing heuristic is illustrated with the LOGO procedure FIND. FIND is an operation with two inputs, the first of which is a word and the second of which is the position of the character in the word to be "found". Examples of its use are:

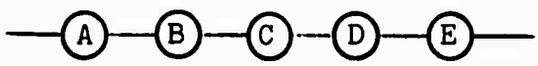
```
FIND "ABC" 1 = "A"
FIND "ABC" 3 = "C"
```

The easy case for FIND is when the first character is to be found. So we begin by writing this part of the procedure

```
TO FIND /SENTENCE/ /NUMBER/
1Ø TEST IS /NUMBER/ 1
2Ø IF TRUE OUTPUT FIRST OF /SENTENCE/
```

Now we turn to the reduction of the harder cases to this easy case. Sometimes, especially for young children, a physical model is useful. So we construct one to illustrate this idea here.

Model for FIND "ABCDE" 4



String of beads representing "ABCDE"

To perform FIND "ABCDE" 4, one merely peels beads off the string, reducing the count by one, each time, until it becomes 1.

Discussion of this model leads to the conclusion that FIND /SENTENCE/ /NUMBER/ is equivalent to the problem FIND BUTFIRST OF /SENTENCE/ DIFFERENCE OF /NUMBER/ AND 1. So we MAKE two new things:

BUTFIRST OF /SENTENCE/

DIFFERENCE OF /NUMBER/ AND 1

and we give these the names "NEWSEN" and "NEWNUM", respectively. Thus,

```
TO FIND /SENTENCE/ AND /NUMBER/
1Ø TEST IS /NUMBER/ 1
2Ø IF TRUE OUTPUT FIRST OF /SENTENCE/
3Ø MAKE
     NAME:  "NEWSEN"
     THING:  BUTFIRST OF /SENTENCE/
```

```
4Ø MAKE
     NAME:  "NEWNUM"
     THING:  DIFFERENCE OF /NUMBER/ AND 1
5Ø OUTPUT FIND OF /NEWSEN/ AND /NEWNUM/
END
```

LOGO provides a natural framework for approaching problems with well formulated strategies. Thus, as well as the "reduce the hard cases to easy ones" heuristic, other heuristics can be implemented in LOGO in a straightforward fashion. An example of such a useful heuristic is "subdivide a complex problem into subproblems". The use of this heuristic was discussed at some length in application to developing strategic game-playing programs such as NIM.

The NIM-playing program was divided into subprograms for initializing play, requesting a user's move, checking the legality of a move, generating the computer's move, sequencing the play (computing the next player), keeping score (computing the current number of chips remaining), and checking after each move to see whether the game has been won or lost. These components can be further subdivided into simpler ones until each program is adequately clear and transparent. (Alternatively, a component program can be made more complicated. For example, the first version of a program for generating the computer's move might simply choose a move at random. In subsequent extensions it can be replaced by a series of programs to carry out more effective strategies for computing moves.)

An example of a related planning heuristic developed in the workshop was "build complex procedures out of previously developed simpler ones". The use of this heuristic was illustrated in the generation of a series of successively more

English-like grammatic sentences, and of complex structures such as poetic forms of various kinds. The reverse problem of analyzing such given structures to determine the rules which could have been used to compose them was also discussed. The feasibility of implementing planning heuristics like "to analyze a structure first try to synthesize it from simpler structures" was considered in the context of generating algebra story problems starting from formal equations.

Heuristics for Debugging a Procedure

In addition to the general lack of the notion of a planning phase of work on a problem, students seldom have definite ideas or methods for diagnosing or even detecting the errors in their own work. Students frequently give up when their steps in solving a problem are not successful -- rather than trying to understand and correct them. The potential value of LOGO in this connection showed up in even the simplest tasks in our earlier teaching. The kind of problem "debugging" experience it makes possible was illustrated in the workshop by presenting actual instances of student programming particularly chosen to show the erratic course of program development in some detail.

An example is provided by the following discussion of the work of a beginning student, Steven, a few weeks after his introduction to LOGO. He was working on a project to write a program called COUNTDOWN which was to mimic the numerical countdown procedure accompanying a space launch. Steven's program was to work as follows. (The ← indicates that LOGO is ready for the user's input.)

```
←COUNTDOWN
10 9 8 7 6 5 4 3 2 1 0 BLASTOFF!
←
```

He then wrote a more general COUNTDOWN procedure with a variable starting point. For example, if one wished to start at 15:

```
←COUNTDOWN 15
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 BLASTOFF!
←
```

He had already studied LOGO programs having a similar effect, though in rather different, nonnumerical, contexts. Thus, for example, he had used the program CHOP which worked as follows:

```
←CHOP "ABCDE"
ABCDE ABCD ABC AB A
←
```

In the case of CHOP, each successive output is obtained from the previous one by chopping off the rightmost letter. The procedure terminates after it has "chopped off" all the letters and there is nothing left in the word (i.e., the word has become /EMPTY/). Steven had this procedure in mind when he tried to write COUNTDOWN. His first attempt, however, followed CHOP a little too closely. It was written as follows.

```
TO COUNTDOWN /NUMBER/
1 TYPE /NUMBER/
2 TEST IS /NUMBER/ /EMPTY/
3 IF TRUE TYPE "BLASTOFF!"
4 IF TRUE STOP
5 MAKE
     NAME:  "NEW NUMBER"
     THING:  DIFFERENCE OF /NUMBER/ AND /NUMBER/
6 COUNTDOWN /NEW NUMBER/
END
```

When Steven tried his procedure, this is what happened.

```
←COUNTDOWN 5
5 Ø Ø Ø Ø Ø ...          (it went on and on until he stopped
←                        the program manually)
```

Obviously, something was wrong. He saw his first "bug". He had performed the wrong subtraction in instruction line 5; he meant to decrement the number by 1. He fixed this by changing the instruction to:

```
5 MAKE
     NAME:  "NEW NUMBER"
     THING:  DIFFERENCE OF /NUMBER/ AND 1
```

Then he tried again.

```
←COUNTDOWN 5
5 4 3 2 1 Ø -1 -2 -3 -4 ... (and again he had to stop the program)
←
```

Somehow, his stopping rule in instruction line 2 had failed to stop the program. He saw his bug -- instead of testing the input to see if it was /EMPTY/, he should have tested to see if it was Ø. Thus,

```
2 TEST IS /NUMBER/ Ø
```

He made this change in line 2 and then tried once more.

```
←COUNTDOWN 5
5 4 3 2 1 Ø BLASTOFF!
←
```

And now COUNTDOWN worked.

As a follow-on, he wrote a LOGO procedure for counting down by two's. His strategy was to build the new procedure (he called it COUNTDOWN-2) from the current one, COUNTDOWN, simply by

changing instruction line 5 to decrement /NUMBER/ by 2 instead of 1.

```
TO COUNTDOWN-2 /NUMBER/
1 TYPE /NUMBER/
2 TEST IS /NUMBER/ Ø
3 IF TRUE TYPE "BLASTOFF!"
4 IF TRUE STOP
5 MAKE
      NAME:  "NEW NUMBER"
      THING:  DIFFERENCE OF /NUMBER/ AND 2
6 COUNTDOWN-2 /NEW NUMBER/
END
```

Then he tried it out.

```
←COUNTDOWN-2 5
5 3 1 -1 -3 -5 -7 ...   (and so on, until he stopped the program)
←
```

He spotted his bug immediately -- the stop rule had to be changed. It worked all right for an even starting number but not for an odd one. So he changed it to:

```
2 TEST EITHER (IS /NUMBER/ Ø) (IS /NUMBER/ 1)
```

Now his program worked for odd-number sequences,

```
←COUNTDOWN-2 5
5 3 1 BLASTOFF!
←
```

as well as for even ones.

```
←COUNTDOWN-2 1Ø
1Ø 8 6 4 2 Ø BLASTOFF!
←
```

His work in developing subsequent procedures (for counting up from a given number to a given larger number, for counting down

from a number to an arbitrary smaller number, and for counting up and down between two limits, i.e., oscillating, a specified number of times) was also reconstructed in like fashion.

Such protocols of student sequences, together with the ones drawn from the participants' own work, provided a rich source for studying bugs of many kinds. Through such comparative and clinical study we described several of the more common types, the program contexts in which these occurred, and good ways to find and correct them.

Program Forms and Structures

As well as heuristic aspects of problem-solving, LOGO was used to study the associated formal aspects. A particularly important one is the concept of program form. A series of standard recursive program forms of increasing complexity was introduced. These served as models for expressing a great variety of problem-solving processes. Some standard uses of these forms were discussed along with the bugs typically encountered in each case. The simplest form, simple recursion, is shown in:

```
TO SING
1 PRINT "LA LA"
2 SING
END

←SING
LA LA
LA LA
LA LA
 : : :
```

A variant is simple recursion with an input, as in

```
TO SAY /SOMETHING/
1 PRINT /SOMETHING/
2 SAY /SOMETHING/
END

←SAY "CAT"
CAT
CAT
CAT
: : :
```

Simple recursion is used to express non-terminating invariant processes. A more interesting form includes both varied effects and a termination condition. This form of recursion is equivalent to simple iteration. An example is Steven's COUNTDOWN procedure, and the procedure SLEW discussed in Section 2.2.

A more complex form of recursion uses the OUTPUT command to transmit intermediate outputs. Recursive procedures of this kind can be used to express significant processes. For example, the procedure FIND was used to carry out the "reduction to easy cases" heuristic. Variations where the recursion is embedded in some larger operation are often useful -- examples are the procedures REVERSE and FACTORIAL given in Section 2.2.

Recursive procedures of a variety of forms of still greater complexity and power, including some which are not reducible to iteration, can easily be written. Recursive forms like those already introduced, however, are sufficient for representing problem-solving processes in virtually all applications of interest outside of advanced mathematical work.

Further complexity in formal problem-solving capabilities is better obtained by appropriately combining procedures of the various forms already introduced with nonrecursive procedures of certain standard forms to create composite structures. The idea of program structure gives the other dimension needed to enable relatively complex problem-solving processes to be built up from relatively simple procedures. Examples of some standard forms of multi-procedure structures were introduced to serve as models for student work. The extended development in the geometry drawing sequence, discussed in Section 4, is a concrete illustration. Examples of some student program structures are shown and discussed in Section 5.1.

In the last phases of the teaching experiment, the participants worked on a set of diverse demonstration projects of their own. The work was presented and critically analyzed by the entire group in a series of clinics. The participants were generally successful in working on problems with well formulated strategies. But they needed help in planning and organizing their work with more open-ended and complex problems. Thus we felt it necessary to write a number of extended LOGO sequences as paradigms for teachers. As a first step in this direction, we developed the material presented in Section 4, which introduces problem-solving with LOGO.

3.2 Teaching "Problem Students" - Muzzey Junior High School, 1970

In this section we describe a teaching experiment conducted with a small class of eighth-grade students at Muzzey Junior High School from March 1970 through June 1970. The class comprised six students each of whom had a history of resistance to

participating in intellectual work in the school. These students were frequently expelled from class for disciplinary reasons. They had difficulties with reading as well as in their mathematics and science.

In fact, we chose from the entire eighth-grade class those students with the greatest deficiencies in reading comprehension as consistently measured by standard tests -- the tested measures varied from three to five years below eighth-grade norms. The students showed no indication of deafness or other organic factors -- their learning performance problems were judged to be primarily motivational in character. Their IQ levels ranged from 87 to 117, averaging somewhat over 100. They were regarded as underachievers by teachers. Two of them were judged to be performing enormously below their potential.

The main object of this teaching was to explore the use of LOGO in motivating such students to do intellectual work. We believed that they would find appropriate work with LOGO on the computer so compelling that they would become genuinely engaged in many kinds of problem-solving tasks. We planned in the last phase of the teaching to see whether we could break through their conceptual barriers as well, and help them gain in both mathematical problem-solving skills and reading comprehension.

Lesson Materials

The students were involved in two main kinds of class activities. They first worked through a series of 24 programmed lessons giving an introduction to LOGO programming. The students usually worked at the computer terminal individually. Occasionally two were scheduled to work together on a lesson. As they learned how

to write procedures, they worked on various projects of their own choice, though they usually needed a great deal of guidance in modifying their ideas and in helping them plan and carry out their work.

The programmed lessons were written in LOGO itself in an interactive mode and in a relatively open-ended style, mixing straight exposition and test items with freer problem assignments calling for the student to write his own procedures. The lessons had previously been used with ease both by younger students and adults. As a side advantage, these lessons actively involved the students in doing a great deal of reading.

In the following pages, a fragment of a student's lesson transcript is reproduced to illustrate the material and the type of student-computer interaction involved. The student's typing is underscored. The ↓ mark indicates that the student is to press the carriage return key when he has finished reading the current line and is ready for the next line.

```
HERE IS A PROCEDURE WE USED WHEN WE STARTED WORKING WITH THE
COMPUTER.↓

FIRST TELL THE COMPUTER TO -- LIST SAYHI.↓

THEN TRY OUT THE PROCEDURE SAYHI TO SEE WHAT IT DOES.↓

WHEN YOU ARE READY TO GO ON TYPE -- MORE.↓

←LIST SAYHI
TO SAYHI
1Ø PRINT "HI"
2Ø SAYHI
END
```

```
←SAY HI                    (A typical bug - splitting a LOGO procedure
SAY IS NOT DEFINED         name made up of English words)
←SAYHI
HI
HI
HI
HI
HI                         (The student stopped the program by hitting
                           the break key at this point)
I WAS AT LINE 1Ø IN SAYHI
←MORE
NOW IT'S YOUR TURN TO WRITE A PROCEDURE.↓

TRY TO WRITE A PROCEDURE WHICH WILL PRINT YOUR NAME JUST THE
WAY SAYHI PRINTED HI.↓

WHEN IT WORKS TYPE -- NEXT.↓

←TO SAYMIKE
>1Ø PRINT "MIKE"
>2Ø SAYHI                  (Note the bug - probably a careless copy-
>END                       ing error when Mike looked back at the
SAYMIKE DEFINED            SAYHI definition above)

←SAYMIKE
MIKE
HI                         (The error is evident)
HI
HI
HI                         (He stopped the operation of SAYHI again)

I WAS AT LINE 1Ø IN SAYHI
←EDIT SAYMIKE
>2Ø SAYMIKE
>END
SAYMIKE DEFINED

←LIST SAYMIKE
1Ø PRINT "MIKE"            (Students were taught to list procedures
2Ø SAYMIKE                 which they had just edited to check their
END                        changes)
←SAYMIKE
MIKE                       (This time SAYMIKE worked)
MIKE
MIKE
MIKE                       (So he stopped)
```

```
←NEXT                    (and continued to the next part of the
                         assignment)
ARE YOU READY FOR A CHALLENGE?  ANSWER YES OR NO.
*YES

THIS TIME TRY TO WRITE A PROCEDURE THAT WILL WORK TO COPY
ANY WORD YOU GIVE IT.↓

CALL YOUR PROCEDURE -- WORK.  SO WHEN YOU WRITE WORK AND
THEN USE IT WITH THE INPUT "MARGE" IT SHOULD TYPE --
MARGE -- RIGHT DOWN THE PAGE.
     : : :
```

Student Projects

Observing the students' work at the computer it was apparent that LOGO provided a means of overcoming their resistance to formal ways of thinking. Working with computers was seen by them as "a good thing", just like shop and gym. Our task apparently reduced to finding programming contexts and problems going beyond the expository lesson materials which would be accepted as "relevant" by the students.

We found from early on in the class that most students were interested in using LOGO at two distinct levels of involvement. First, they simply liked to work at the computer terminal. The content and context of the work was often unimportant; indeed, the students often were quite happy doing routine, tedious, repetitive, mechanical tasks assigned to them so long as they could do these interactively at the terminal. In carrying out this assigned work, including much of the lesson material, they did not always find it important to think a great deal about what they were doing. They simply liked to do it, just as they liked running. Their compelling interest in using the machine

continued throughout the three-month period, from start to finish. During this course of time, they gradually acquired the formal material covered in the lessons.

The other and deeper level of involvement came from working on their own projects. There were three sources of such projects: some projects came out of what the students perceived as real, personal problems, some were expressions of protest directed at the school establishment, and some developed out of activities and games they already were interested in. Examples of these three kinds follow.

One student first consolidated the concept of formal procedure as the direct result of a real life problem that confronted him on his way to school. To seek redress from a bitter fight with a school bus driver that morning, he urgently needed to compose an affidavit. He decided to write it as a LOGO program since this would facilitate making additional copies for the school principal, the bus company, and his mother's attorney. (The school did not have typewriters and Xerox-type copying equipment readily available to students.) The first part of his program, COMPLAIN, is listed next.

```
TO COMPLAIN
10 PRINT "ON THE DAY OF MAY 4, 1970 THE BUS DRIVER TOLD ME TO GET OFF
   THE BUS AND I SAID ,WHY AND HE SAID GET OFF THE BUS AND I DIDN'T ."
20 PRINT "HE WANTED ME OFF BECAUSE OF MY BUS PASS AND I SAID IT WAS ALL
   RIGHT HE HAD SAID, BEFORE TO GET IT CHANGED AND I DID BY MR. TERRY
   AND HE PUT A 14 IN THE MIDDLE OF THE CARDA AND HE WANTED A NOTE FROM
   MR. TERRY AND HE WANTED IT ON MAY 4, 1970."
30 PRINT "THE DAY THAT IT HAPPEN WAS ON MAY 1,1970 AND I TOLD HIM I WAS
   GOING TO WALK TO SCHOOL THAT DAY."
40 PRINT "AND ON MAY4,1970 I GOT ON THE BUS AND I SHOWED HIM MY PASS
   AND HE SAID GET OFF AND SAID NO ,HE GOT OUT OF HIS SEAT AND GRABED
   ME AND TOLD HIM TO LET GO AND DIDN'T ."
50 PRINT "HE TRYED TO TRIP ME AND THEN HE STARTED TO PUSH ME AROUND AND
   AS HE WAS PUSH ME OFF THE BUS HE WAS KNOCKING DOWN OTHER PEOPLE."
60 PRINT "HE TOLD MR.TERRY WHAT HAPPEN AND HE SAID HE HAD ROAD BUS 15
   AND I TOLD MR. TERRY THAT I ROAD BUS 14 AND I COULD PROVE THAT I WAS
   ON BUS 14 ."
   : : :
```

The effect of COMPLAIN is evident. The procedure is an instance of the most elementary program form. The same student became involved in writing more complex programs through subsequent personal incidents. After one of these he was charged to write the sentence "I will never throw a book out of the window again" 200 times. He conceived the notion of doing this by writing the following LOGO procedure.

```
TO SWEAR-OFF /NTIMES/
1Ø TEST IS /NTIMES/ Ø
2Ø IF TRUE STOP
3Ø PRINT "I WILL NEVER THROW A BOOK OUT OF THE WINDOW AGAIN"
4Ø SWEAR-OFF (DIFFERENCE OF /NTIMES/ AND 1)
END
```

The procedure is essentially the same as Steven's COUNTDOWN procedure discussed in Section 3.1. The effect, however, is different. When SWEAR-OFF 2ØØ was executed it produced a list of 200 copies of the designated sentence. This computer printout was deemed an acceptable way of carrying out the punishment.

As a follow up we gave the student the problem of writing a more general procedure COPY with two inputs designating the message to be copied and the number of times it was to be copied. For example, COPY "I'LL NEVER SLEEP IN CLASS" 1ØØØØ would print the sentence "I'LL NEVER SLEEP IN CLASS" ten thousand times. He needed some help, but he was able to write the following procedure.

```
TO COPY /ANYTHING/ /NTIMES/
1Ø TEST IS /NTIMES/ Ø
2Ø IF TRUE STOP
3Ø PRINT /ANYTHING/
4Ø COPY /ANYTHING/ (DIFFERENCE OF /NTIMES/ AND 1)
END
```

This task was a formidable one for the student in question. It showed a considerable advance in his formal and intellectual grasp during the three-month period. Other students arrived at this level of skill at earlier points and went on to carry out larger projects involving the development of more complex procedures and program structures.

An example is a program for playing ROULETTE which a student wrote on his own initiative and with little outside help. A run from one of the later versions of his program is shown next.

```
ROULETTE
YOU START WITH A $100 BILL. $100 IS THE HOUSE LIMIT. YOU MUST BET $1 OR
MORE.
THE WHEEL SPINS. PLACE YOUR BET ON (1) A SINGLE NUMBER. (2) ANY TWO
NUMBERS. (3)ANY THREE NUMBERS (4) ANY FOUR NUMBERS .(5) ANY SIX
CONSECUTIVE NUMBERS. ((6) TWELVE CONSECUTIVE NOS. (7) ANY 18
CONSECUTIVE NOS. (8) ALL ODD OR EVEN NOS.
*1
HOW MUCH MONEY DO YOU BET?
*50
OK, YOU HAVE DECIDED TO BET ON ONE SINGLE NUMBER. YOU MAY BET ON ANY
NUMBER. 0-36. IF YOU BET ON ONE NUMBER 1-36 AND THE NUMBER IS 0 YOU MAY
KEEP YOUR BET ON THE TABLE FOR THE NEXT BET. WHAT NUMBER DO YOU BET
YOUR MONEY ON?
*33
I'M AFRAID YOU HAVE LOST YOUR BET.
YOU HAVE ONLY 50 DOLLARSTHE NUMBER WAS 6
WOULD YOU LIKE TO BET AGAIN? ANSWER Y OR N
*Y

YOU NOW HAVE  50 DOLLARS
WHAT TYPE OF BET ARE YOU MAKING ?
*2
HOW MUCH MONEY DO YOU BET?
*25
```

```
YOU HAVE DECIDED TO BET ON 2 NOS.PLEASE NOTE:YOU MAY NOT BET ON ZERO
YOUR FIRST NUMBER IS:
*26
AND YOUR SECOND NUMBER IS:
*11
ALL RIGHT, LET'S SEE HOW YOU DID. THE NUMBER WAS 20
SORRY BUT YOU CANT WIN THEM ALL
YOU NOW HAVE ONLY  25 DOLLARS
WOULD YOU LIKE TO BET AGAIN? ANSWER Y OR N
```

⋮ ⋮ ⋮

The procedure in its final form was several pages long. It probably was the most intense, extensive, and concerted intellectual enterprise the student had ever undertaken.

Some students were involved in an extended project to generate geometric drawings and pictures at the teletype. A sequence based on this drawing project, as further developed in the University of Massachusetts teaching, is described in Section 4.1.

By the end of the course most students' intellectual resources -- recognizing problems, organizing work into transparent programs, debugging simple programs, and modifying and extending work -- were much improved. This success carried over to other areas of school work -- teachers remarked particularly on the students' increased classroom involvement and participation. These findings are subjective but "objective" evaluations were also carried out. In doing this we found that computer testing provides an improved means of measuring performance of low achieving students. This study is described in the Appendix.

3.3 Teaching Unmathematical Undergraduates -- University of Massachusetts, Boston, 1971

An undergraduate course, within the Mathematics Department of the University of Massachusetts at Boston, was conducted by George Lukas in the spring semester of 1971. This course was one of a number of courses intended to meet the needs of undergraduates who, it was felt, had no chance of passing the normal, required, mathematics course. Selection of students for this special program was based on a score of less than 400 on the mathematics aptitude part of the College Entrance Examination Boards and on an interview with the faculty member in charge of the program.

We felt that LOGO could serve a very special role for students at this level. The chief deficiency in such students is a lack of basic problem-solving skills, and not, as appears superficially, a lack of mathematical aptitude. The lack of problem-solving skills is most evident in work with mathematics, but, if careful study is made of language skills and other intellectual areas, the same deficiency is noted in each. Thus, we wanted to use LOGO, not as a vehicle for conveying specific subject matter, but to teach the most fundamental aspects of reasoning at a formal level -- generalization, planning, error debugging, etc.

This teaching experience has had a number of useful results: We have developed and tested sequences for use at this level of teaching, based on word-form generation and on teletype geometry. These are included in later sections of this report. We developed a means of having the computer save student work in the form of "dribble files" for later analysis. This too is discussed later. Finally, and most important, we ascertained the utility of using LOGO in this way for teaching basic problem-solving

skills to students who are considerably below average in this area.

Nine students were chosen at random from the group of University of Massachusetts students eligible for this course. They met for five hours a week and spent all their class time at teletypewriters. There were no homework assignments. There were three teletypewriters connected to LOGO via BBN's TENEX system by telephone lines. The students were carefully divided into groups of three, each of whose members worked together. Reassignment of students to groups was made from time to time to keep each group balanced so that each student contributed to the work. Each group of three worked in a separate office. The instructor walked from office to office in the course of a lesson, monitoring the student work. He interceded only when a serious error had been made which the students were unlikely to diagnose on their own, or when a new topic was to be introduced.

The course began with an introduction to the elements of LOGO. Some existing materials of a CAI nature, written in LOGO, were used for this. The remainder of the term was spent on various projects. These included the geometry and language generation mentioned earlier, as well as a craps playing program, code deciphering programs, and work on a number of similar topics. The criteria for choice of topic were that the students could achieve interesting results, that it involved new aspects of problem-solving skills, and that it would engage student interest over an extended period of time. Given the resistance of the students to material that looked anything like mathematics, satisfying the last of these was by no means a trivial matter.

As the term progressed, student ability to handle program details and simple program forms became automatic in most cases. This was very encouraging, as it indicated an internalization of rather general algorithms, something the students were unable to do previously. Also, their ability to communicate their ideas in general terms improved, and this development of a problem-solving meta-language is extremely important. The quality of results achieved, as seen, for example in the geometry sequence, improved over the course of the term and concurrently so did student confidence. To indicate the extent to which students at the end of the term felt themselves capable of handling formal processes, over half indicated their intention of taking further mathematics courses.

Extensive examples of both student-written procedures and the uses to which they were put are contained in the sections on the geometry and language sequences and in the section describing "dribble files".

4. INTRODUCTORY LOGO COURSE ON PROBLEM-SOLVING

The three-sequence course is described in the sections following. The material was designed for introductory use. The initial sequence on teletype geometry is developed in greatest detail as it evolved from the teaching at Muzzey Junior High School and in more refined form at the University of Massachusetts. The shorter sequences give two distinctly different problem contexts -- generating English and controlling the robot "turtle". The more advanced sequences on problem-solving have been written as part of our LOGO mathematics curriculum in work supported by the National Science Foundation (Ref. 1).

4.1 Geometry Sequence

We present a teaching sequence in which geometric ideas are developed by use of the teletype as drawing device. The sequence is based on part of the teaching done at University of Mass. at Boston. The programs and examples are taken from student work, unless otherwise indicated. The only changes have been in the names used for procedures and dummy variable names, and this was only done where clarity was substantially improved thereby.

The sequence falls naturally into two parts. In the first, drawing procedures draw figures line by line. There are several advantages in starting with this approach: A simple recursive form suffices for most procedures so that a student can write many procedures quickly. There is no need for communication of results when a procedure is invoked by another one. In other words, the invoking procedure is not affected by the result of executing the one it invoked. This means that we are writing only commands and not operations, thus problems of communication are avoided. Finally, the ideas developed in this introductory sequence lead naturally to the more sophisticated ideas and program structures involved in a Cartesian description of geometry, the second part of the sequence.

The second part of the geometry sequence uses a Cartesian description of figures -- descriptions of figures as pair lists are now the basic objects to be studied. Storing the figures makes possible a wide range of geometric and set-theoretic operations on figures. Due to the fact that these ideas were presented so late in the term at U.Mass./Boston, the sequence description is no longer so closely tied to student work. The procedures

described in this part of the work were given to the students; only the examples arise from student work. If the ideas had been presented earlier, students would have had little trouble writing most of the programs they used. The only really difficult ones are those for drawing and for ordering sets of points.

The use of the material presented here, preceded by a suitable introduction to the LOGO language, and including some of the suggested extensions, would form a coherent one-term course.

Students begin by generating patterns, using just the PRINT command within a procedure definition. Some of these patterns are freeform,

```
TO CURVE
1Ø PRINT "               ("
2Ø PRINT "                ("
3Ø PRINT "                ((("
END
```

others are more

```
TO DIAMOND
1Ø PRINT "    X"
2Ø PRINT "  X X X"
3Ø PRINT "X X X X X"
4Ø PRINT "  X X X"
5Ø PRINT "    X"
END
```

or less

```
TO DIAMOND
1Ø PRINT "      :"
2Ø PRINT "   :::::"
3Ø PRINT ":::::::::::"
4Ø PRINT "  BBBBBBB"
5Ø PRINT "      :"
END
```

regular. The progression, in time, is generally from less to greater regularity. At this stage of procedure, though, the

pattern form is in the student's head. It is the form, or equivalently the algorithm defining the form, that we want him to externalize into the computer. It is very easy to encourage him in this -- there are several advantages to it: fewer and shorter instructions are required; a single procedure can be written to generate a whole class of patterns; and, therefore, combination of patterns is simplified.

If one asks a student the simplest figure that can be generated on a teletype, the answer is nearly invariably a straight line. In fact, this is almost the only possible starting point, although many paths are subsequently possible. Thus,

```
TO MARK /CHARACTER/ /N/
1Ø TYPE /CHARACTER/
2Ø TEST IS /N/ 1
3Ø IF TRUE STOP
4Ø MARK /CHARACTER/ (DIFFERENCE /N/ AND 1)
END

←MARK "+" 8
+++++++++←                          (MARK, as written, does not
                                    produce a carriage return)
```

The MARK procedure above, or a similar procedure, can now be used to generate a variety of left-justified patterns, zigzags, and various geometric figures. A somewhat more fruitful approach is to embed MARK within a procedure SUPERMARK, which indents a given number of spaces before MARKing. The use of SUPERMARK makes it easy to "draw" figures, like diamonds and hexagons, having a vertical axis of symmetry. It is also useful when several figures of different sizes are to be stacked neatly.

```
TO SUPERMARK /N/ /LET/ /M/
1Ø MARK /BLANK/ /N/
2Ø MARK /LET/ /M/
3Ø PRINT ""                         (Carriage return)
END
```

SUPERMARK is very general, but requires three inputs each time it is used, which is inconvenient. Many students settle on a standard space about which to center their lines. For example, to center lines on the 19th column,

```
TO MIDDLE /N/ /CHAR/
1Ø SUPERMARK (DIFF 19 QUOTIENT /N/ 2) /CHAR/ /N/
END

←MIDDLE 8 "*"
                  *********
←MIDDLE 16 "#"
            ################
```

And now a "flood" of patterns ensues:

```
TO RECTANGLE /HEIGHT/ /WIDTH/ /CHAR/
1Ø TEST IS /HEIGHT/ Ø
2Ø IF TRUE STOP
3Ø MIDDLE /WIDTH/ /CHAR/
4Ø RECTANGLE (DIFF /HEIGHT/ 1) /WIDTH/ /CHAR/
END

←RECTANGLE 6 1Ø "?"
                ??????????
                ??????????
                ??????????
                ??????????
                ??????????
                ??????????

TO TRIANGLE /CHAR/ /WIDTH/ /CHANGE/ /HEIGHT/
1Ø TEST IS /HEIGHT/ Ø
2Ø IF TRUE STOP
3Ø MIDDLE /CHAR/ /WIDTH/
4Ø TRIANGLE /CHAR/ (SUM /WIDTH/ /CHANGE/) /CHANGE/
      (DIFF /HEIGHT/ 1)
END
```

In this last procedure, the students overshot their mark. They found, in trying TRIANGLE out, that they had, in fact, written a program which generated any trapezoid, symmetric about the 19th column!

```
←TRIANGLE "." 4 2 4
    ....
   ......
  ........
 ..........
```

and by using negative /CHANGE/, we can invert this.

```
←TRIANGLE "." 1Ø -2 4
 ..........
  ........
   ......
    ....
```

We can even use this procedure to make triangles!

```
←TRIANGLE "*" 1 4 6
      *
     ***
    *****
   *******
  *********
 ***********
```

These procedures are representative of student work at this level. To create more complex figures, say hexagons, there are now two courses. A student can continue to write a completely new procedure for each class of figures he wishes to generate. Or, he can realize (as most do eventually) that the shape procedures written up to that point can themselves be used with considerable savings in labor. Thus, following the writing of the TRIANGLE procedure given above, the same students wrote DIAMOND, again overshooting their mark.

```
TO DIAMOND /CHAR/ /WIDTH/ /CHANGE/ /HEIGHT/
1Ø TRIANGLE /CHAR/ /WIDTH/ /CHANGE/ /HEIGHT/
2Ø TRIANGLE /CHAR/ (SUM /WIDTH/ (PRODUCT
      (DIFF /HEIGHT/ 2) /CHANGE/)) /CHANGE/
      (DIFF /HEIGHT/ 1)
END
```

The "joining" of the two triangles (trapezoids) presents a mathematical problem of some complexity. Initial efforts yielded "diamonds" like

```
  **
 ****
******
  **
 ****
******
```

and

```
  **
 ****
******
  **
```

These initial attempts prodded the students into a more systematic, general approach to this problem and they solved it in very nearly algebraic terms. Not having any mathematical background beyond arithmetic, this was a considerable achievement both in terms of concepts developed and in results. Then,

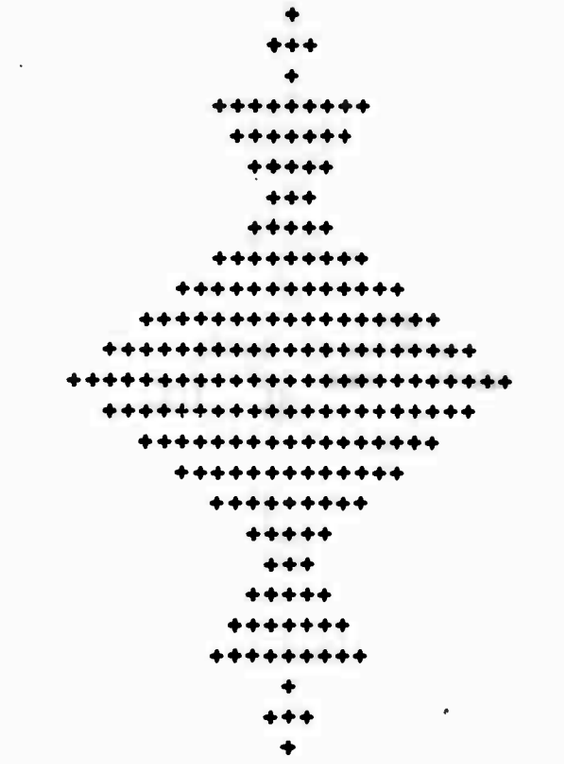
```
←DIAMOND "+" 3 4 4
      +++
    +++++++
  +++++++++++
+++++++++++++++
  +++++++++++++++
    +++++++++++
      +++++++
        +++
```

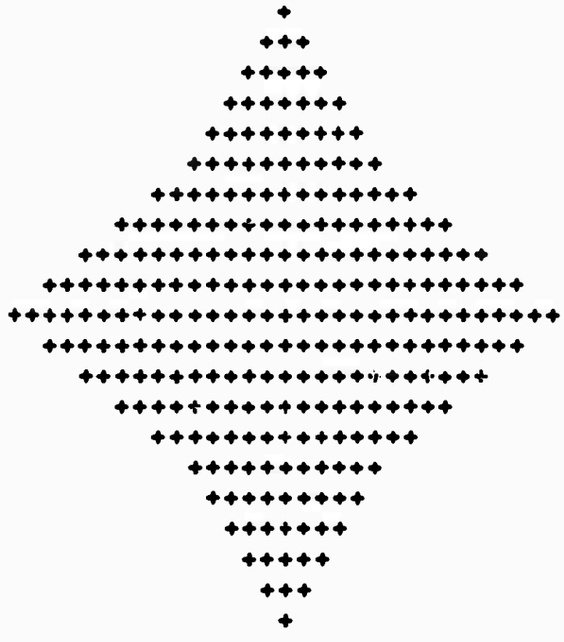
Such procedures, yielding polygons of various types, can themselves be combined but the only natural direction to go is vertical stacking. Since all procedures above the level of MARK draw figures in their entirety, and since there is no means of returning to a previous line under computer control, we cannot extend

our procedures to generate sets of figures which are next to each other. This is a serious deficiency of this current approach and its amelioration is discussed later. The following is typical of the stacking generated by the students.

```
←GLIRP 2 3 3 5
                    000
                    000
                    000
                     X
                    XXX
                   XXXXX
                  XXXXXXX
                 XXXXXXXXX
                  XXXXXXX
                   XXXXX
                    XXX
                     X
                    000
                    000
                    000
                     X
                    XXX
                   XXXXX
                  XXXXXXX
                 XXXXXXXXX
                  XXXXXXX
                   XXXXX
                    XXX
                     X
```

Stacking procedures included some of the type above, for which all parameters of the stack had to be specified as input. More interesting results were obtained by the use of the LOGO operation RANDOM to generate randomly-chosen patterns. Another idea, not found by the students, is to generate patterns with further constraints such as symmetry about a horizontal line. This leads to patterns like the ones below:





Some students choose not to automatically center their lines with the use of MIDDLE: they use SUPERMARK directly. This leads to a very different choice of pattern type, in fact the vertical stacking above is somewhat tedious when SUPERMARK is used directly. This is due to the fact that the use of SUPERMARK leads naturally to the inclusion of the indentation of a figure as an input parameter. But this extra degree of freedom, if systematically varied, gives new and interesting patterns. The student-written procedure STRIPE, for example, gives us:

```
←STRIPE 3 2
***
***
 ***
 ***
  ***
  ***
   ***
   ***
    \
     \
      ↘
```

Procedures such as STRIPE and GLIRP are the most advanced ones achievable from the MARK, SUPERMARK beginnings: it is difficult to extend these further (although many more procedures <u>at</u> this level can be written).

The reason that we are blocked at this level is that we can produce most of the basic figures of interest but we can only manipulate them in very simple ways. We can indent figures, that is to say, translate them horizontally. We can produce any desired vertical grouping of our basic figures*. That is about all, however. We cannot perform so simple an operation as making

*That is, we can superimpose patterns as long as they are on separate sets of rows.

a horizontal array containing several polygons. Our present approach forces us to print each figure line by line; we cannot have a procedure manipulating the figure as a whole, e.g., ROTATE (RECTANGLE "*" 3 4).

Thus, we start afresh from a completely different point of view, our goals and hence our methods being quite different from those we chose initially. We will now concentrate on the implementation of geometric transformations rather than on generation of specific figures. Although all earlier programs will be useless in this new approach, the algorithms developed for the generation of the various shapes of interest are easily reprogrammed.

This approach was only used very briefly at the end of the semester at U.Mass./Boston. In order to attain a reasonable level of achievement within the context of this material, some basic programs were given to the students to experiment with. The major really useful result gleaned from the students' experience with it was that enthusiasm and aptitude for this material ran very high and one might not expect this to be the case, considering the anti-mathematical prejudices of the majority of the class. The students used the translation and reflection procedures given to them with considerable insight and facility. The following teaching sequence based on geometric transformations follows along the lines studied at the end of the U.Mass./Boston course.

The first consideration in the implementation of geometric transformations must be that of data base. Probably the simplest choice is to represent a figure as a set of ordered pairs of numbers, corresponding to the Cartesian coordinates of the characters composing the pattern. A third character for each

point, to indicate the nature of the character at that point, adds both versatility and complexity. We will not do so here. A slightly unusual convention adopted in the following is that of numbering the vertical axis to increase in the downwards direction. This corresponds to the way the teletype goes to successive lines. The pairs can be represented in several ways in LOGO; certainly the simplest is to write them as LOGO sentences, the add elements being x-coordinates and even ones the corresponding y-coordinates.

Our first set of procedures will translate the computer's version of a figure into geometric form. These are fairly complex procedures, certainly more complex than the procedures for manipulation of pair lists, to be written later. For this reason, some teachers may choose to regard them as part of LOGO and not have the students write the drawing procedures.

Before we give the drawing procedures, we write three very generally useful procedures particularly applicable to pair list problems.

```
TO NTH /N/ /LIST/                    (Gives /N/th element of /LIST/)
1Ø TEST IS /N/ 1
2Ø IF TRUE OUTPUT FIRST /LIST/
3Ø OUTPUT NTH (DIFF /N/ 1)
     (BUTFIRST /LIST/)
END

TO DELETE /N/ /LIST/                 (Deletes first /N/ element from
                                     /LIST/)
1Ø TEST IS /N/ Ø
2Ø IF TRUE OUTPUT /LIST/
3Ø OUTPUT DELETE (DIFF /N/ 1)
     (BUTFIRST /LIST/)
END
```

```
TO PULL /N/ /LIST/                (Outputs first /N/ elements of
1Ø TEST IS /N/ Ø                   /LIST/ as a sentence)
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT SENTENCE
     FIRST /LIST/
     PULL (DIFF /N/ 1) (BUTFIRST /LIST/)
END
```

Thus, given a list of two pairs, /LIST/, "1 2 -1 -2" NTH 2 /LIST/ is 2, the second coordinate of the first pair; DELETE 2 /LIST/ deletes the first pair, and PULL 2 /LIST/ gives just the first pair.

We next write some general "drawing" procedures. They allow for an arbitrary choice of origin and marking character. PLOTP /POSITION/ /POINT/ /EDGE/ /CHAR/ is the basic procedure. It takes its present position as /POSITION/, then moves to /POINT/ and types /CHAR/. If /POINT/ is on a subsequent line and to the left of /POSITION/, PLOTP must first carriage return, then space across to get the right "x-coordinate". /EDGE/ is the x-value assigned to the left-hand column. If the point has already been passed, PLOTP outputs "FALSE", if successful, it outputs "TRUE".

```
TO PLOTP /POSITION/ /POINT/ /EDGE/ /CHAR/
1Ø TEST EITHER
     GREATERP NTH 2 /POSITION/ NTH 2 /POINT/
     GREATERP /EDGE/ FIRST /POINT/   (Can we plot /POINT/?)
2Ø IF TRUE OUTPUT "FALSE"            (If not, we output "FALSE")
3Ø TEST GREATERP
     FIRST /POSITION/                (Are we already too far to
     FIRST /POINT/                    the right?)
4Ø IF TRUE TYPE /CARRIAGE RETURN/
5Ø IF TRUE SPACE (DIFF FIRST /POINT/ (If so, return to margin and
     /EDGE/)                          space across suitably)
6Ø IF FALSE SPACE (DIFF FIRST /POINT/(Otherwise, move over from
     FIRST /POSITION/)                FIRST /POSITION/)
7Ø SKIP DIFF (NTH 2 /POINT/)         (Move vertically the requisite
     (NTH 2 /POSITION/)               number of rows)
8Ø TYPE /CHAR/
9Ø OUTPUT "TRUE"
END
```

SKIP /M/ and SPACE /N/ move the carriage vertically and horizontally /M/ and /N/ spaces.

```
TO SKIP /M/                              (Without carriage return)
1Ø TEST IS /M/ Ø
2Ø IF TRUE STOP
3Ø TYPE /LINE FEED/
4Ø SKIP (DIFF /M/ 1)
END
```

```
TO SPACE /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø TYPE /BLANK/
4Ø SPACE (DIFF /N/ 1)
END
```

PLOTP plots (or tries to plot) a single point. We incorporate this procedure within a higher level one, PLOTLIST /LIST/ /EDGE/ /CHAR/, which successively plots all but the first pair of /LIST/. /EDGE/ and /CHAR/ have the same meaning as in PLOTP. PLOTLIST plots the second pair on /LIST/ relative to the first one. If PLOTP is successful, then we eliminate the first pair and keep on. Otherwise, the second pair has not been plotted and we are still at the position of the first point. We therefore delete the second pair, and keep going.

```
TO PLOTLIST /LIST/ /EDGE/ /CHAR/
1Ø TEST GREATERP 3 (COUNT /LIST/)  (Is there only one pair left?)
2Ø IF TRUE STOP
3Ø TEST PLOTP
     SENTENCE (SUM FIRST /LIST/ 1) (We are already one space to the
              NTH 2 /LIST/          right of the first pair.  This
                                    is our position.)
     PULL 2 (DELETE 2 /LIST/)      (Plot second pair)
     /EDGE/
     /CHAR/
```

```
4Ø IF TRUE PLOTLIST                        (If second point is plotted,
     (DELETE 2 /LIST/) /EDGE/ /CHAR/        eliminate first point)
5Ø IF FALSE PLOTLIST                       (If not, eliminate the
     SENTENCE (PULL 2 /LIST/)               second pair, we are still
        (DELETE 4 /LIST/)                   at the position of the
        /LIST/ /EDGE/                       first pair.)
END
```

And now, we need only a top-level procedure, DRAW /LIST/ /ORIGIN/ /CHAR/. It prefaces /LIST/ with /ORIGIN/, makes /EDGE/ NTH 2 /ORIGIN/, and calls PLOTLIST.

```
TO DRAW /LIST/ /ORIGIN/ /CHAR/
1Ø PLOTLIST
     SENTENCE SENTENCE (DIFF FIRST /ORIGIN/ 1)
        NTH 2 /ORIGIN/ /LIST/           (PLOTLIST assumes we are 1
     NTH 2 /ORIGIN/                      square to the right of the
     /CHAR/                              first pair)
END
```

```
←MAKE "VERTICAL LINE" "3 Ø 3 1 3 2 3 3 3 4"
←DRAW /VERTICAL LINE/ "Ø Ø" "+"
   +
   +
   +
   +
   +←    (no carriage return)
←MAKE "TRIANGLE" "2 Ø 1 1 2 1 3 1 Ø 2 1 2 2 2 3 2 4 2"
←DRAW /TRIANGLE/ "Ø Ø" "?"
  ?
 ???
?????←
```

We find, however, that the inability of the teletype to return to previous lines severely limits our drawing ability.

```
←MAKE "BOTH" SENTENCE OF
     /TRIANGLE/
     /VERTICAL LINE/
```

```
←DRAW /BOTH/ "Ø Ø" "*"
  *
 ***
*****
   *
   *
←PRINT /BOTH/
2 Ø 1 1 2 1 3 1 Ø 2 1 2 2 2 3 2 4 2 3 Ø 3 1 3 2 3 3 3 4
```

The top point of the vertical line, 3 1, is too late in the list to be marked.

Thus, a second important program is required to put lists of pairs in proper order. Without such an ordering procedure we cannot combine figures, or even transform them in some ways (like rotating them).

We first write a procedure ADDLISTS which combines two ordered lists, giving the correct order for their union.

```
TO ADDLISTS /LIST1/ /LIST2/
1Ø TEST EITHER
     EMPTYP /LIST1/                    (If either list is empty,
     EMPTYP /LIST2/                     output the other)
2Ø IF TRUE OUTPUT (SENTENCE /LIST1/ /LIST2/
3Ø TEST IS (PULL 2 /LIST1/) (PULL 2 /LIST2/)
4Ø IF TRUE OUTPUT SENTENCE             (If the first pair of /LIST1/
     PULL 2 /LIST1/                    and /LIST2/ are identical,
     ADDLISTS (DELETE 2 /LIST1/)       output this element (once)
              (DELETE 2 /LIST2/)       and repeat with it deleted from
                                       both /LIST1/ and /LIST2/)
5Ø TEST EITHER
     GREATERP (NTH 2 /LIST2/)          (First element of second list
              (NTH 2 /LIST1/)          is lower)
     BOTH
      (IS NTH 2 /LIST2/) (NTH 2 /LIST1/) (First elements in same row,
      GREATERP (FIRST /LIST2/)         first element of second is
               (FIRST /LIST1/)         rightmost)
6Ø IF TRUE OUTPUT SENTENCE
     PULL 2 /LIST1/
     ADDLISTS (DELETE 2 /LIST1/) /LIST2/
7Ø OUTPUT SENTENCE
     PULL 2 /LIST2/
     ADDLISTS /LIST1/ (DELETE 2 /LIST2/)
END
```

Then, using ADDLISTS, a procedure ORDER can be written. ORDER repeatedly decomposes its input into halves until there are at most two pairs in each piece. ORDER then uses ADDLISTS to join these sublists, placing their elements in the right order.

```
TO ORDER /LIST/
1Ø TEST GREATERP 5 (COUNT /LIST/)
2Ø IF TRUE OUTPUT ADDLISTS
     (PULL 2 /LIST/)
     (DELETE 2 /LIST/)
3Ø OUTPUT ADDLISTS OF
     ORDER PULL (EVENHALF /LIST/) /LIST/
     ORDER DELETE (EVENHALF /LIST/) /LIST/
END
```

Where EVENHALF /LIST/ is the closest integer to half of the count of /LIST/. (We don't want a list of 3 pairs separated into 2 triples.)

```
TO EVENHALF /LIST/
1Ø OUTPUT PRODUCT 2 QUOTIENT (COUNT /LIST/ 4)
END
```

```
←PRINT ORDER /BOTH/
2 Ø 3 Ø 1 1 2 1 3 1 Ø 2 1 2 2 2 3 2 4 2 3 3 3 4
 DRAW (ORDER /BOTH/) "Ø Ø" "+"
  ++
 +++
+++++
   +
   +
```

Now we can address ourselves to the more interesting (and easier) problems of manipulating pair lists. This is perhaps the best point in this sequence for average students to start writing their own programs. Translating a figure by /ACROSS/ units horizontally and /VERTICAL/ units vertically involves simply adding /ACROSS/ to each first coordinate of the points constituting the figure and /VERTICAL/ to each second coordinate.

```
TO TRANSLATE /FIGURE/ /ACROSS/ /VERTICAL/
1Ø TEST EMPTYP /FIGURE/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT SENTENCE SENTENCE
     SUM (FIRST /FIGURE/) /ACROSS/
     SUM (NTH 2 /FIGURE/) /VERTICAL/
     TRANSLATE (DELETE 2 /FIGURE/) /ACROSS/ /VERTICAL/
END
```

The procedure ADDLISTS can be used to combine figures:

```
←MAKE "TRIANGLE TWO" TRANSLATE /TRIANGLE/ 6 Ø
←MAKE "TWO TRIANGLES" ORDER ADDLISTS /TRIANGLE/ /TRIANGLE TWO/
←DRAW /TWO TRIANGLES/ "Ø Ø" "+"
   +       +
  +++     +++
 +++++   +++++
```

To write any such transformation procedure, we need only specify the action on the first point of the list. Simple recursion can then repeat this action on subsequent pairs until the list is exhausted. To reflect a figure about any vertical line /L/ units from the origin, for example,

```
TO REFLECTVERT /L/ /LIST/
1Ø TEST EMPTYP /LIST/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT SENTENCE SENTENCE
     (DIFF /L/ FIRST /LIST/)
     NTH 2 /LIST/
     REFLECTVERT (DELETE 2 /LIST/)
END
```

We write in just this manner:

```
REFLECTHOR /L/ /LIST/        (reflects pairs on /LIST/ about
                              horixontal /L/)
REFLECTORIGIN /LIST/         (reflects /LIST/ through the origin by
                              simply multiplying every number on
                              /LIST/ by -1)
REFLECT45 /LIST/             (reflects /LIST/ about the line 45° to
                              the horizontal by interchanging the
                              coordinates of each pair)
```

Rotation is just as easy from a programming point of view, but, because the formula giving the new coordinates in terms of the old ones involves some trigonometry, it is more difficult for many students. A table of sines and cosines for angles at 15° increments is adequate, since the "graininess" of the teletype gives smaller rotations an extremely uneven character.

The above include all transformation procedures given to the U.Mass. students. There were, in addition, three non-transformational procedures they could use. They could, for example, use ADDLISTS to take the union of two figures. It is also interesting to find the intersection of two figures.

```
TO CONTAINSP /PAIR/ /LIST/      (Tests if /LIST/ contains /PAIR/)
1Ø TEST EMPTYP /LIST/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø TEST BOTH
     IS (FIRST /PAIR/) (FIRST /LIST/)
     IS (NTH 2 /PAIR/) (NTH 2 /LIST/)
4Ø IF TRUE OUTPUT "TRUE"
5Ø OUTPUT CONTAINSP /PAIR/ (DELETE 2 /LIST/)
END
```

```
TO INTERSECT /LISTA/ /LISTB/    (Gives intersection of /A/ and /B/)
1Ø TEST EITHER EMPTYP /LISTA/
     EMPTYP /LISTB/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø TEST CONTAINSP (PULL 2 /LISTA/) /LISTB/
4Ø IF TRUE OUTPUT SENTENCE
     PULL 2 /LISTB/
     INTERSECT
        DELETE 2 /LISTB/
        /LISTA/
5Ø OUTPUT INTERSECT
     DELETE 2 /LISTB/
     /LISTA/
END
```

And now a variety of set theoretic operations can be constructed such as symmetric difference, complement, etc.

All the procedures described as part of this second, Cartesian, drawing sequence were given to the students. They then spent the little time left in the term familiarizing themselves with these:

```
←MAKE "DIAGONAL" "1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8"
←DRAW /DIAGONAL/ "1 1" "+"
+
 +
  +
   +
    +
     +
      +
       +
        +

←MAKE "DIAGONAL2" TRANSLATE /DIAGONAL/ "5 Ø"
←DRAW /DIAGONAL2/ "1 1" ":"
      :
       :
        :
         :
          :
           :
            :
             :

←MAKE "D3" ADDLISTS /DIAGONAL/ /DIAGONAL2/
←DRAW /D3/ "1 1" "⁜"
⁜     ⁜
 ⁜     ⁜
  ⁜     ⁜
   ⁜     ⁜
    ⁜     ⁜
     ⁜     ⁜
      ⁜     ⁜
       ⁜     ⁜
        ⁜     ⁜
←  etc.
```

In only one case was a new program written by the students, built upon what was given them:

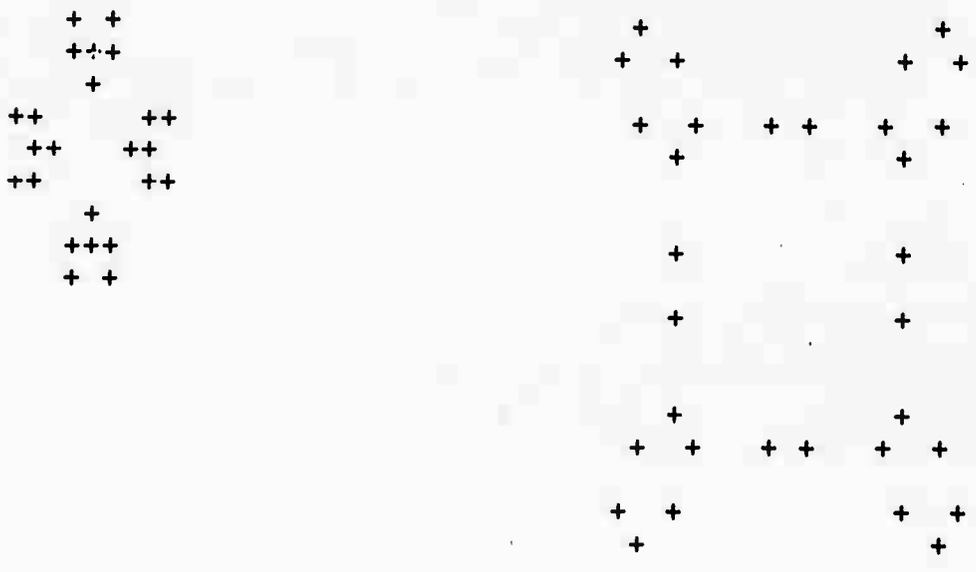
```
TO MOVE /FIGURE/ /NUMBER/
1Ø TEST IS /NUMBER/ Ø
2Ø IF TRUE STOP
3Ø DRAW /FIGURE/ "Ø Ø" "*"
4Ø MAKE "FIGURE" TRANSLATE /FIGURE/ "2 2"
5Ø MOVE /FIGURE/ (DIFF /NUMBER/ 2)
END

←MAKE "BOX" "Ø Ø 1 Ø Ø 1 1 1"
←MOVE /BOX/ 4
**
**

   **
   **

      **
      **

         **
         **
←
```

This is a very rudimentary animation. On this note the term ended.

There are a large number of things to do at this level which the students were working on as the term ended. Also, there are a number of very interesting extensions. For example, the combination of random figure generation with reflections produces interesting symmetries.

```
TO EIGHTFOLD /N/                    (/N/ is the number of pairs on the
                                    pair list that will be generated)
1Ø MAKE "PAIR LIST" RANDOMLIST      (RANDOMLIST is user-written and
     OF /N/                         generates a random list of /N/
                                    pairs)
2Ø MAKE "PAIR LIST"                 (Form the union of /PAIR LIST/ and
    ADDLISTS OF (/PAIR LIST/)       the pair list formed by reflecting
     AND (REFLECT45 OF              it around the 45 degree line, and
     /PAIR LIST/)                   make this the new /PAIR LIST/)
3Ø MAKE "PAIR LIST"                 (Form the union of the new list
    ADDLISTS OF (/PAIR LIST/)       and its reflection about the
     AND (REFLECTY OF               Y-axis)
     /PAIR LIST/ AND Ø)
4Ø MAKE "PAIR LIST"                 (Do the same with the resulting
    ADDLISTS OF (/PAIR LIST/)       list and its reflection about the
     AND (REFLECTX OF /PAIR         X-axis)
     LIST/ AND Ø)
5Ø DRAW ORDER OF /PAIR LIST/        (Put the resulting pair list in
     "+"                            lexicographic order and draw it
                                    using +'s)
END
```

EIGHTFOLD generates random drawings such as the following.



Another interesting extension begins with a simple procedure which enables a user to type in a figure, pointillistically, the procedure converting it to a pair list. One can then write programs which determine if two given figures are congruent, or geometrically similar.

A student might choose, instead, to study more complex transformations such as uniform or nonuniform changes of metric. This leads into yet another rich area of study.

4.2 Language Sequence

Most students find the automatic random generation of poetry and prose forms of great interest. Work in this area is especially beneficial for the average student who considers formation of algorithms and problem-solving as skills associated exclusively with mathematics and the sciences. His discovery that these skills are equally applicable to problems related to language and discourse is, therefore, an important one. The sequence presented here, an automatic generation of word-forms, is based on teaching done at U.Mass./Boston in the spring of 1971. The conduct of the course and a description of the students was given previously. Programs and examples are taken from student work over the course of about three weeks.

The first step in randomly generating word forms is to write a procedure R-CHOOSE, which outputs an element chosen at random from the list given as its input. We need, as a subprocedure, one which removes the element in a given position on a given list.

```
TO CHOOSE /N/ /LIST/
1Ø TEST IS /N/ 1
2Ø IF TRUE OUTPUT (FIRST /LIST/)
3Ø OUTPUT CHOOSE (DIFF /N/ 1) (BUTFIRST /LIST/)
END

←CHOOSE 3 "ABRACADABRA"
R
```

We also need a procedure which uses the built-in random digit generator, RANDOM, to generate random numbers between 1 and a given upper limit. To do this, we first write a procedure RND which produces a random number of the requisite number of digits, and then RAND, which keeps on using RND until the number obtained lies in the right range.

```
TO RND /# DIGITS/
1Ø TEST IS /# DIGITS/ 1
2Ø IF TRUE OUTPUT RANDOM
3Ø OUTPUT WORD OF
     RANDOM
     RND (DIFF /# DIGITS/ 1)
END


TO RAND /NUMBER/
1Ø MAKE "DIGITS" (COUNT /NUMBER/)
2Ø MAKE "TRIAL" RND OF /DIGITS/
3Ø TEST BOTH
     GREATERP /TRIAL/ Ø
   AND EITHER
     GREATERP /NUMBER/ /TRIAL/
     IS /NUMBER/ /TRIAL/
4Ø IF TRUE OUTPUT /TRIAL/
5Ø IF FALSE OUTPUT RAND /NUMBER/
END

←PRINT RAND 3
2
←PRINT RAND 3
2
←PRINT RAND 3
1
←PRINT RAND 34567
29843
←
```

Now, R-CHOOSE is easy.

```
TO R-CHOOSE /LIST/
1Ø OUTPUT CHOOSE (RAND COUNT /LIST/) /LIST/
END

←PRINT R-CHOOSE "GOATS SHEEP COWS"
SHEEP
←PRINT R-CHOOSE "A O V P A"
A
```

One can now make up lists for each of the main parts of speech and use R-CHOOSE with these as input:

```
←MAKE "VERBS" "APPEARS WAS SMELLS GROWS LIVES DEVELOPS MOVES
STAGGERS SEEMS FLOATS STANDS DIES SMOKES DECAYS SMILES YAWNS
CHEWS PRE-REGISTERS FLUNKS-OUT GROOVES"

←MAKE "NOUNS" "TREE GRASS LONNIE RAVEN SUMMER ROCK BILLBOARD
MOUNTAIN WATER COMPUTER WINDOW CAVE SOCK PAVEMENT DIRT ELEVATOR
CARROT WITCH MOON WORLD"

←MAKE "ADVERBS" "SLOWLY QUICKLY SMOOTHLY NOISILY QUIETLY ANGRILY
HAPPILY PROFUSELY DEJECTEDLY KNOWINGLY SUSPICIOUSLY BRILLIANTLY
SEEMINGLY GRACEFULLY STUPIDLY ABRUPTLY PATIENTLY WILLINGLY
FORCEFULLY PEACEFULLY"

←MAKE "ADJECTIVES" "FAT LAZY GREEN DUMB COOL DANK FLUID
COMPLICATED MEAN FLAMING UGLY HARSH LUMINOUS SWEATY HUNGRY
DRUNK DEGENERATE SOFT DRY HUGE"
```

The number of such lists is dependent on the imagination and sophistication of the students. The creation of general compound sentences is not possible with just the lists given. Also, by making lists which apply only in certain situations, semantic distinctions can be made. For example, we could have /PEOPLE ADJECTIVES/ be "FAT THIN TALL SHORT LAZY HAPPY INDUSTRIOUS".

In any case, the use of R-CHOOSE with lists like the above makes the generation of simple word forms easy. For example:

```
TO POEM
1Ø OUTPUT SENTENCE SENTENCE SENTENCE SENTENCE
     "THE"
     R-CHOOSE /ADJECTIVES/
     R-CHOOSE /NOUNS/
     R-CHOOSE /ADVERBS/
     R-CHOOSE /VERBS/
END


TO POEM-1
1Ø PRINT POEM
2Ø PRINT POEM
3Ø PRINT POEM
4Ø PRINT POEM
5Ø PRINT POEM
END


←POEM-1
THE DANK CAVE SUSPICIOUSLY STANDS
THE DUMB WITCH PROFUSELY PRE-REGISTERS
THE MEAN ELEVATOR FORCEFULLY SMOKES
THE SOFT WITCH KNOWINGLY MOVES
THE DUMB COMPUTER QUIETLY YAWNS
```

The random verse generating procedures can now be extended in any of several ways. More complex sentence forms can be produced if additional parts of speech are taken into account in the same way as the four already treated. Semantic connections can be established by making lists containing appropriate associations. For example, as mentioned before,

```
←MAKE "NAMES" "JOHN JACK FRED"
←MAKE "PEOPLE ADJECTIVES" "FAT THIN TALL SHORT LAZY HAPPY
  INDUSTRIOUS"

TO DESCRIPTION
1Ø PRINT SENTENCE
     R-CHOOSE /PEOPLE ADJECTIVES/
     R-CHOOSE /NAMES/
END

←DESCRIPTION
FAT JACK
←DESCRIPTION
LAZY FRED
```

Another extension is the generation of verse with some metric or other constraints. This can be done with relative ease, again by separating the parts of speech into classes. If the verse form is broken, then separation according to number of syllables is required. Meter requires similar, though more complex, considerations.

The U.Mass. students writing the poetry sequence discussed here decided to take yet another problem, that of producing rhymed verse. They looked at the blank verse they were generating and realized that each line ended with a verb. It was, therefore, only necessary to select rhyming verbs. This was done by having /VERBS/ a list of *names for classes* of rhyming verbs, rather than of the verbs themselves.

Thus, they made the associations:

```
/B/ IS "MAKES TAKES BREAKS FLAKES WAKES"
/C/ IS "FLIES LIES DRIES PRIES DIES"
/D/ IS "SINGS BRINGS FLINGS SPRINGS RINGS"
/BB/ IS "GROWS BLOWS SNOWS GOES KNOWS"

/VERBS/ IS "B C D BB"

TO TWORHYME
1Ø MAKE "ZZ" R-CHOOSE /VERBS/
2Ø PRINT SENTENCE SENTENCE SENTENCE SENTENCE
     "THE"
     R-CHOOSE /ADJECTIVES/
     R-CHOOSE /NOUNS/
     R-CHOOSE /ADVERBS/
     R-CHOOSE (THING OF /ZZ/)
3Ø PRINT SENTENCE SENTENCE SENTENCE SENTENCE
     "THE"
     R-CHOOSE /ADJECTIVES/
     R-CHOOSE /NOUNS/
     R-CHOOSE /ADVERBS/
     R-CHOOSE (THING OF /ZZ/)
END
```

```
TO POEM-RHYME
1Ø TWORHYME
2Ø TWORHYME
3Ø TWORHYME
END

←POEM-RHYME
THE LAZY ELEVATOR PATIENTLY TAKES
THE HUGH ROCK SMOOTHLY FLAKES
THE FAT WINDOW SUSPICIOUSLY RINGS
THE FLUID DIRT QUICKLY RINGS
THE DANK SOCK FORCEFULLY BREAKS
THE HARSH SOCK KNOWINGLY MAKES
←
```

4.3 Turtle Sequence

We have developed a remote-controlled vehicle, the "turtle", which responds to a set of motion commands embedded within LOGO. This section will deal with its use at an elementary level. Use of the turtle in introductory classroom work provides a strong motivational factor, but a more important result is the introduction of new classes of algorithms especially useful for unsophisticated beginners. Such students will often find it easier to develop algorithms and write LOGO programs for "concrete" problems like traversing a given pattern, than to deal with the "abstract" simple string manipulation problems which serve as an introduction in the absence of the turtle.

Following a description of the turtle, we give a sequence of programs, centered on use of the turtle, which show a natural, gradual progression from the most rudimentary algorithms (and LOGO programs) to quite sophisticated ones. This sequence has not yet been comprehensively used in a classroom situation as have the two preceding. Preliminary results, however, based on short-term use of the turtle by single students, indicate that the sequence is realistic and engaging.

Finally, we will briefly discuss some of the many possibilities opened up by use of feedback from the turtle through the operation of various sensing devices. The material presented deals exclusively with touch sensors. Preliminary forms of such sensors have already been implemented on our turtle.

The Turtle

The turtle's "skin" consists of a shallow cylinder three inches high, mounted on two wheels and two ball bearings, surmounted by a transparent hemispherical dome 12 inches in diameter. Details of the design and construction of the turtle and associated interfaces are given in Section 5.2. It has a repertoire of five actions, performed upon execution of corresponding no-input LOGO commands:

FRONT	turtle moves forward 4 inches
BACK	turtle moves backwards 4 inches
RIGHT	turtle rotates 15° clockwise
LEFT	turtle rotates 15° counterclockwise
HORN	turtle toots

The touch sensors currently used are two thin wires bent around the front of the turtle like insect antennae. They are sufficiently far from the body that the possibility of contact can be discovered before the turtle actually collides. Touching of an object causes one of two flags to be set, depending on which sensor was activated. The no-input operations TOUCH LEFT and TOUCH RIGHT output the states of the flags and reset them to FALSE.

First Steps

The very simplest work with the turtle consists of typing direct commands:

```
←FRONT     (moves ahead one step)
←RIGHT     (
←RIGHT     (turns 45° clockwise)
←RIGHT     (
←FRONT     (moves ahead one step)
```

The sequence above has the turtle travel in a "knight move".

This use of the turtle can soon be supplanted by the writing of simple turtle procedures. At first, these will use the basic turtle commands directly:

```
TO EL
1Ø FRONT
2Ø FRONT
3Ø FRONT
4Ø FRONT
5Ø FRONT
6Ø RIGHT      (each RIGHT is 15°)
7Ø RIGHT
8Ø RIGHT
9Ø RIGHT
1ØØ RIGHT
11Ø RIGHT
12Ø FRONT
13Ø FRONT
END
```

It is immediately apparent that the small quanta of rotation (15°) and of linear travel (4 inches) necessitate a large number of instructions even for modest patterns. An easy way to much reduce this labor is to define a new set of basic motions in terms of procedures with inputs.

```
TO RIGHTTURN /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø RIGHT
4Ø RIGHTTURN (DIFF /N/ 1)
END

TO LEFTTURN /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø LEFT
4Ø LEFTTURN (DIFF /N/ 1)
END

TO FRONTS /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø FRONT
4Ø FRONTS (DIFF /N/ 1)
END

TO BACKS /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø BACK
4Ø BACKS (DIFF /N/ 1)
END
```

These new procedures are useful in defining "large patterns",

```
TO BIGELL
1Ø FRONTS 1Ø
2Ø RIGHTTURN 6
3Ø FRONTS 2Ø
END
```

but, more important, they are useful in defining procedures which allow variations in execution

```
TO ELL /N/
1Ø FRONTS /N/
2Ø RIGHTTURN 6
3Ø FRONTS (PRODUCT 2 /N/)
END
```

Or, to trace a square of side /N/,

```
TO SQUARE /N/
1Ø FRONTS /N/
2Ø RIGHTTURN 6
3Ø FRONTS /N/
4Ø RIGHTTURN 6
5Ø FRONTS /N/
6Ø RIGHTTURN 6
7Ø FRONTS /N/
8Ø RIGHTTURN 6
END
```

At any point the student is free to design his own tools. He may very well notice, in the course of writing polygon tracing procedures such as the above, that a linear motion is always followed by a turn. Thus, a useful "tool" is

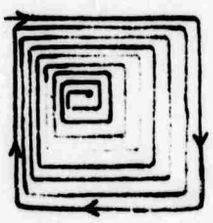
```
TO ELLL /LENGTH/ /N/          (Repeats a forward motion and right
1Ø TEST IS /N/ Ø               turn /N/ times)
2Ø IF TRUE STOP
3Ø FRONTS /LENGTH/
4Ø RIGHTS 6
5Ø ELLL /LENGTH/ (DIFF /N/ 1)
END
```

Tracing a square in terms of ELLL is very easy.

```
TO SQUARE /SIDE/
1Ø ELLL /SIDE/ 4
END
```

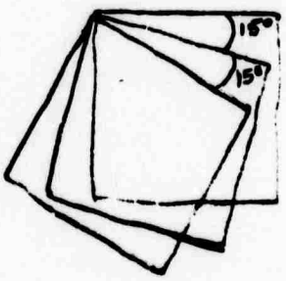
Such patterns as square can themselves be used as parts of more complex patterns.

```
TO SQUIRAL /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø SQUARE /N/
4Ø SQUIRAL (DIFF /N/ 1)
END
```



```
TO PRECESS /SIDE/ /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø SQUARE /SIDE/
4Ø RIGHTS 1
5Ø PRECESS /SIDE/ (DIFF /N/ 1)
END
```

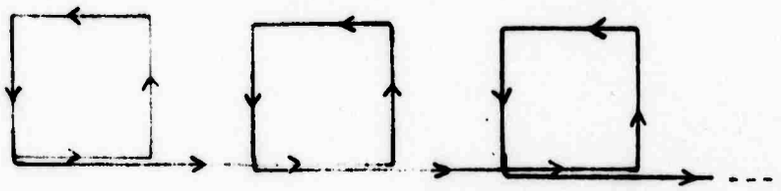
PRECESS gives the following sort of figure:



Many other procedures can be written which transform tracings of primitive figures:

```
TO LOOP /S/ /N/
1Ø SQUARE /S/
2Ø FRONTS /S/
3Ø LOOP (DIFFERENCE /N/ 1)
END
```

which gives a path like



More general figurations are obtained by writing transformation procedures which have the name of the primitive shape as an input. For example, we can generalize PRECESS in this way:

```
TO PRECESS /SHAPE/ /SIDE/ /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE STOP
3Ø DO SENTENCE OF /SHAPE/ AND /N/*
4Ø RIGHTTURN 1
5Ø PRECESS /SHAPE/ /SIDE/ (DIFFERENCE /N/ 1)
END
```

SQUIRAL and LOOP are easily generalized in just the same way.

---

*DO is a LOGO command which results in its one input being executed as a LOGO command. DO is useful in cases where, as here, some procedure name is not specified within the procedure being written.

Programs With "Memory"

A somewhat different, complementary approach is to write procedures which interact with the user. A simple, somewhat amusing procedure of this form is:

```
TO CONTRARY
1Ø DO OPPOSITE OF REQUEST
2Ø CONTRARY
END
```

CONTRARY uses the procedure OPPOSITE to do just the reverse of the typed instructions. (These instructions are assumed to start with one of the commands FRONTS, BACKS, RIGHTTURN, or LEFTTURN.)

```
TO OPPOSITE /COMMAND/
1Ø TEST IS FIRST OF /COMMAND/ "FRONTS"
2Ø IF TRUE OUTPUT SENTENCE "BACKS"
     BUTFIRST /COMMAND/
3Ø TEST IS FIRST OF /COMMAND/ "BACKS"
4Ø IF TRUE OUTPUT SENTENCE "FRONTS"
     BUTFIRST /COMMAND/
5Ø TEST IS FIRST OF /COMMAND/ "RIGHTTURN"
6Ø IF TRUE OUTPUT SENTENCE "LEFTTURN"
     BUTFIRST /COMMAND/
7Ø TEST IS FIRST OF /COMMAND/ "LEFTTURN"
8Ø IF TRUE OUTPUT SENTENCE "RIGHTTURN"
     BUTFIRST /COMMAND/
9Ø EXIT SENTENCE "I DON'T KNOW THE     (Exit if the starting command
     OPPOSITE OF" /COMMAND/            is not one of the four above)
END

←PRINT OPPOSITE "RIGHTTURN 3"
LEFTTURN 3
←
```

Another, more generally useful, interactive program is one which both executes the user's typed-in commands and forms a list of

them, outputting the completed list when the command END is encountered. This gives the turtle a sort of memory and transformations of various kinds can then be applied to these "memorized" paths. Commas are used to separate commands. (Blanks are not adequate for this purpose, since a turtle command may have an input.)

```
TO REMEMBER
1Ø MAKE "MOVE" REQUEST
2Ø TEST IS /MOVE/ "END"
3Ø IF TRUE OUTPUT /EMPTY/
4Ø DO /MOVE/
5Ø OUTPUT SENTENCE SENTENCE
     /MOVE/ "," REMEMBER
END
```

This procedure is used as follows

```
←PRINT REMEMBER
*SQUARE 4                    (Turtle traces a square of side 4)
*FRONTS 1Ø                   (Turtle moves forward 10 steps)
*END
SQUARE 4 , FRONTS 1Ø ,
←
```

To make use of such lists of memorized moves, we need an easy means of extracting the first command and also of obtaining the part of the list remaining. This is easily done with two new procedures. Each searches for the first comma, but otherwise they act differently. They are the analogues of FIRST and BUTFIRST for our new data structure.

```
TO FIRSTCOM /LIST/
1Ø TEST IS FIRST /LIST/ ","
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT SENTENCE OF
     FIRST /LIST/
     FIRSTCOM (BUTFIRST /LIST/)
END
```

```
←PRINT FIRSTCOM "FRONTS 2 , BACKS 4 , RIGHTTURN 3 ,"
FRONTS 2
←
```

```
TO BUTFIRSTCOM /LIST/
1Ø TEST IS FIRST /LIST/ ","
2Ø IF TRUE OUTPUT BUTFIRST /LIST/
3Ø OUTPUT BUTFIRSTCOM (BUTFIRST /LIST/)
END
```

```
←PRINT BUTFIRSTCOM "FRONT 2 , BACKS 4 , RIGHTTURN 3 ,"
BACKS 4 , RIGHTTURN 3 ,
←
```

With these two procedures, we can reverse any path given by a list of commands -- replacing each command by its opposite and reversing the order in which they appear.

```
TO REVERSE /PATH/
1Ø TEST IS /PATH/ /EMPTY/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT SENTENCE SENTENCE
     REVERSE BUTFIRSTCOM /PATH/
     OPPOSITE (FIRSTCOM /PATH/)
     ","
END
```

```
←PRINT REVERSE "FRONTS 3 , RIGHTTURN 2 , FRONTS 5 ,"
BACKS 5 , LEFTTURN 2 , BACKS 3 ,
←
```

A procedure which returns the turtle to its original position after having executed any number of typed commands is easy to write, given the above procedures

```
TO RETURN
1Ø EXECUTE REVERSE REMEMBER
END
```

where the procedure EXECUTE performs a series of LOGO commands, separated by commas:

```
TO EXECUTE /LIST/
1Ø TEST EMPTYP /LIST/
2Ø IF TRUE STOP
3Ø DO FIRSTCOM /LIST/
4Ø EXECUTE BUTFIRSTCOM /LIST/
END
```

An example of the use of RETURN is:

```
←RETURN
:FRONTS 7          (Turtle moves forward 7)
:LEFTTURN 3        (Turtle turns counterclockwise 45°)
:FRONTS 4          (Turtle moves forward 4)
←                  (At this point the turtle moves backwards 4,
                    turns right 45° and goes backwards 7,
                    finishing at its initial position)
```

One can extend the manipulation of paths given as lists of commands considerably further. Possibilities for extension are creation of paths symmetric in different ways with respect to the given path, use of the given path in area-covering procedures, etc. We turn, however, to yet another topic -- the automatic generation of procedures corresponding to given paths.

Procedure-Writing Procedures

The procedure CREATE, given below, writes a procedure with name /PROCEDURE NAME/, which traces out the path given by /PATH/. Note again that the LOGO command DO simply executes its one input as a complete LOGO instruction line.

```
TO CREATE /PROCEDURE NAME/ /PATH/
1Ø DO SENTENCE "TO" /PROCEDURE NAME/
2Ø CREATESTEPS /PATH/ 1Ø
3Ø DO "END"
END
```

```
TO CREATESTEPS /PATH/ /N/
1Ø TEST IS /PATH/ /EMPTY/
2Ø IF TRUE STOP
3Ø DO SENTENCE /N/ FIRSTCOM /PATH/
4Ø CREATESTEPS (BUTFIRSTCOM /PATH/) (SUM /N/ 1Ø)
END
```

Then, for example,

```
←CREATE "ELL" "FRONTS 3 , LEFTTURN 6 ,"   (and the procedure ELL
←LIST ELL                                  has been created.)

TO ELL
1Ø FRONTS 3
2Ø LEFTTURN 6
END
←
```

Much more elegant program generating procedures are possible. We assume each command on the input list has one input which must be given explicitly, since the commands are executed as direct lines. Then by including a dummy variable on the title line of the procedure being defined, and by multiplying all linear motions by that variable, we "generalize" the given path. Such a general procedure traces all paths which are either identical to the given one or larger than it by integral factors. Thus,

```
TO GENERALIZE /NAME/ /PATH/
1Ø DO SENTENCE SENTENCE "TO" /NAME/ "/J/"
2Ø GENERALSTEPS 1Ø /PATH/
3Ø DO "END"
END

TO GENERALSTEPS /N/ /PATH/
1Ø TEST EMPTYP /PATH/
2Ø IF TRUE STOP
3Ø TEST EITHER
     IS FIRST /PATH/ "LEFTTURN"      (If command is a turn,
     IS FIRST /PATH/ "RIGHTTURN"      enter it unchanged)
```

```
4Ø IF TRUE DO SENTENCE
     /N/
     FIRSTCOM /PATH/
5Ø IF FALSE DO SENTENCE SENTENCE
     /N/                    (Line # of first command on /PATH/)
     FIRST /PATH/           (Command)
     "PRODUCT OF"
     FIRST BUTFIRST /PATH/    (Input of first command of path)
     "AND /J/"
6Ø GENERALSTEPS (SUM /N/ 1Ø) (BUTFIRSTCOM /PATH/)
END
```

We use GENERALIZE to create general paths patterned on specific ones.

```
←GENERALIZE "TRIANGLE" "FRONTS 1 , RIGHTTURN 8 , FRONTS 1 ,
 RIGHTTURN 8 , FRONTS 1 , RIGHTTURN 8 ,"
←LIST TRIANGLE

TO TRIANGLE /J/
1Ø FRONTS PRODUCT OF 1 AND /J/
2Ø RIGHTTURN 8
3Ø FRONTS PRODUCT OF 1 AND /J/
4Ø RIGHTTURN 8
5Ø FRONTS PRODUCT OF 1 AND /J/
6Ø RIGHTTURN 8
END
```

And we now have a "general" triangle tracing procedure.

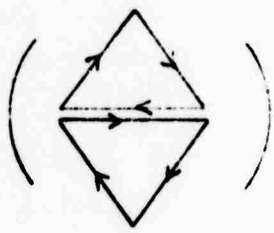
Furthermore, procedures created by GENERALIZE (as well as others of suitable form) can themselves be used as part of generalizable procedures. For example, the procedure TRIANGLE above can be used as part of a diamond-drawing procedure. To make the use of GENERALIZE easier, we embed it within a procedure ACCEPT, which builds up the input to GENERALIZE by means of REQUESTS; performing the typed commands as well:

```
TO ACCEPT /NAME/
1Ø GENERALIZE /NAME/ ACCEPTSTEPS
END
```

```
TO ACCEPTSTEPS
1Ø MAKE "MOVE" REQUEST
2Ø TEST IS /MOVE/ "END"
3Ø IF TRUE OUTPUT /EMPTY/
4Ø DO /MOVE/
5Ø OUTPUT SENTENCE SENTENCE
     /MOVE/
     ","
     ACCEPTSTEPS
END
```

Thus,

```
←ACCEPT "DIAMOND"
*TRIANGLE 1
*RIGHTTURN 4
*TRIANGLE 1
*END
←LIST DIAMOND

TO DIAMOND /J/
1Ø TRIANGLE PRODUCT 1 AND /J/
2Ø RIGHTTURN 4
3Ø TRIANGLE PRODUCT 1 AND /J/
END
```



The Use of Sensors

Thus far we have simply been using the turtle as an alternative output device. This kind of use restricts work with the turtle to the contexts described in the earlier sections. Adding sensors to the turtle enables it to interact with its environment, opening wide ranges of new phenomena to be studied. Perhaps the simplest and most natural form of sensor is the obstacle detecting type. Such sensors can be implemented in a number of ways -- through photoelectric, mechanical, or electrical switches for example. We have, thus far, experimented with several versions of mechanical touch sensors.

First work with such sensors might consist of moving the turtle across a crowded room. This is distinguished from the more complicated maze traversal programs which follow naturally through a series of relatively simple situations given to the student. Another possibility for rather simple introductory programs is area-covering search techniques. These topics have complex and mathematically interesting ramifications.

A more difficult and rewarding area of study is associated with topological situations involving recognition. For example, we can start with the proglem of recognizing a simple polygonal shape using the turtle as probe. A very difficult problem -- recognizing connectivity of given configurations soon follows. Here we are well within the realm of finite topology problems encountered in artificial intelligence. Thus, use of sensors, spans an enormous range of teaching possibilities.

5. METHODOLOGICAL DEVELOPMENTS

Even when students are nominally working on the same problems their methods, programs, and resolutions of difficulties, i.e., their problem-solving and programming interactions, are very different and highly individual. When several students are working concurrently and independently during a LOGO laboratory session, an instructor cannot adequately monitor and follow all their work during that session: too much is happening and a great deal of concentration is required to penetrate particular sources of difficulties and to suggest new directions of work with even a single student. To make easier the reading, editing, and analysis of student work, we developed facilities for recording the students' interactions with LOGO as they are produced. In the next section we describe the generation of such "dribble files", illustrate their use in analyzing student work, and discuss some important extensions of this facility.

Section 5.2 is a comprehensive engineering description of the LOGO-controlled robot turtle.

5.1 Dribble Files

The use of programming languages to individualize instruction creates new problems for the teacher, both in monitoring the students' work and in helping to debug it. In this connection, we wrote a set of programs for creating dribble files as a by-product of student work. We store the student's type-ins but not the associated computer responses. (These can be regenerated subsequently.) We have produced dribble files of student work in the introductory mathematics course given at the University of Massachusetts. In about eight weeks the nine students in the course generated the equivalent of about 1500 printed pages of dribble file information. (This would be approximately doubled with the inclusion of the associated responses.)

In the next pages, we shall discuss the content and the uses of such dribble file information. The following example shows the listing of a fragment of a dribble file made from the work of one of the students in this course, RC. The file is identified on the top line: RC.DRB;2, along with the date and time it was generated. We have prefixed the lines with reference numbers Ø through 17. Each line starts off with the time stamp. Thus in line Ø the number Ø:ØØ:1Ø means Ø hours ØØ minutes and 1Ø seconds of time required to complete this line.

The material concerns the development of a procedure for drawing triangles. Lines Ø and 1 direct the definitions of the procedures NUM and TRIANGLE to be listed. The resulting printouts are shown on the right. The TRIANGLE procedure is edited several times during this session: in lines 2, 3, and 4; later in lines 6, 7, 8, and 9; and once again in lines 12, 13, and 14. In between these successive editing modifications, the effect of the changes made in TRIANGLE is tested by executing the procedure NUM

with the input 8. This is done in line 5, then in line 11, and finally in line 15. The computer printouts from the executions - the various drawings - are shown on the right. At the end of this exchange (line 16), the final version of TRIANGLE (which still has a "bug" in its stopping rule) is listed. In line 17 NUM and TRIANGLE are stored in their current forms in a file labeled "JOHN CAD".

```
RC.DRB;2   THU 29-APR-71 12;4ØPM

Ø Ø:ØØ:1Ø LIST NUM
                                    TO NUM /N/
                                    1Ø MAKE "NU" 1
                                    2Ø TRIANGLE /N/
                                    END

1 Ø:ØØ:11 LIST TRIANGLE
                                    TO TRIANGLE /N/
                                    1Ø MARK "X" /NU/
                                    2Ø PRINT /EMPTY/
                                    3Ø TEST IS /NU/ /N/
                                    4Ø IF TRUE STOP
                                    5Ø MAKE "NU" SUM OF /NU/ AND 1
                                    6Ø TRIANGLE /N/
                                    END

2 Ø:ØØ:1Ø EDIT TRIANGLE
3 Ø:ØØ:43 1Ø MIDDLE 5Ø "X" /NU/
4 Ø:ØØ:Ø5 END
5 Ø:ØØ:Ø9 NUM 8
                                       X
                                       XX
                                      XXX
                                      XXXX
                                     XXXXX
                                     XXXXXX
                                    XXXXXXX
                                    XXXXXXXX

6 Ø:Ø1:Ø5 EDIT TRIANGLE
7 Ø:ØØ:26 5Ø MAKE "NU" SUM /NU/ 2
8 Ø:ØØ:25 3Ø TEST IS SUM /NU/ /N/
9 Ø:ØØ:Ø3 END
1Ø Ø:ØØ:Ø9 NUM8\8
                                    NUM8 NEEDS A MEANING
11 Ø:ØØ:Ø7 NUM 8
                                    X

                                    SOMETHING MISSING. IS NEEDS
                                    ANOTHER INPUT.  I WAS AT LINE 3Ø
                                    IN TRIANGLE
```

```
12 Ø:Ø2:49 EDIT TRIANGLE
13 Ø:ØØ:17 3Ø TEST IS SUM /NU/ 2 /N/
14 Ø:ØØ:Ø2 END
15 Ø:ØØ:Ø6 NUM 8
                                        X
                                       XXX
                                      XXXXX
                                     XXXXXXX
                                    XXXXXXXXX
                                   XXXXXXXXXXX

                                   WHEN YOU STOPPED ME I WAS AT
                                   LINE 5Ø IN TRIANGLE

16 Ø:ØØ:38 LIST TRIANGLE           TO TRIANGLE /N/
                                   1Ø MIDDLE 5Ø "X" /NU/
                                   2Ø PRINT /EMPTY/
                                   3Ø TEST IS SUM /NU/ 2 /N/
                                   4Ø IF TRUE STOP
                                   5Ø MAKE "NU" SUM /NU/ 2
                                   6Ø TRIANGLE /N/
                                   END

17 Ø:Ø1:ØØ SAVE JOHN CAD
```

This material illustrates dribble files for a single session. To investigate the acquisition of problem-solving skills, it is useful to consider a student's work from a more global point of view. To see the kind of analysis possible, we consider the work of a single student, RC, on the teletype geometry sequence. RC, early in the term, was confronted by the need for a procedure to find the integral half of a number. Her algorithm consisted of successively adding 1 to a trial "half" and testing to see whether its double was within 1 of the original number. Having, after considerable effort, written the recursive procedure FIND to do this, she then saw the need for another program to do the initialization and wrote HALF. Annotated listing of both programs are given following. They fall neatly into distinct parts as labeled. The algorithm itself is, perhaps, not one that a more sophisticated programmer would use. Also, in many places RC is more obscure than is necessary. Real student-written programs are like this, however.

```
                TO HALF /N/                          (/N/ is the number
                                                      to be halved)
Initialize      1Ø MAKE "TRIAL" Ø                    (Set the "trial"
                                                      value of half to Ø)
Call Simply-    2Ø OUTPUT FIND OF /N/                (Output the result
Recursive                                             of FIND as the
Procedure                                             answer)
                END

                TO FIND /N/

End-Test        1Ø TEST GREATERP OF 2                (Is 2x/TRIAL/
                   AND DIFFERENCE OF                  within 1 of /N/)
                   (/N/) (PRODUCT 2 /TRIAL/)
                2Ø IF TRUE OUTPUT /TRIAL/            (If so, /TRIAL/ is
                                                      the answer)
Increment       3Ø MAKE "TRIAL"                      (Otherwise, add 1
                   (SUM OF /TRIAL/ AND 1)            to /TRIAL/)
Recursion       4Ø OUTPUT FIND OF /N/                (and repeat FIND)
                END
```

About a week later the same student wrote a pair of programs to automatically draw triangles. We showed the dribble file for the last part of this development just above. The form of these programs is nearly identical to the ones for halving. The only change is that each step of the recursive procedure TRIANGLE results in an action and this was not true for FIND. This task, however, only took about half the time required for the earlier one. Along with this, the problem was approached much more directly, as is evident from looking at the dribble file. Clearly, this program form was being internalized.

```
                TO NUM /N/                           (/N/ is the number
                                                      of X's in the bottom
                                                      row of the triangle)
Initialize      1Ø MAKE "NU" 1                       (Make the number of
                                                      X's in the current
                                                      row 1)
Call Simply-    2Ø TRIANGLE /N/                      (Draw the triangle)
Recursive
Procedure
                END
```

```
                TO TRIANGLE /N/
Action          1Ø MIDDLE 5Ø "X" /NU/        (Mark the current
                                              row)
                2Ø PRINT /EMPTY/              (Start the next row)
End-Test        3Ø TEST IS (SUM OF            (Is this the last
                   /NU/ 2) (/N/)              row)
                4Ø IF TRUE STOP               (If so, done)
Increment       5Ø MAKE "NU"                  (Otherwise, get
                   (SUM OF /NU/ 2)            number of marks in
                                              current row)
Recursion       6Ø TRIANGLE /N/               (and repeat
                                              TRIANGLE)
                END
```

This example forms a small part of RC's work on the geometric figure drawing sequence. In all, she used three different program forms: the one which we have just discussed which we will call form II; simple recursion which we label form I; and form Ø which is a linear sequence of steps. The diagram given as Figure 1 shows all the connections between the various parts of the drawing sequence. The program forms are indicated in parentheses after each procedure name. A more complete and useful "flow chart" would give the conditions for recursion and termination of each procedure of form I or II. This information has been omitted, however, for the sake of clarity and conciseness.

Another student was working on programs for drawing geometric figures during the same period. The diagram associated with the work of this student, AF, is shown in Figure 2. These students spent about three weeks near the beginning of the term writing these programs. Thus it is apparent that complex structures can be generated quickly, even by "beginners". These examples show some of the issues involved in analyzing complex student interactions. Great differences in program organization in the two cases are apparent, even though the set of programs have the same final effect.

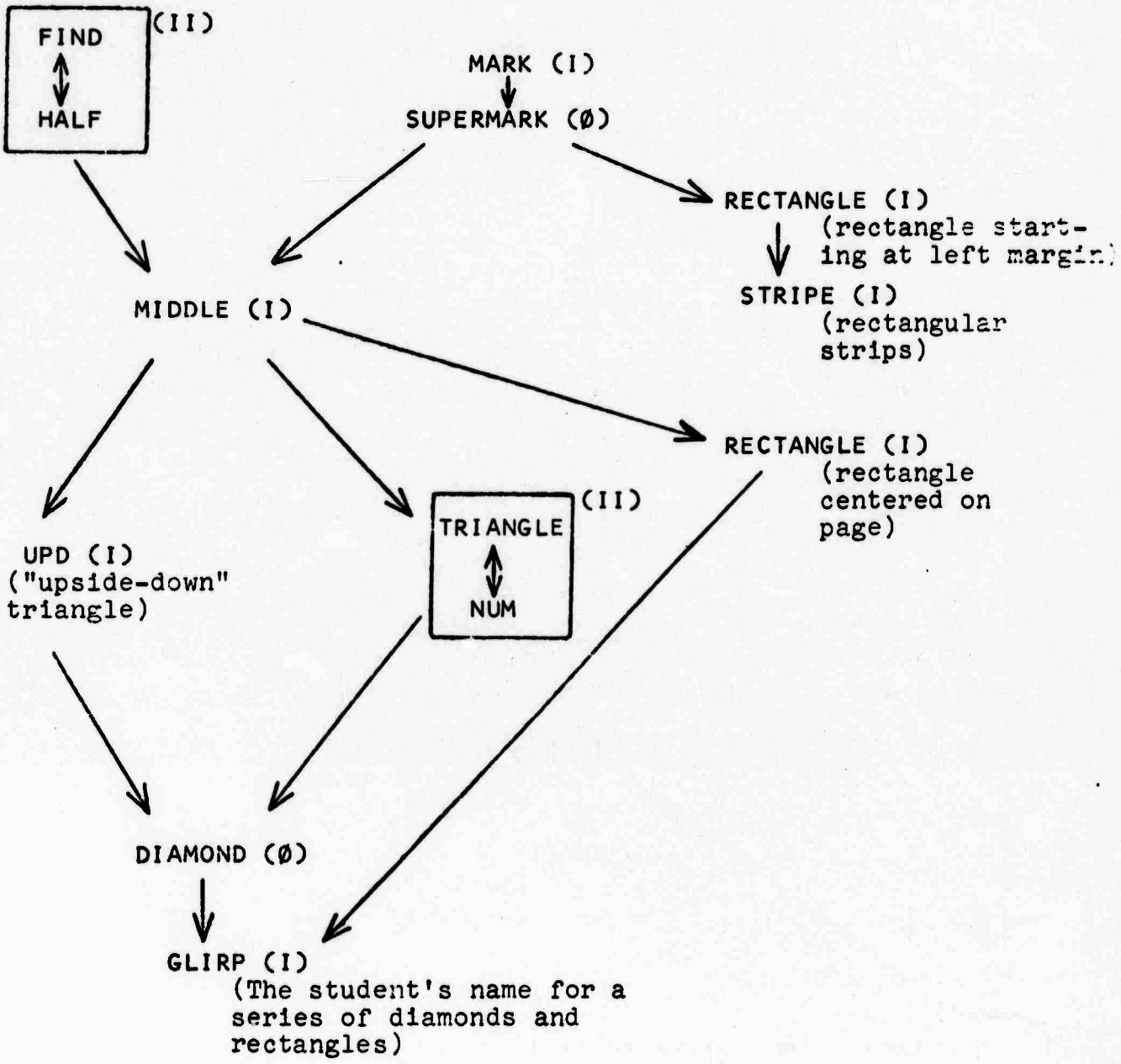


Figure 1. Diagram of RC's Drawing Program

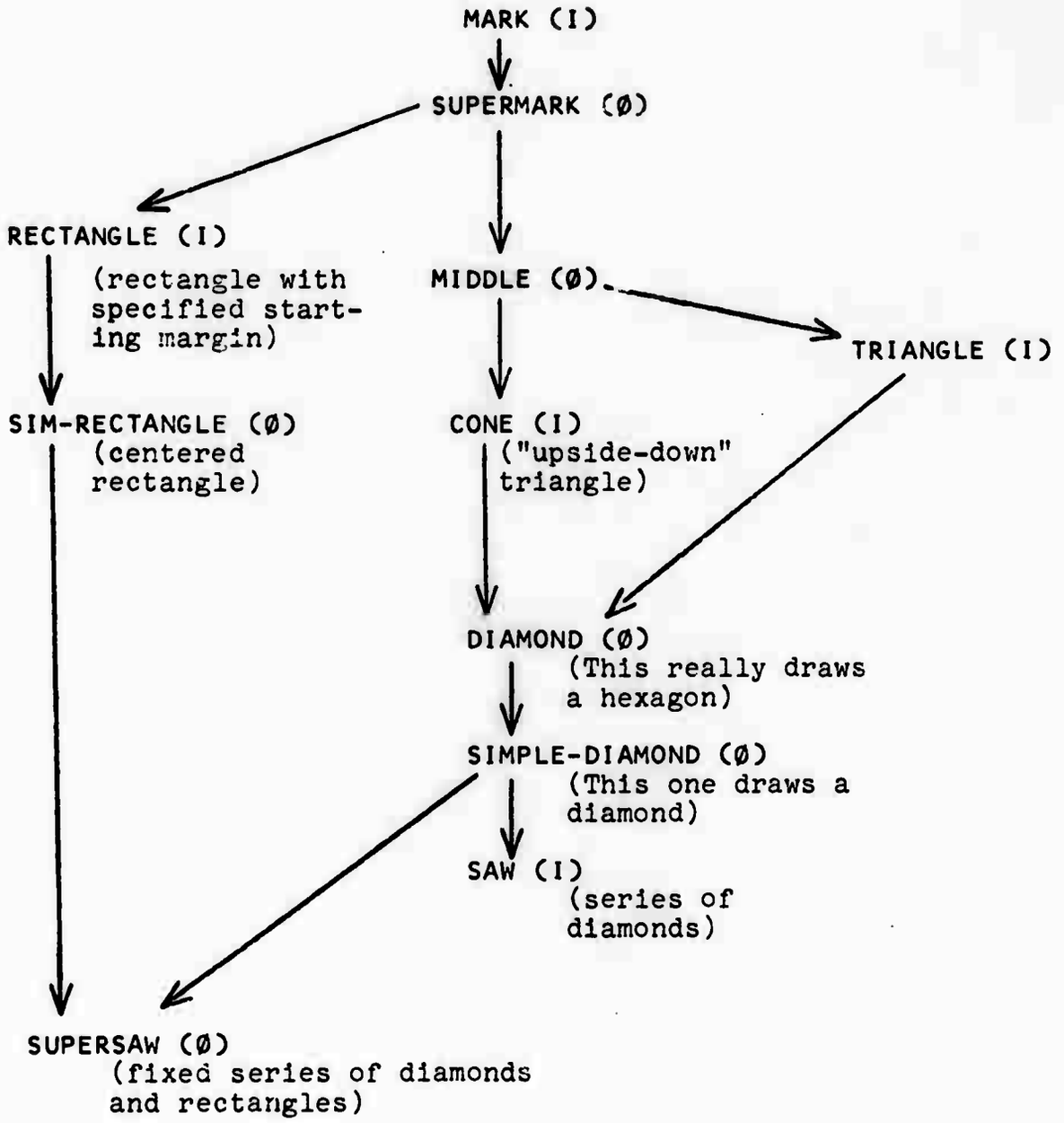


Figure 2. Diagram of AF's Drawing Programs

Extending the Utility of Dribble Files

The "dribble file" we have described contains all details of the student-computer interaction as it occurs at the teletypewriter. By *replaying* a dribble file, we can even get all the information at the systems level. Thus, the dribble file certainly contains all the raw data available for analysis. The very completeness and bulk of the information in the dribble file, however, discourage us from doing any searching and processing directly. We could have collected the data selectively to reduce the size of the file but preselection of the data to be preserved can turn out badly. Furthermore, any preselection rules can be applied to the dribble file itself which can then be saved as a backup. With this strategy, if it turns out in light of consequent results that a poor choice has been made, a new "preselection" can be done on the dribble file. This is our rationale for saving all the data.

Since it is inefficient to use dribble files directly, we must ask what aids exist or can be devised to make their use manageable. The most rudimentary such aid is a text-editing language, such as TECO, as implemented on the PDP-10 computer system. Direct character-by-character matching is made very easy by such a language. Thus, for example, one could delete all time marks in a given file or all directly executed input lines. One could also delete all lines followed by an error message, if one can specify the format of an error diagnostic statement. These actions are all the results of simple format matching. Also, it is easy for a user to insert comments into a dribble file using TECO. If, in addition, one can combine series of the basic searching, inserting, deleting, and pointer-moving commands, with

numeric and branching capabilities, there is the possibility of extremely sophisticated types of processing. In fact, the "Q-registers" that TECO provides for storing stacks make the language perfectly general and permit Turing-machine-like programs to be written for all computable functions. One could, for example, with some effort write a program using TECO to find and enumerate all simple recursive programs in a dribble file.

Unfortunately, in a practical sense this is about as far as one can go with TECO. First, one is writing programs in what is essentially machine-language, a rather tedious undertaking. Also, it is difficult to write programs that are easily extensible.

Thus, two requirements that a dribble file analysis language must satisfy are already apparent. The language itself must be natural in form to accommodate the unsophisticated user, and user-written procedures must be transparent to permit their use in further procedures. It is clear that such a language should appropriately incorporate the text-handling and editing features which are already in common use. By making the language self-extensible, so that sets of programs of any depth can easily be written, we satisfy the requirement of transparency. Also, it is much easier to write general programs in a self-extensible language. For example, instead of writing a TECO program which looks in the dribble file for an object of some given form, one can, with about the same effort, write a program in such a language, one of whose inputs specifies the form to be found. Also, a SNOBOL-like set of matching procedures could be written in the file analysis language itself instead of being given as part of the set of primitives.

Let us discuss the use of the analysis language next. Often a teacher wishes to classify the programs written by his student in a way he specifies. It would clearly be very

inefficient to have to perform this analysis on the same programs each time he looked at the dribble files. The standard result of an analysis of a dribble file should therefore be a new file containing the processed data, with tags joining it to the original file at points of correspondence. This means that the user can look through the processed file using his own set of descriptors and can go back to the raw data whenever necessary.

The idea of being able to operate upon the file at multiple levels of detail is of very general use. In analyzing the work of a student through his dribble files, there are several levels which may be of nearly simultaneous interest. For the top level, a good mode of presentation might be a flow chart, dynamically changing as the user scans the student's work, indicating all the programs in the student's workspace and showing the changing connections between them. At a lower level one might have a complete specification of all the student's programs at that moment in time. At the lowest level one would probably want a "cleaned up" version of the dribble file with (what the user considers) the obscuring features deleted. The analysis of the dribble file would begin at the top level. When programs of particular interest appeared, they might be listed or executed. Still further, the details of their creation and use by the student might be explored at the lowest level.

Thus, we anticipate the need for a set of programs enabling the user to switch back and forth between levels, zooming in when he needs more information, allowing him to vary his scanning rates, to go back and forth between current and previous material, to switch from scanning to execution mode, and so on. Let us consider next some kinds of information that will be of interest.

Apart from considerations specific to the content being studied, the user probably will be interested in general questions regarding the formal structure and organization of the student's work. Examples are: (1) What kinds of programs were used, i.e., what standard functions did the programs have (such as initialization, testing, and computation)? (2) What elementary program forms were used (loop-free sequence, iteration, simple recursion, etc.)? (3) What was the program organization, i.e., how were the various programs combined (program tree, substructure type, recursion diagram)? Thus, using this extended system, we can characterize the functional, formal, and organizational features of the work of particular students.

5.2 The "Turtle" - A LOGO-Controlled Vehicle

A teaching sequence centered on use of a LOGO-controlled remote vehicle, the "turtle", was described in Section 4.3. Here we discuss the design and construction of the turtle. Figure 3 is a block diagram showing the main components involved in turtle operation, and the links between them. A turtle command is initiated by the computer, which sends a signal to the teletype, which in turn activates a transmitter. This signal is picked up and decoded by a receiver built into the turtle. This decoded signal is fed into a control unit which applies a suitably polarized voltage to each of the motors turning the turtle's two wheels. The angle of rotation is precisely measured by a cam switch mounted on each wheel. Besides the four motion commands, the HORN command sounds the turtle's horn. The turtle's motors and electronics are powered by a nickel-cadmium battery pack.

The design criteria were that the turtle's motions should be accurately repeatable and that the material used in construction be readily available to facilitate replication. Each of the components of the turtle system, and the sequence of events involved in complete execution of a turtle command is described in detail below.

Computer CPU

LOGO executes a turtle command by sending a string of characters to the teletype (TTY). The first character of this string is a control character to specify the command, which the TTY decodes and sends to the transmitter unit (TU). This control character is followed by a number of waiting characters which are sent to ensure that succeeding turtle commands are not transmitted

before the completion of the current one. The character strings resulting from the five turtle commands are:

↑A#####	FRONT	(Go 1 unit forward)
↑K#####	BACK	(Go 1 unit backward)
↑H#####	RIGHT	(Rotate 15° clockwise)
↑V#####	LEFT	(Rotate 15° counterclockwise)
↑]##	HORN	(Sound horn)

(↑ before a character indicates control character.
# indicates wait character.)

All of the characters in these strings are non-printing. The number of #'s in the strings varies because the times necessary for completion of the actions vary.

Teletype (TTY)

The teletype is a KSR 33 teletype to which six function levers and function switches have been added. The turtle command characters are six reserved, non-printing control codes. Receipt of one of these characters causes the corresponding function lever to activate a function switch which transfers a signal down one of six control wires to the transmitter unit.

Transmitter Unit (TU)

The transmitter unit contains two sections-the transmitter (XMTR)* and the timing circuits. The XMTR operates on the 27 MHz Citizens Band, modulating the RF carrier with one of six discrete audio

*The transmitter and receiver are standard Citizens Band units, purchased complete. The modifications made are described below.

frequencies. Computer execution of a turtle command results in a signal on one of the six lines from the TTY. This pulse is smoothed and lengthened by the timing circuitry and is used to activate a relay which switches an appropriate capacitor into the audio oscillator circuitry of the XMTR and starts the oscillator. The RF control signals are then transmitted for a time period sufficient for the receiver unit (RU) to receive them accurately.

Receiver Unit (RU)

The receiver unit (RU) used in the turtle control link is similar to those commonly used in transistor portable radios, with two major differences. The first is that RU is crystal-controlled to accept signals only on the control transmitter frequency. The second difference is that the RU has no loudspeaker, instead it has a resonant reed relay with six tuned reeds. When a signal is applied, that reed which is tuned to the frequency of the excitation signal, vibrates back and forth making an electrical connection at one extreme of its path. The six output wires of the RU are taken from the contacts of these reeds. Thus, the RU acts not only as a receiver, but also as a discriminator which determines which function was selected at the transmitter. The signal appearing on one of the six output wires of the RU is smoothed to a 12-volt level and sent to the control logic.

Control Unit (CU)

Two of the six outputs of the receiver unit are designated as accessory channels. Currently one of these controls the bell* and the other is a spare.

*Actually a Mallory SONALERT® with a 20μf capacitor in parallel.

The other four channels are used for motor controlling -- one channel for each of the four motion commands. The control unit consists of two parts, the motor control logic (MCL) and accessory controls. The MCL has separate control sections for each of the two wheels. These wheels have cam switch operators on their shafts. The MCL determines when the specified rotation has been completed by each wheel by counting the number of operations of the cam switches. The use of cam switches makes it possible to maintain high precision in the motion of the turtle and obviate concern for all but gross differences in the speeds of the two motors and in battery life.

Functioning of the MCL

The sequence of events is depicted by the flow chart in Fig. 4. When a motion control command is received from the RU, a 12-volt signal is diode gated to actuate motor relays which start the motors in appropriate directions. (See Table 1.)

Any motion command also generates an ENB (enable) signal. This 12-volt signal is converted to a Ø or 5 volt TTL logic signal (for the integrated circuits) which is used to set the holding flip-flops FF_1 and FF_2 via the SET_1, SET_2 signals, respectively. (See Fig. 6a.) When these flip-flops are set, they energize relays G_1 and G_2 which, in turn, generate signals $GSIG_1$ and $GSIG_2$. The GSIG signals are fed back to the coils of the motor relays to keep them actuated after the control command is removed. These relays remain "on", keeping the motors rotating until motion is completed and FF_1 and FF_2 are cleared. Each wheel has its own complement of counting and reset logic and operates independently.

A typical sequence of events for the right wheel logic starts with a "FRONT" signal from the RU. A flow chart of the sequence appears in Fig. 4. The actions of the wheel logic in response to this command can be broken down into a sequence of distinct steps: All lettered actions under any one step number occur nearly simultaneously.

1. A. The signal comes from the RU into the diode gating network. (Left side of Fig. 5a.)

   B. This signal goes through the coils of FR_1 and FR_2 energizing them. (Fig. 5a.) From this point on, the logic associated with motor 1 and with motor 2 function identically so we will refer only to the former.

   C. It also appears at the ENB (enable) point as the ENB signal.

2. A. Two sets of normally open contacts of the FR_1 relay now close applying 6 volts to the motor M_1. (Fig. 5d.)

   B. The 12-volt ENB signal is converted to a logic signal SET_1 for the integrated circuits and is used to set FF_1. (Fig. 6a.)

   C. The output of FF_1, signal G_1 goes through a relay driver (Fig. 6b) and energizes relay G_1. (Right side of Fig. 6a.)

   D. There are two sets of contacts in the G_1 relay: one set open, removing the brakes from the motor (Fig. 5d); the other set of contacts close, generating $GSIG_1$ (Fig. 5c).

3. $GSIG_1$ is fed back to the coil of FR_1 through a closed set of contacts of FR_1 (Fig. 5a) to hold this relay actuated until the motion is completed.

(At this point, the motor and wheel are rotating forwards, FF_1 is set, the brakes are off, and the motor control relay FR_1 is being held on by $GSIG_1$. This state will be maintained until the counting logic has counted the correct number of increments of wheel rotation (six for an F command).)

4. A. The cam switch senses (in about 1/2 second) that the wheel has rotated one detent (15°) (Fig. 6a) and generates signal $CLOCK_1$.

   B. $CLOCK_1$ triggers the counting 8-bit shift register SR_1 (which has been previously cleared to all Ø's) which shifts a logic 1 into bit 1 (Fig. 6a).

5. Subsequent pulses of $CLOCK_1$ from the cam switch will shift logic 1's right until bit 6 switches from logic Ø to logic 1.

6. When the output of bit 6 of SR_1 goes to a logic 1, this output signal fires the ONE-SHOT generating a 100 ms. square pulse. This $CLEAR_1$ pulse is used to clear SR_1 and to reset FF_1 (Fig. 6a).

7. Resetting FF_1 removes signal G_1 and deactivates relay G_1 (Fig. 6a).

8. A. With G_1 deactivated, $GSIG_1$ goes to zero (Fig. 5a) removing the holding voltage on FR_1 (Fig. 5a).

   B. The motor voltage is removed (Fig. 5d).

   C. The brakes (Fig. 5d) stop the motor.

If a rotation (R or L) command is initiated, the system resets after both wheels have gone only one detent, (15°). This is accomplished through the use of the H relay.

When an R or L rotation is requested, along with starting the motors in the correct directions, the H_{IN} signal is generated in the diode gating (Fig. 5a and Table 1). This signal energizes the H relay (Fig. 5b) which is held on by $GSIG_1$ and/or $GSIG_2$. The shift registers SR_1 and SR_2 both have the outputs of bit 1 connected through the now closed contacts of the H relay to the resetting ONE-SHOTs (Fig. 6a). When the H relay is energized, the systems reset on the first clock pulse. If one wheel completes before the other, the logic for that wheel will reset, but the GSIG of the moving wheel will hold the H relay energized until both wheels have completed their 15° rotation. At this point, the system is ready for another motion command.

Mechanical Design

The driving mechanism of the turtle consists of a pair of motors each driving one wheel, through a gear train. The angular velocity at the wheels is about 14 rpm. The axle for each wheel holds a driving wheel, a cam wheel, and a driving gear (Fig. 7). The cam switches are mounted on the base plate in such a way as to insure operation regardless of the direction of rotation of the wheels. The basic dimensions of the turtle are:

diameter of turtle	12"
diameter of wheels	3.75"
distance between wheels	7.5"

There are two ball coasters mounted on the bottom of the base plate for balance and there is a 12" diameter clear plastic hemisphere which mounts on the top above a 3" rim.

Sensors

Possibilities exist for several types of sensors on the turtle. Thus far the only ones we have incorporated are touch sensors. These are implemented as two long semi-rigid wire actuators attached to microswitches which are thrown when the turtle comes close to a solid object. The signal train for returning these data to LOGO is very similar to the command train sequence except that the radio link is on a different frequency to avoid interference. When the sensor signal gets to the teletype, a relay is actuated which closes the appropriate keyboard contacts and triggers the teletype to send the character selected to the computer.

Other possible sensors include a pair of photocells which could be used to seek a light or follow a printed line, an electromagnetic detector of metallic strips or filings, and an ultrasonic "ear" to allow sound seeking or to relay audible signals.

Table 1.

Command Received	Right Motor	Left Motor	HSig
F (FRONT)	Forwards	Forwards	Ø V
B (BACK)	Backwards	Backwards	Ø V
R (RIGHT)	Backwards	Forwards	+12 V
L (LEFT)	Forwards	Backwards	+12 V

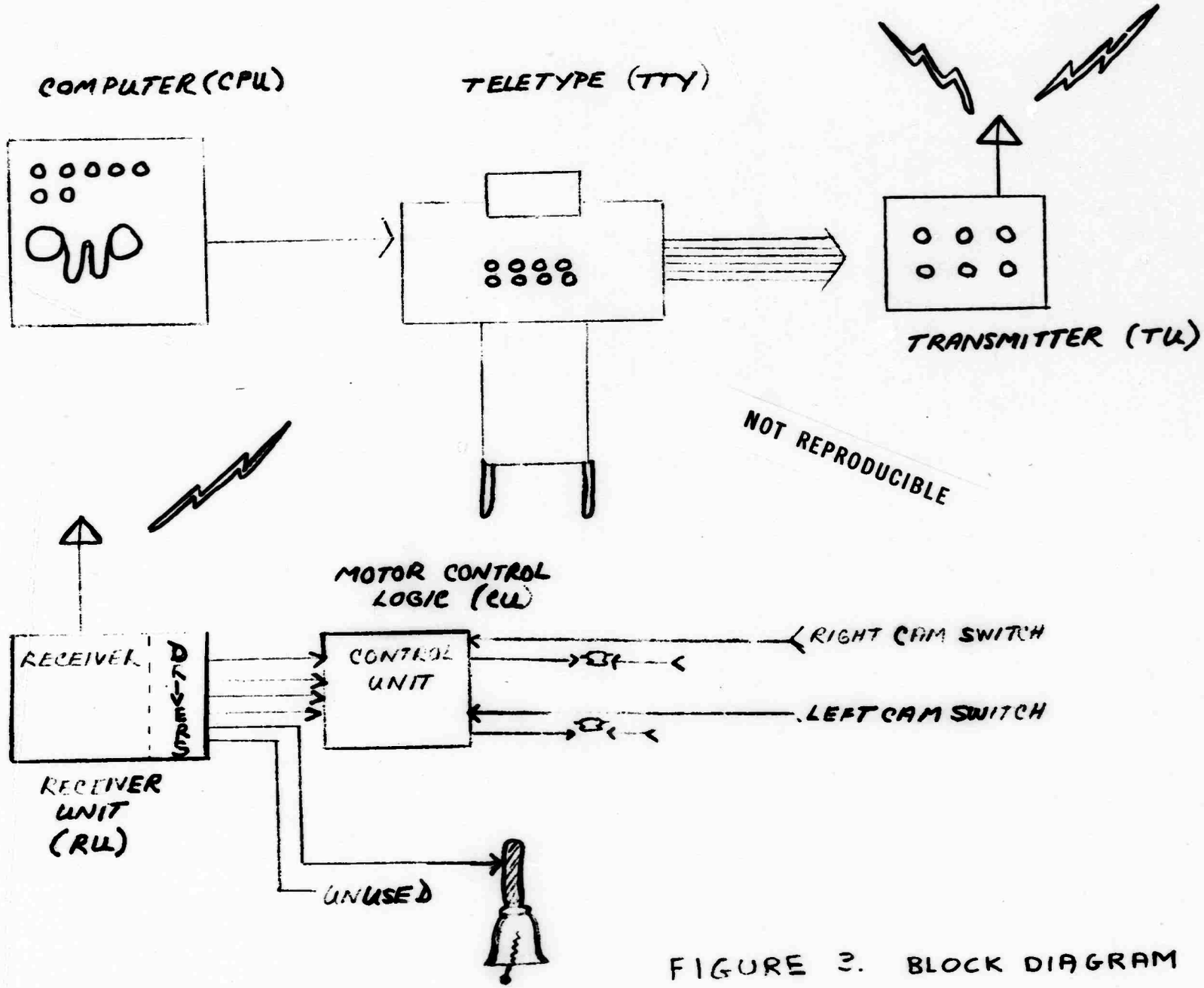


FIGURE 3. BLOCK DIAGRAM

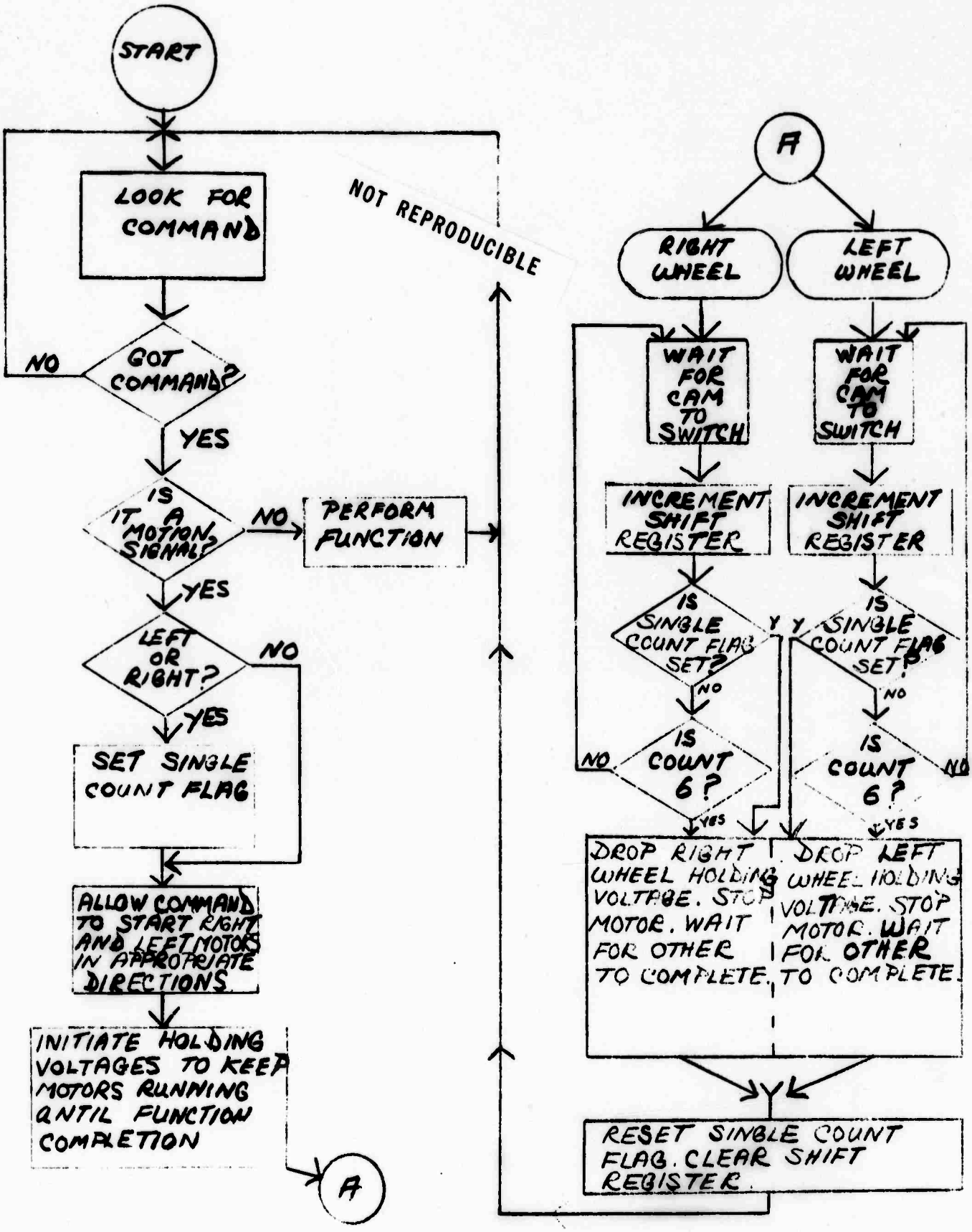
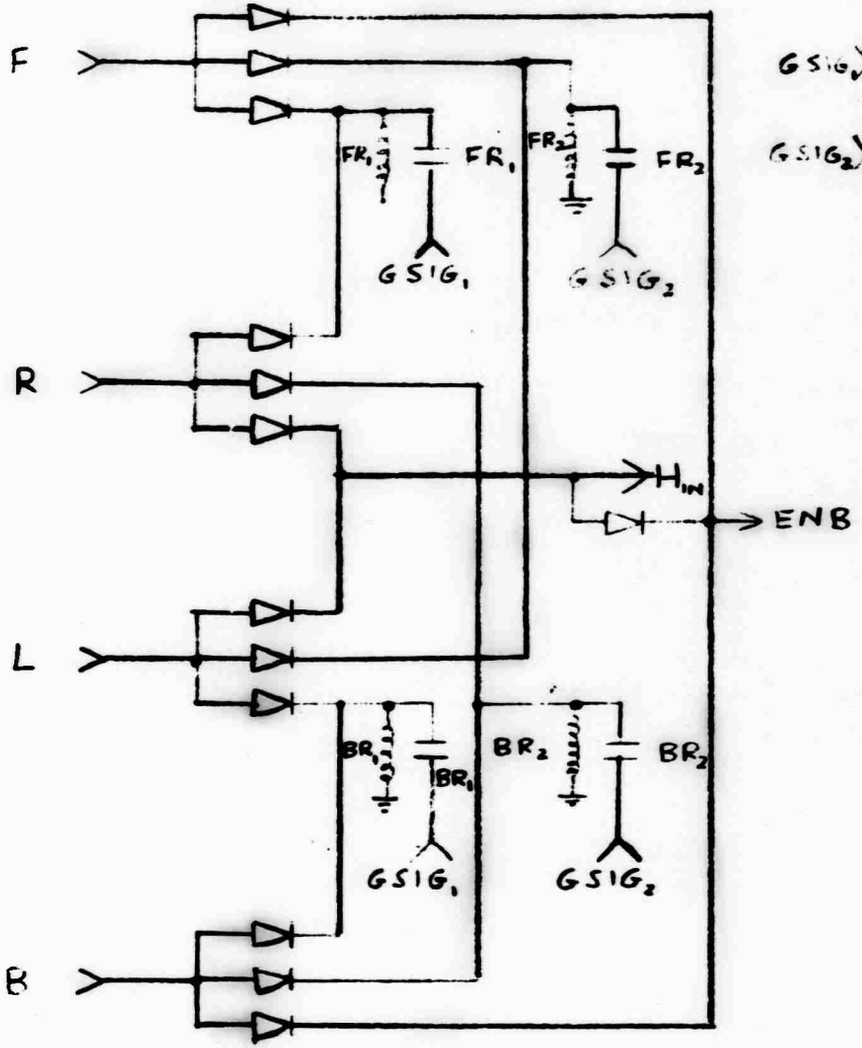
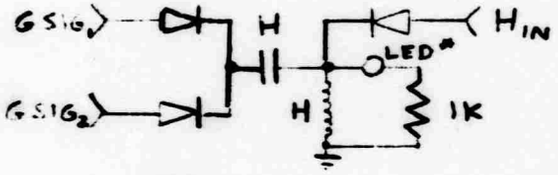


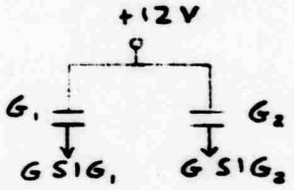
FIGURE 4. FLOWCHART FOR MCL



5a.



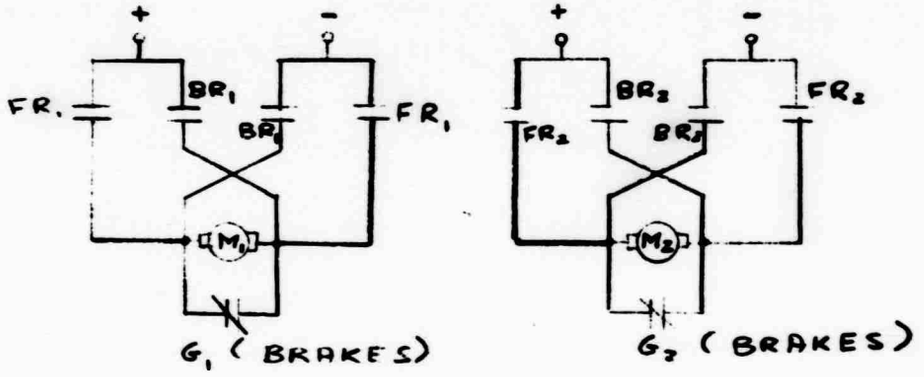
5b.



5c.

RELAY	NO	NC
FR1	3	ϕ
FR2	3	ϕ
BR1	3	ϕ
BR2	3	ϕ
G1	1	1
G2	1	1
H	3	ϕ

RELAY TABLE



5d.

* OPTIONAL

FIGURE 5. SCHEMATIC DIAGRAMS

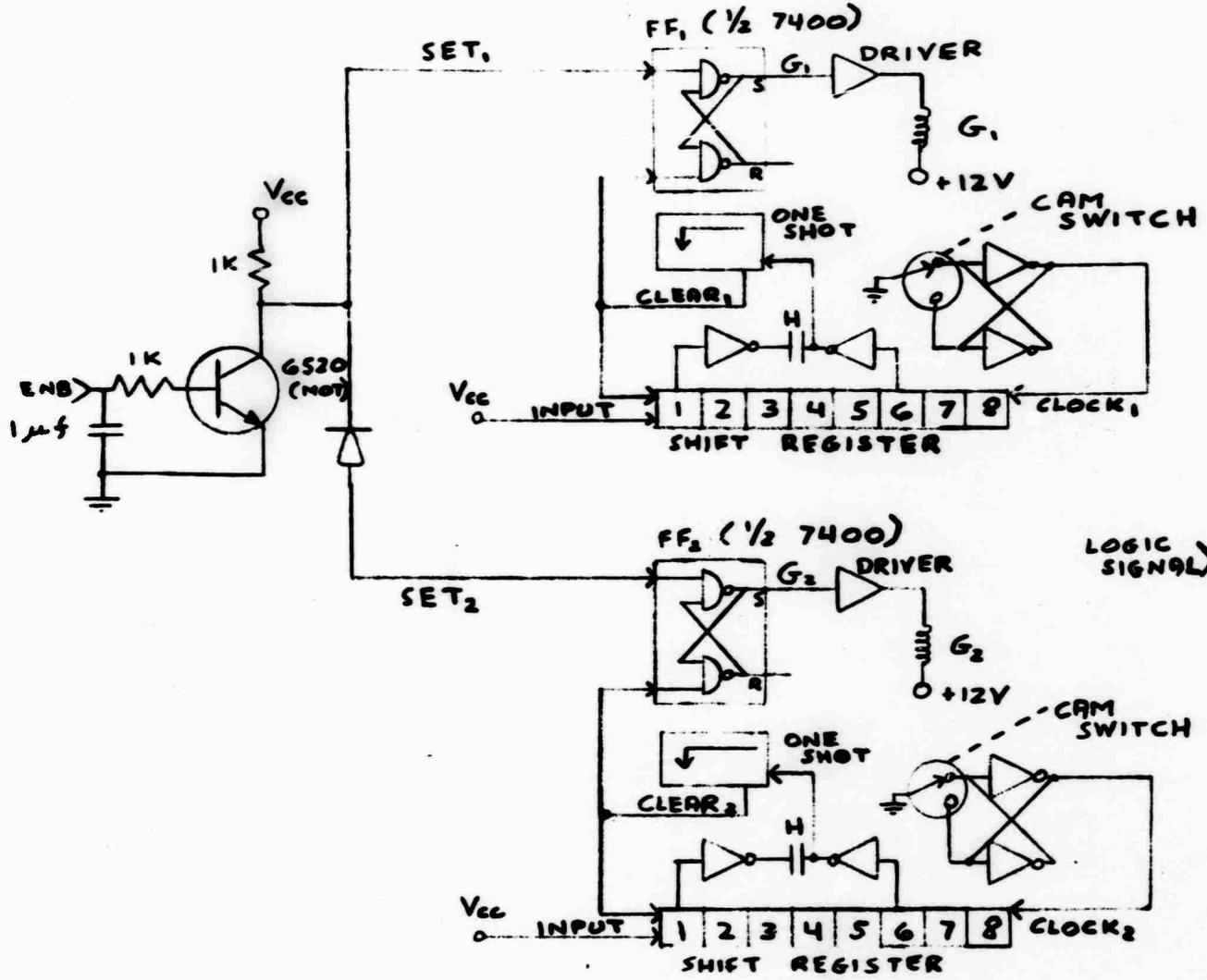


FIGURE 6a. COUNTING AND RESET LOGIC (V_{cc} = +5V)

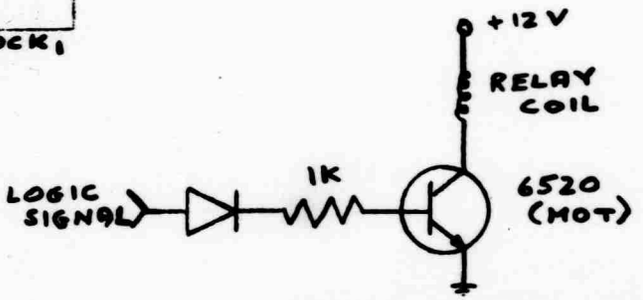


FIGURE 6b. SCHEMATIC OF A DRIVER

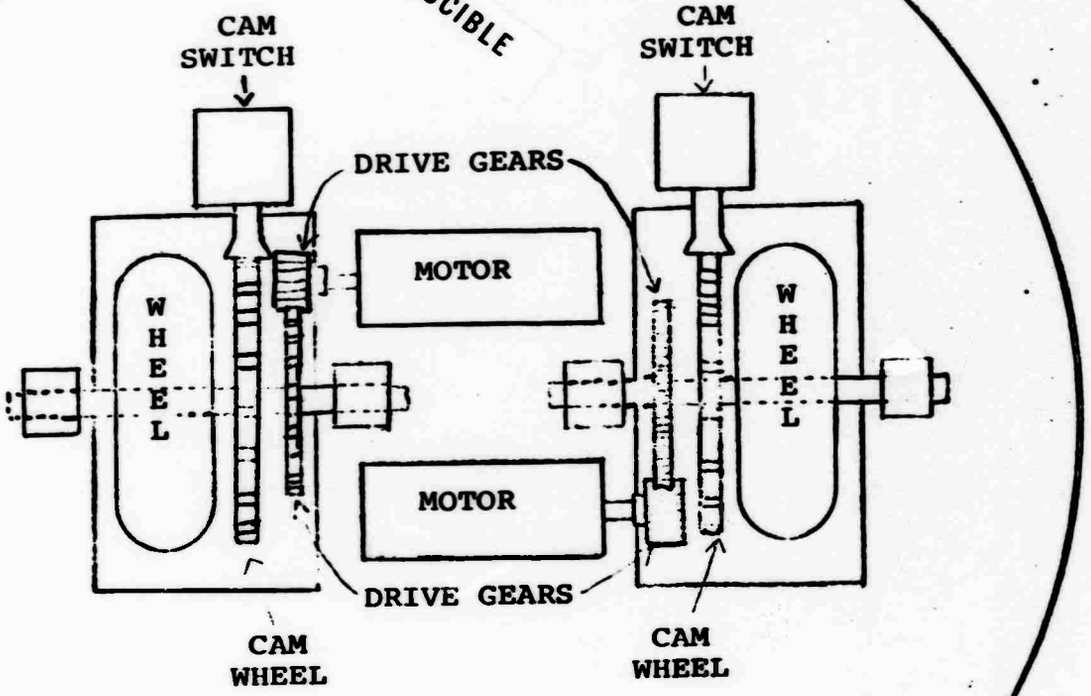


FIGURE 7 MECHANICAL DESIGN

6. REFERENCES

Feurzeig, W., Lukas, G., ..., BBN Report 2165, "Programming Languages as a Conceptual Framework for Teaching Mathematics," Final Report on National Science Foundation NSF-C 615, June 1971.

Feurzeig, W. and Lukas, C., LOGO - A Programming Language for Teaching Mathematics, Educational Technology (in preparation) Nov. 1971.

Polya, G., *How to Solve It*, Princeton, 1945, Doubleday Anchor, 1957.

" *Mathematics and Plausible Reasoning*, Princeton, 1954.

" *Mathematical Discovery*, Vols. 1,2, Wiley 1962,1965.

APPENDIX

REEVALUATING LOW ACHIEVERS WITH COMPUTER-ADMINISTERED TESTS

This appendix contains a detailed report of the reading test study carried out as part of the teaching experiment described in Section 4.2. The material is being submitted for publication in the Journal of Educational Measurement.

Abstract

We have recently obtained experimental results which indicate that standard tests of reading comprehension significantly underassess the actual achievement levels of many low-scoring children. When the computer was used to administer a standard test, the scores of low-achievers increased dramatically. Comparable improvements did not occur with high-achievers -- in fact, their performance actually suffered somewhat on the computer-administered version of the test. These shifts in performance are not substantially changed when the effect of regression toward the mean is taken into account. Neither can they be explained as an artifact due to order of administration nor by Hawthorne or novelty effects. They probably are due to attention-engagement phenomena associated with control, interaction, pacing, and partitioning aspects of the computer presentation. If these results are generally true for large segments of the school population, they will bear importantly on testing theory and educational placement.

1. Introduction

This report describes an experiment comparing the results of administering computer and paper-and-pencil tests on reading achievement to a group of eighth grade pupils at the Muzzey Junior High School in Lexington, Massachusetts in June 1970. Other comparisons of the results of tests by paper-and-pencil and by computer administration have been made previously. In one study with a group of "lowest-achieving" elementary pupils, a considerable number did better when they took the California Reading Test on a computer than when they took an equivalent form with paper-and-pencil. (Serwer and Stolurow, 1970).

In another study, two matched samples of lower-division students attending a university summer session took the sentence understanding (Part I) and word meanings (Part III) sections of the Cooperative School and College Ability Test (SCAT); one group took it under standard test conditions, the other took it under

computer administration. The computer subjects scored significantly higher than the paper-and-pencil subjects; the variance in their scores was significantly smaller primarily because of the smaller proportion of low scores in the computer administration. In effect, for the computer group, there were fewer low scores, more middle-range scores, and about the same number of high scores. (Vinsonhaler, Molineux, and Rogers, 1965).

Our testing experiment was a by-product of a larger research investigation involving the teaching of an experimental computer-based course in programming and problem-solving to six eighth-grade pupils. The pupils chosen were six boys with the greatest measured deficiencies in reading comprehension level in the school. Their comprehension scores ranged from three to five years below their current (eighth) grade level. They did not have observed physical or perceptual handicaps of any kind. With one exception, they had little interest in intellectual activity (and sometimes showed enormous resistance to it).

The teaching project had two objectives. One was to explore the use of computers and programming as a means of "turning on" these intellectually-alienated pupils and involving them in constructive thinking and problem-solving work. The other objective was to investigate our conjecture that the pupils' work with programming might significantly help their reading.

We were sure that all the pupils would enjoy working at the computer, and this was indeed confirmed. We designed the course on this premise: if learning to use the computer required pupils to do a great deal of reading of computer-administered lessons, they would be very willing to do so. We thought that this involvement might open up ways to improve their reading comprehension skills.

We did not expect that the pupils would show significant gains in reading over the relatively short time span of the course (about three months). Our interest was limited to making a judgment about the feasibility of such an indirect approach to reading instruction through the teaching of a programming language.

Nevertheless, our plan included administering standard pretests and posttests of reading comprehension to our six pupils and to a comparable control group. As the time for administering the posttests approached, we surmised that, in the same sense that our pupils did not give serious attention to official school work in the classroom, they probably *did not take tests*. We conjectured that their reading test scores did not adequately reflect their

actual achievement levels (or, at least, the levels at which they were actually capable of performing). Moreover, we hypothesized that if they took equivalent forms of the posttest, one in conventional paper-and-pencil fashion and one administered on the computer, they would do a great deal better on the latter.

We thought this hypothesis might hold generally for a larger group of reading "underachievers." We therefore extended our posttest design with the purpose of performing a new experiment focused on testing rather than instruction. We selected an expanded sample of 61 eighth-grade pupils in the school (including four of our six computer class pupils). The sample included about twenty pupils with a history of low reading achievement scores, about twenty pupils reading at levels that seemed low relative to their intelligence quotients, and about twenty performing at moderate to high levels.

All students took two forms of the (Triggs) Diagnostic Reading Test, upper level, survey section, comprehension scale designed for use from seventh through twelfth grades. Form B was administered in the standard fashion; form D was administered on the computer. Approximately half of the students (25 of the 61), randomly chosen, took the computer test before the paper-and-pencil test.

This achievement test presents a fairly extensive reading passage followed by twenty four-part multiple-choice questions. In the computer administration, the reading material was fragmented and presented sequentially on a teletype in segments averaging about three lines each. The pupil called for each segment when he was ready by striking a teletype key. In the same fashion, he called for and responded to each question in sequence whenever he wished. He could skip questions and subsequently go back and change answers.

No time limits were imposed in either administration (though we clocked the computer test times every three minutes). Pupils required from thirty to forty-five minutes to take the tests. The computer version generally took five to ten minutes longer than the paper-and-pencil version. Virtually all pupils reported that they enjoyed taking the computer version, even those who scored relatively low on it.

2. Analysis and Findings

The basic data from which our findings are derived are shown in Table 1. Pupils have been given identification numbers according to the rank of their I.Q.'s; I.Q. scores are from the Stanford-Binet tests that most of the pupils took when they were in the second grade. Junior high pupils take Survey E (seventh through ninth grades), Form 1M of the Gates-MacGinitie Reading Test in April each year. Table 1 lists their scores on the comprehension scale of the tests they took in the seventh and eighth grades; these scores are recorded in terms of grade-level equivalents. The comprehension scale of the survey section of the upper-level Diagnostic Reading ("Triggs") Test was used as the experimental measure; Form B was administered as a paper-and-pencil test and Form D as a computer-administered test in June of the pupils' eighth grade year. Table 1 records the number of correct answers to the 20 questions asked.

[Tables 1, 2 and 3 about here]

The distribution of computer scores over the corresponding paper-and-pencil scores is shown in Table 2. For purposes of analysis, the range of the 21 possible scores in the Diagnostic Reading Test (0-20) have been divided into septiles of three potential scores each. The data from Table 2 are analyzed in terms of these septiles in Table 3. In the latter table, the gross difference in means has been adjusted on the last line by subtracting the effect of the difference in the grand means; the last line of Table 3 represents a regression-toward-the-mean effect.

[Tables 4 and 5 about here]

Table 4 shows the distribution of computer scores across the paper-and-pencil scores. Table 5 presents this distribution in terms of septiles of the range of scores.

We shall analyze these scores in two general ways: one by looking at the relations between the paired scores, the other by looking at the total distribution of scores.

2.1 Analysis of Paired Scores

The most striking finding in the data is the great superiority of computer administered over paper-and-pencil administered test scores for those who scored low on paper-and-pencil tests; this is shown in Table 5. One way to analyze this and other effects

is within the framework of the regression-toward-the-mean phenomenon. We expect to find this phenomenon whenever two sets of similarly derived scores are compared. In Table 3, there is a tendency for paper-and-pencil scores to be closer to the mean on the average than the corresponding computer scores; similarly, in Table 5, there is a tendency for the computer scores to be closer to the mean than the corresponding pencil-and-paper scores. As mentioned above, however, where paper-and-pencil scores are low, the regression-toward-the-mean effect is an extreme one.

In order to improve our understanding of this effect, we suggest three models to describe a regression toward the mean. If two sets of scores are perfectly related, that is, if one score predicts another score perfectly, there is no such regression; the difference in the means of the two sets of scores will predict the relation between any two pairs. If, on the other hand, two such sets are randomly related, the regression toward the mean, theoretically at least, will be complete; even if one knows one of a pair of scores, the best predictor of its counterpart is the mean of its set. In practice, we normally anticipate neither of these extreme conditions, but a condition somewhere between them. We do not expect two sets of scores on similar tests to be perfectly correlated because of the differential reactions of individual subjects; neither do we expect random relationships since the same individuals are taking tests measuring similar attributes. What we predict is simply that the Y scores will be closer to the mean than the X scores in any given stratum.

[Figure 1 about here]

In Figure 1, we have tried to model these three conditions. X-scores are shown on the abscissa and Y-X differences on the ordinate. For the case where scores are perfectly correlated and differences are constant, our model would consist of a horizontal line set at the zero level. For the case where scores are randomly related, we show a diagonal broken line from corner to corner with a slope of b =-1.00; at any point along this line Y-X (the ordinate distance) equals the X-score minus the mean of the Y scores; as drawn the line assumes the X and Y means to be equal.

These regression lines are theoretically derived. We must derive the in-between case empirically.

We have done this by using the scores of the comprehension scale on the Gates-MacGinitie reading test pupil scores for 1969 and

1970 as shown in Table 1. The scores are comparable to those on the Diagnostic Reading Test in the sense that they are for the same pupils and measure the same ability. Since we are not interested in a substantive comparison of the two sets of scores, we pooled the regression of the eighth grade (1970) on the seventh grade (1969) scores and that of the seventh grade (1969) scores on those for the eighth grade (1970). This pooling doubles the number of pairs available to us, gives us two sets of scores—X and Y—both of whose means and distributions are identical, and gives us a symmetrical distribution of differences.

These data for the Gates-MacGinitie scores have been analyzed in the same fashion as the Diagnostic Reading Test scores. First, the range of the Gates-MacGinitie scores was made approximately comparable to the range of the Diagnostic Reading Test scores. This range was then divided into septiles and the regression toward the mean was calculated. The results are shown in Table 6.

The range of Gates-MacGinitie scores is shown across the top of Figure 1. The ordinate for the difference (regression toward the mean) in Gates-MacGinitie scores, similarly proportioned, is scaled on the right-hand side of the graph. The X_j and $Y_j - X_j$ values from Table 6 are plotted on Figure 1 and joined by a light, solid line. A regression curve fitted to these points is represented by a dashed line. This dashed line constitutes a model for the regression toward the mean of actual test scores where the forms of the test and the methods of administration were the same. Whereas for the line representing randomly paired scores, slope is expressed by $b = -1.00$ and in the perfectly coordinated pairs $b = 0.00$, for the line relating comparable tests, $b = -.44$. Since these tests were given a year apart, there may be more error variance, hence a steeper slope than would be true for tests taken close together. (The fit of the line to the data points is a good one: $r = -.98$.)

We now have three models with which to compare the regression of computer scores on paper-and-pencil scores in the Diagnostic Reading Test: random pairing, perfectly related pairs, and imperfect but related pairs of scores on the same test. The actual regression of computer on paper-and-pencil scores is shown by the heavy solid line. This line connects points that represent the values on the bottom line of Table 5; the scales for these points are along the bottom and left-hand margin of Fig. 1.

What seems to happen is that from the first to the fifth septile of the scoring range, the regression of the computer on pencil-and-paper scores follows the regression curve for random pairings. It is as though youngsters who scored in these ranges on the paper-and-pencil test took an entirely different test when they took the computer test. From the fifth to the seventh septile, it may be that the regression pattern is more like that of the two Gates-MacGinitie tests, although the data points are too few to establish a trend with any degree of certainty.

Another way of comparing the paired scores is by correlating them. The intercorrelations of all five sets of Table 1 scores are shown in Table 7. As one would anticipate from the regression-toward-the-mean analysis, the computed correlation between the computer and the paper-and-pencil Diagnostic Reading Test scores is .25 (accounting for $(.24)^2$ or six percent of the variance). The correlation between the two Gates-MacGinitie administrations—both paper-and-pencil—is much higher—.52. In fact, the correlations between the computer-administered Diagnostic Test and the three paper-and-pencil tests are comparable, ranging from .20 to .30. Similarly, the intercorrelations between the three paper-and-pencil tests are comparable—and higher—ranging from .50 to .57. (As measures of reliability and validity, they are disappointingly low.) One small piece of evidence that the computer administration may be no less valid than the paper-and-pencil administrations is that its correlation with the second-grade I.Q. scores—.37—is in the range of the correlations of the paper-and-pencil scores with I.Q.—.29 to .41. (Obviously, this does not establish the validity of the computer-administered test.) These data indicate a strong "methods variance" (Campbell and Fiske, 1959); the tests' modes of presentation may be as important in determining the correlations between them as are the reading comprehension abilities that they are designed to measure.

In short, this correlation analysis confirms the near-randomness of the relation between computer and pencil-and-paper scores on these reading tests; it further indicates that a test's method of presentation may have a substantial influence on the resulting scores.

2.2 Analysis of Score Distributions

From our analysis of paired computer and paper-and-pencil scores, we turn to a brief analysis of the distribution of these scores. Table 5 shows that the means of the two distributions are 11.0

for the paper-and-pencil scores and 12.1 for the computer score. The difference does not reach the .05 level of significance level if we apply a t-test for the difference of means of matched samples. (Hays, 1963, pp 333-335).

[Figure 2 about here]

The two distributions are shown graphically in Figure 2. Though the two means are not vastly different, the variabilities are; the range of the paper-and-pencil scores is from 2 to 18; its variance is 18.8; the range for the computer scores is from 6 to 17; its variance is 8.2. When these variances are compared by the F test, the probability that they are different by chance is less than .01. (Hays, 1963, 348-352). It is clear from Figure 2 that much of the larger variation in the paper-and-pencil scores comes from a greater number of low scores.

3. Discussion

In summary, a group of eighth graders took two forms of the comprehension scale of the Diagnostic Reading Test, one form by paper-and-pencil and one form by computer. Looking at the paired scores, a strong regression-toward-the-mean effect is apparent. Lower paper-and-pencil scores seem to be randomly paired with computer scores, while the more moderate regression that one usually encounters when tests are administered twice to the same group may be characteristic of high computer scores. The correlation between the two sets of scores is low—very low considering that we are dealing with two forms of the same test scale administered within days of one another. Correlations among three pencil-and-paper forms are higher. In the aggregate, mean scores for the two administrations differed by only a single scale score; but the range and variance of the computer scores were much smaller than for the paper-and-pencil scores; range and variance were smaller largely because there were fewer low scores on the computer than on the paper-and-pencil tests.

One effect that might influence the results is an order effect. Students might learn enough from the first test to make some difference in their scores on the second test. Table 8 compares first- and second-administration scores for extreme cases, that is the pairs where pupils had at least one score of five or less and those where they had one score of fifteen or more. Two pairs in which the differences were as large as twelve fall into both classes. The pairs are also divided into those where the paper-and-pencil form (B) was administered first and those where the computer form (D) was administered first. Numbers of cases (pupils) and mean differences between second scores are given for each category. Where pupils improve their performance from the first test to the second, the differences will be positive. In pairs having at least one score of five or less, there was an average gain in scores when Form B was administered first; but there was at least as large a loss when Form D was administered first. This pattern can be accounted for by the higher scores on the computer administration when paper-and-pencil scores were low. In pairs having one score of fifteen or more, there is an average loss from the first to the second administration no matter which form was taken first. This can be accounted for by the normal regression toward the mean between first and second administrations. No order effect is apparent.

Another possible explanation for the increase of computer over paper-and-pencil scores is that, for most of the students, computers are new and exciting; these would be motivating factors resembling the familiar "Hawthorne effect" (Roethlisberger and Dickson, 1939). If this effect is operating, the operation is confined to those who score low on the Diagnostic paper-and-pencil test, since, as we have seen in Table 5, the scores of those whose paper-and-pencil scores were in the upper three septiles of the range average lower on the computer-administered test.

While the numbers are small, we do have a group that should be reasonably free of the novelty effect. Earlier, we spoke of having exposed six boys who had reading difficulties to a computer language; these boys each had at least an hour a week at a teletype terminal over a period of ten weeks; we assume that this is long enough to have attenuated any novelty effect that the computer may have had for them. Four of these boys are in our sample.

Although the number is small, they have been matched with pupils who had the same scores on the paper-and-pencil administration of the Diagnostic Reading Test and whose eighth grade scores on the Gates-MacGinitie Reading Test were not over the 7.5 grade level equivalent. Table 9 shows the results. There are five "matching students," one each with Diagnostic paper-and-pencil scores of two, four, and eight and two with scores of five. The scores of the latter have been averaged and paired comparisons made. On the basis of these four pairings, there would appear to be some novelty effect. Average scores on the Gates-MacGinitie scale are almost identical. The experienced "novelty-proof" group gained 5.5 points on the computer-administered version, the "novelty susceptible" group gained 7.2 points. Certainly, the possibility of motivation from new experience cannot be thrown out; neither does it seem to explain all of the variance.

One way to summarize these results is as follows: pupils who had low scores on reading tests when they were administered by pencil-and-paper actually "took the test" when it was administered by computer. We must assume that the low paper-and-pencil scores that improved with the computer administration represented something less than the actual learning level of the low-scoring students and that the computer system used permitted a better measurement of those pupil's abilities.

A possible explanation is that the presentation pattern used with the computer system is more compatible with the test-taking skills of the students. The teletype emitted the material slowly—108 words per minute—and in chunks—two or three sentences of text (and, later, one question) at a time. The pupil was thus relieved of the task of analyzing the material or of setting a reading rate for himself. The comprehension task was not a big one, but a series of small ones. Not only may this partitioning have reduced the pupil's work in performing the task, but it may have provided for more sustained motivation by giving him a series of subgoals in addition to the goal of completing the whole task. This effect may apply both to the fragmenting of the reading text material and of the presentation of questions one at a time.

A second possibility is that the interactive aspects of the presentation were rewarding. Not only did the pupil get at least a minimal response by asking for the next unit to be presented, but he was able to control the situation by calling for the next unit when he was ready. What may be even more important is that there was a response to his control effort.

A third possibility is the novelty effect which may or may not be separable from the experimental or "Hawthorne" effect. We have already seen evidences for some novelty effect based on the performance of nine students. If there is a novelty effect, it points out the desirability of test conditions that motivate students to work at the testing task, but implies that the effects of the computer administration itself may be transient.

The hypotheses we have put forward to this point are advanced to explain the finding that pupils who did poorly on the paper-and-pencil test improved greatly when they took the computer-administered version. We need also to seek reasons for the reduced performance of students who did well on paper-and-pencil tests. Other hypotheses that would explain the finding that pupils who scored high on the paper-and-pencil test scored lower on the computer administered test tend to be the mirror image of those we have just advanced.

For students who did well on the paper-and-pencil comprehension test, the teletype emission rate or the fragmentation of the material may have made the task more difficult. Similarly, their motivation may have been lowered if their desire to complete the whole task was thwarted by the interruptions. They may have felt that they were in better control of the pencil-and-paper test situation than of the computer situation.

4. Conclusion

Computers are making a strong bid for serious consideration as teaching tools. If using them rather than classical testing methods changes test results, we need to understand the nature of these changes. It is clear that, to the degree that our data are representative, computer administration makes a difference in test results. The reasons for this difference appear to lie in the realm of motivation. Specifically, we expect that variations in partitioning, pacing, and control variables will be significant in accounting for it.

It may be that standard tests rank students low in ability unnecessarily with all the discouragement and erosion of purpose that this labelling implies. There is a strong hint in the data that there are ways to estimate students' learning capacities that are superior to our standard methods for making these estimates.

5. Acknowledgments

The experiment was conducted by Wallace Feurzeig, who performed the initial analysis. Glenn Jones carried out the detailed analysis. Nannette Feurzeig contributed keen insights into the meaning and presentation of the results. George Lukas of BBN and Ralph D'Agostino, BBN consultant and Professor of Mathematics at Boston University, made some invaluable suggestions about the statistical analysis and interpretation of the data. Margaret Morse and Eileen Lynn, reading specialists at Muzzey Junior High School, helped in selecting the pupils used in the study, administered almost all of their tests, managed the scheduling and proctoring, and gave practical suggestions throughout. Melba Jones, reading specialist, contributed time and skill in administering reading tests needed for auxiliary studies. Santo Marino, Principal, and David Terry, Assistant Principal, at Muzzey, generously contributed space and personnel facilities, and as well their good will and counsel.

Table 1. I.Q. and Reading Scores by Pupil

		Reading Tests			
		Grade-level equivalent scores Gates-MacGinitie Given in		Raw Scores Diagnostic (Triggs)	
Pupil	IQ	7th grade	8th grade	Form B	Form D
1	?	3.6	4.8	6	11
2	76	3.4	4.1	8	9
3	82	6.2	7.8	12	11
4	86	4.3	4.8	7	9
5	86	3.5	5.5	10	11
6	87	3.6	4.1	4	8
7	88	5.1	4.1	6	10
8	93	7.8	7.2	10	15
9	94	8.6	7.8	10	14
10	96	5.3	6.5	5	9
11	97	4.5	5.5	10	13
12	98	8.6	7.8	11	11
13	99	?	6.7	5	15
14	99	7.8	6.2	9	9
15	101	8.9	7.2	12	9
16	102	6.2	8.9	14	6
17	105	5.8	6.0	17	13
18	105	4.5	8.2	12	13
19	106	4.3	8.6	10	13
20	107	3.9	2.7	2	12
21	107	8.4	9.2	12	14
22	108	8.6	9.6	18	14

Table 1. I.Q. and Reading Scores by Pupil (cont.)

Pupil	IQ	Reading Test: Grade level equivalent scores, Gates-MacGinitie Given in 7th grade	Reading Test: Grade level equivalent scores, Gates-MacGinitie Given in 8th grade	Reading Test: Raw scores, Diagnostic (Triggs) Form B	Reading Test: Raw scores, Diagnostic (Triggs) Form D
23	108	8.2	3.7	18	15
24	109	8.9	8.4	15	17
25	109	8.9	9.6	18	11
26	110	3.7	4.8	5	17
27	111	6.0	5.5	13	10
28	114	3.2	4.8	8	7
29	116	6.0	3.9	9	10
30	117	2.8	2.6	2	11
31	110	8.9	10.4	13	8
32	111	5.1	3.6	10	14
33	111	4.3	9.6	8	10
34	113	7.2	4.1	6	13
35	115	9.2	10.9	17	12
36	115	5.3	7.6	10	15
37	115	5.8	7.6	13	9
38	116	8.9	9.2	11	9
39	116	6.2	7.8	9	10
40	117	8.6	10.0	14	10
41	117	7.8	2.7	8	9
42	117	12.9	3.1	18	17
43	119	9.2	10.4	12	13
44	120	6.7	8.2	8	9

Table 1. I.Q. and Reading Scores by Pupil (cont.)

		Reading Tests			
		Grade-level equivalent scores		Raw Scores	
		Gates-MacGinitie Given in		Diagnostic (Triggs)	
Pupil	IQ	7th grade	8th grade	Form B	Form D
45	121	6.2	7.8	13	8
46	121	9.6	9.6	12	11
47	122	6.2	5.8	4	14
48	122	8.9	8.9	14	13
49	122	5.3	9.2	15	15
50	125	8.9	10.0	17	14
51	126	3.1	3.1	14	7
52	127	5.5	8.9	11	15
53	127	4.3	7.0	13	16
54	130	8.4	8.6	11	14
55	131	8.2	11.4	16	12
56	132	9.2	10.9	18	16
57	133	7.2	9.2	18	15
58	136	8.9	10.9	9	13
59	140	9.2	10.9	16	16
60	142	6.7	8.6	3	15
61	144	7.4	9.2	14	17

Table 2. Diagnostic Reading Test: Regression of paper-and-pencil scores (Form B) on computer scores (Form D), raw scores

Score: Form D	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Score: Form B	-	-	-	-	-	-	14	8	4	5	6	2	2	6	4	3	13	5	-	-	-
								14	13	7	8	6	16	9	10	5	16	14			
									13	8	9	10	17	10	10	10	18	15			
										8	9	11		10	11	10		18			
										8	13	12		12	12	11					
										9	14	12		12	17	15					
										11		18		14	18	18					
										12				17		18					
										13											

Table 3. Diagnostic Reading Test: Regression of paper-and-pencil scores (Form B) on computer scores (Form D) by numbers, means, and differences of means within septiles of range

Septile of range: Form D	1	2	3	4	5	6	7	
Range: Form D	0-2	3-5	6-8	9-11	12-14	15-17	18-20	Total
Number of Pupils	0	0	6	22	18	15	0	61
Mean Scores: Form B	-	-	11.0	9.6	11.5	12.6	-	11.0
Mean Scores: Form D	-	-	7.3	9.9	13.2	15.7	-	12.1
Difference in Means $\bar{B}-\bar{D}$	-	-	3.7	-0.3	-1.7	-3.1	-	1.1
Regression toward grand mean $\bar{B}-\bar{D}+1.1$	-	-	4.8	0.8	-0.6	-2.0	-	0.0

Table 4. Diagnostic Reading Test: Regression of computer scores (Form D) on paper-and-pencil scores (Form B), raw score

Score: Form B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Score: Form D	-	-	11	15	8	9	10	9	7	9	11	9	9	8	6	15	12	12	11	-	-
			12		14	15	11		9	10	13	11	11	8	7	17	16	13	14		
						17	13		9	10	13	14	11	9	10			14	15		
									9	13	14	15	13	10	13				15		
									10		14		13	16	17				16		
											15		14						17		
											15										

Table 5. Diagnostic Reading Test: Regression of computer scores (Form D) on paper-and-pencil scores (Form B) by numbers, means, and differences of means within septiles of range

Septile of range: Form B	1	2	3	4	5	6	7	
Range Form B	0-2	3-5	6-8	9-11	12-14	15-17	18-20	Total
Number of pupils	2	6	9	15	16	7	6	61
Mean scores: Form D	11.5	13.0	9.7	12.4	10.9	14.1	14.7	12.1
Mean scores: Form B	2.0	4.3	7.2	10.0	12.9	16.1	18.0	11.0
Difference in means: $\bar{D}-\bar{B}$	9.5	8.7	2.5	2.4	-2.0	-2.0	-3.3	1.1
Regression toward grand mean $\bar{D}-\bar{B}-1.1$	8.4	7.6	1.4	1.3	- 3.4	-3.1	-4.4	0.0

Table 6. Gates-MacGinitie Reading Test: Pooled regression of eighth- on seventh- and seventh- on eighth-grade level equivalent scores by numbers, means, and differences of means with septiles of theoretically determined range.

Septile of range	1	2	3	4	5	6	7
Range	1.5-2.9	3.0-4.4	4.5-5.9	6.0-7.4	7.5-8.9	9.0-10.4	10.5-11.9
Number of pairs	4	20	18	18	34	18	5
Mean Scores: Y	4.27	5.28	5.67	7.03	7.48	8.71	8.94
Mean Scores: X	2.70	3.79	5.20	6.57	8.39	9.53	11.00
Difference in means: $\bar{Y}-\bar{X}$ (regression toward the grand mean)	1.57	1.49	0.47	0.46	-0.91	-0.82	-2.06

Table 7. Coefficients of correlation between five test scores for 61 eighth grade pupils

Test	I.Q.	G-M 7th	G-M 8th	D P&P
G-M 7th	.32			
G-M 8th	.41	.52		
D-P-and-P	.29	.57	.50	
D-Comp.	.37	.30	.20	.24

Key:

I.Q.: Stanford-Binet in second grade

G-M 7th: Gates MacCinitie Reading Test, Series E (for grades 7-9), Form 1M, Comprehension Scale, paper and pencil in April of 7th grade.

G-M 8th: Same as G-M 7th in April, 8th grade

D-P-and-P: Diagnostic Reading (Triggs) Test, Upper Level Series (7th Grade to College Freshman Year).
Form B, Administered as a paper-and-pencil test in June, 8th grade.

D-Comp.: Same as D-P-and-P excent Form D, administered on a teletype computer terminal.

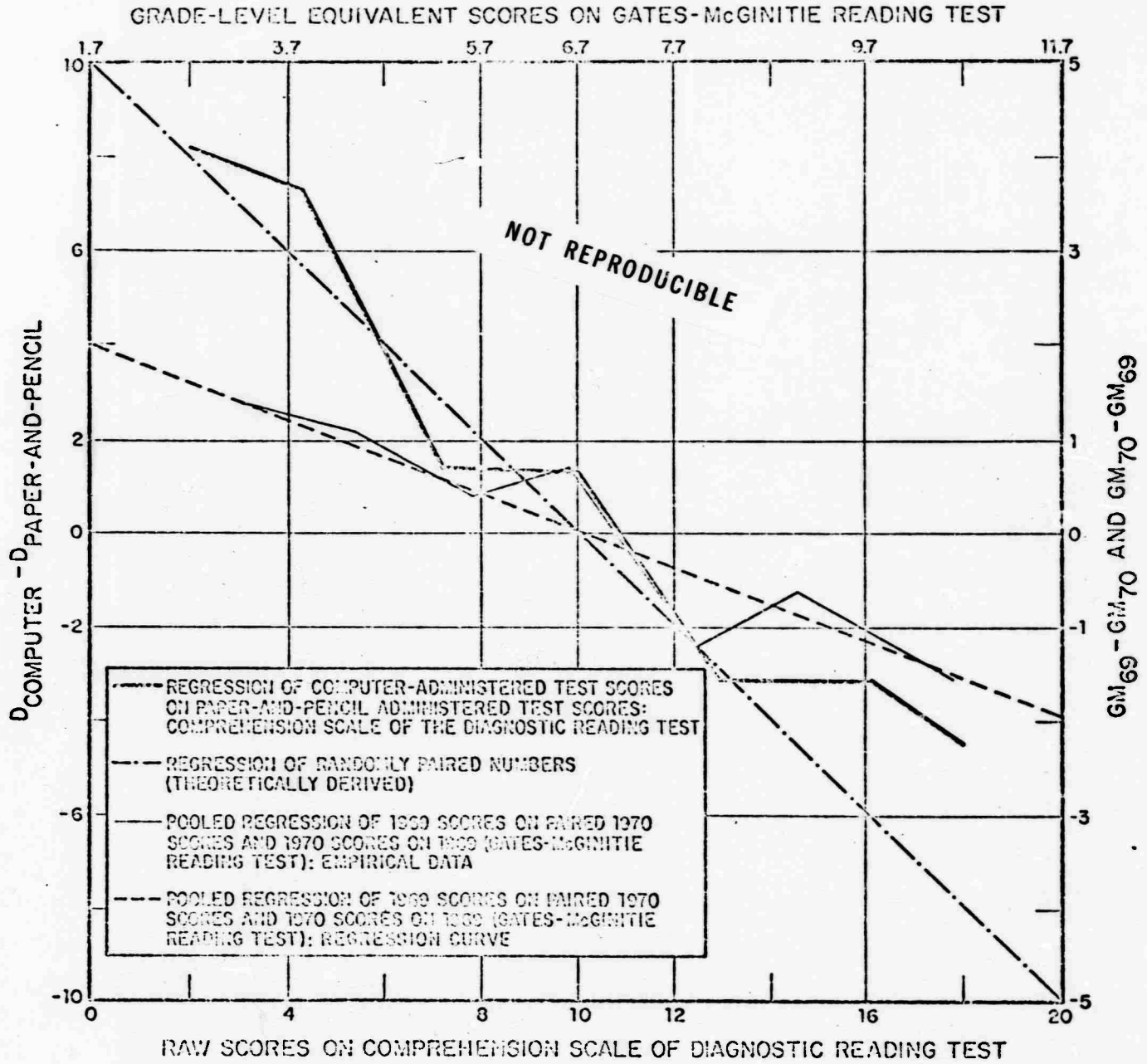
Table 8. Numbers of pupils and mean difference between first and second administrations (second minus first) of comprehension scale on Diagnostic Reading test for pupils having at least one score of five or less or at least one score of 15 or more, by form of test administered first

	Cases with one score in the range of 0-5		Cases with one score in the range of 15-20	
Administered first	N	$\bar{X}$ 2nd-1st	N	$\bar{X}$ 2nd-1st
Form B (paper-and-pencil)	4	+ 8.25	13	- 1.08
Form D (Computer)	4	- 9.50	8	- 2.25

Table 9. Scores on comprehension scales of Gates-McGinitie Reading Test given in the eighth grade and of the Diagnostic Reading Test given by paper-and-pencil and by computer for four students who had computer-terminal experience and for five matched students with little or none, by student and by mean test scores.

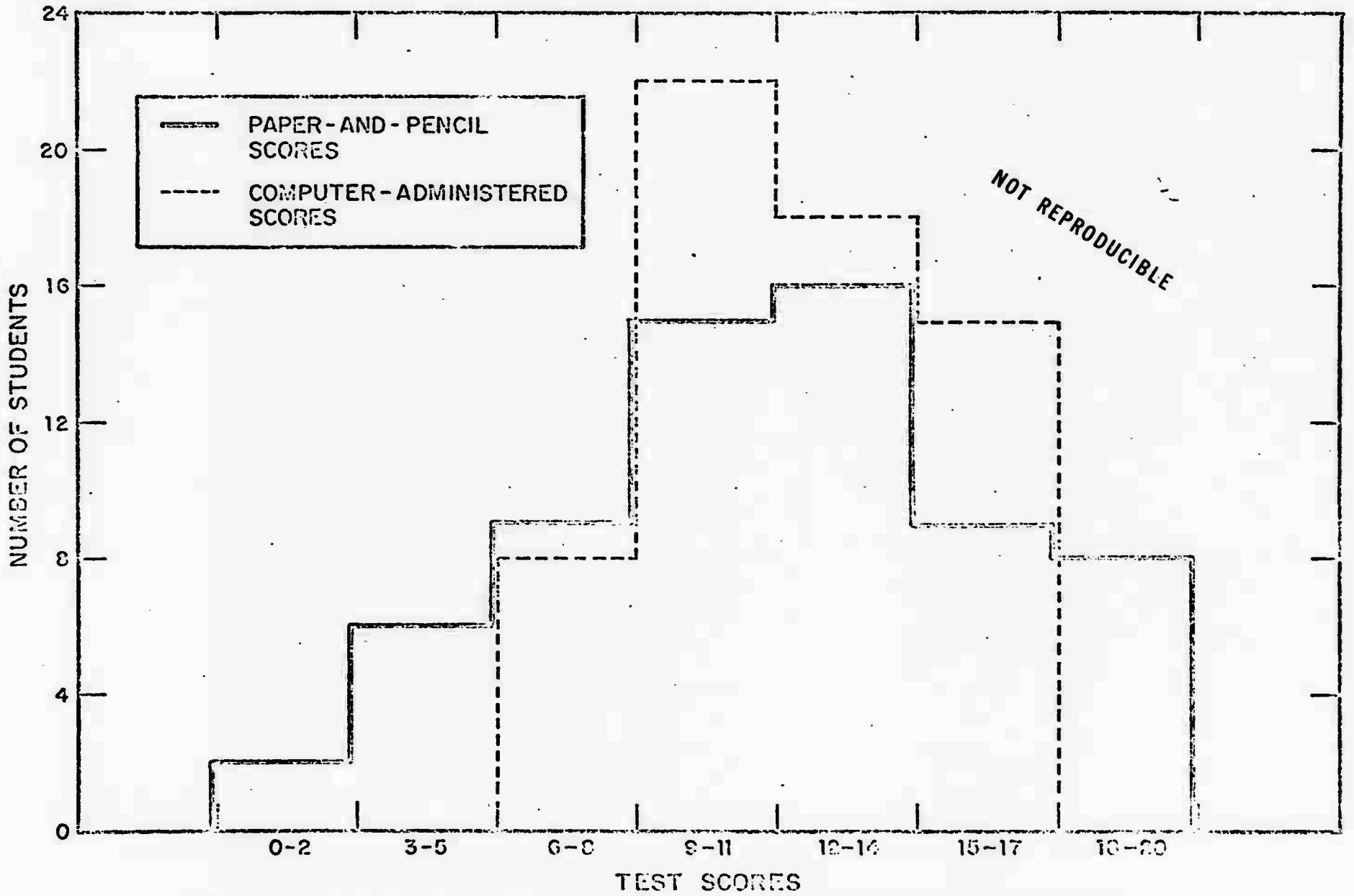
	Experienced				Naive		
	Matching Criterion (D-P&P)	Pupil No.	D Comp.	G-M 8th	Pupil No.	D Comp.	G-M 8th
	2	30	11	2.6	20	12	2.7
	4	6	8	4.1	47	14	5.8
	5	1	15	6.7	10	9	6.5 *
					26	17	4.8 *
	8	28	7	4.8	2	9	4.1
Mean	4.8		10.3	4.8		12.0	4.6
Mean difference (Comp-P&P)			5.5			7.2	

*For purposes of comparison, the scores of the two "naive" pupils who scored five on the Diagnostic paper-and-pencil test have been averaged and treated as single scores.



REGRESSION OF COMPUTER-ADMINISTERED SCORES ON PAPER-AND-PENCIL SCORES: COMPREHENSION SCALE, DIAGNOSTIC READING TEST

Figure 1



DISTRIBUTION OF COMPUTER-ADMINISTERED AND PAPER-AND-PENCIL SCORES: COMPREHENSION SCALE, DIAGNOSTIC READING TEST

Figure 2

References

Campbell, Donald T., and Donald W. Fiske, "Convergent and Discriminate Validation by the Multitrait-Multimethod Matrix," *Psychological Bulletin*, 56: 81-105 (1959).

Committee on Diagnostic Reading Tests, Inc., Diagnostic Reading Tests, Survey Section, Forms B and D, New York, N. Y. (1963).

ETS, Cooperative School and College Ability Tests, Technical Report, Cooperative Test Division, Educational Testing Service, Princeton, N. J., (1957).

Harman, Harry H., Helm, Carl E., and Loye, David E., ed., "Computer Assisted Testing," Conference Proceedings, November 1966, Educational Testing Service, Princeton, N. J. (1968)

Hayes, W. L., *Statistics for Psychologists*. New York: Holt, Rinehart, and Winston (1963).

Roethlisberger, F. J. and Dickson, W. J., *Management and the Worker*. Cambridge: Harvard University Press (1939)

Serwer, Blanche L. and Stolurow, L. M., Computer-Assisted Learning in Language Arts. *Elementary English* 47: 641-650 (1970).

Teachers College Press, Gates-MacGinitie Reading Tests, Survey E, Forms 1,2,3, Forms 1M,2M,3M, Teachers College, Columbia University, New York (1965).

Vinsonhaler, John F., Molineux, J. E., and Rogers, B. G., "An Experimental Study of Computer-Aided Testing," Computer Institute for Social Science Research, Michigan State Univ., East Lansing, Michigan, (November 1965) (Mimeographed).