DOCUMENT RESUME

ED 207 549 IR 009 660

AUTHOR Mayer, Richard E.
TITLE Analysis of a Simple Computer Programming Language: Transactions, Prestatements, and Chunks. Report No. 79-2. Series in Learning and Cognition.
INSTITUTION California Univ., Santa Barbara. Dept. of Psychology.
SPONS AGENCY National Science Foundation, Washington, D.C.
PUB DATE [79]
GRANT SED-77-19875
NOTE 34p.; For a related document, see IR 009 662.

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Computer Science Education; Instructional Innovation; *Learning Processes; Learning Theories; *Programing Languages; *Teaching Methods.
IDENTIFIERS *BASIC Programing Language

ABSTRACT

This discussion of the kind of knowledge acquired by a novice learning BASIC programming and how this knowledge may be most efficiently acquired suggests that people who do programming acquire three basic skills that are not obvious either in instruction or in traditional performance: (1) the ability to analyze each statement into a type of prestatement, (2) the ability to enumerate the transactions involved for each prestatement, and (3) the ability to chunk prestatements into general clusters or configurations. The instructional implications of a psychological analysis of the basic concepts underlying performance in BASIC programming are considered, and an alternative instructional approach--the "transactional approach"--is recommended for teaching programming. This approach involves teaching the underlying concepts of transactions, prestatements, and chunks using a concrete model of the computer, before emphasizing hands-on learning. It is argued that once the student has acquired the relevant subsuming concepts, the relationship between program and output will be more meaningful. Nine references are listed, and appendices include the eight levels of knowledge for BASIC; examples of transactions, prestatements, and chunks and diagrams of the traditional and transactional approaches. Other publications in this report series are listed. (MER)

SERIES IN LEARNING AND COGNITION

Analysis of a Simple Computer Programming Language: Transactions, Prestatements and Chunks

Richard E. Mayer

Report No. 79-2

Preparation of this report was supported by Grant SED 77-19875 from the National Science Foundation

A Psychology of Learning BASIC Computer Programming: Transactions, Prestatements and Chunks

This paper addresses the question: What does a person know following learning of BASIC programming? Several underlying conceptual structures are identified: (1) a transactions is an event that occurs in the computer and involve some operation on some object at some location, (2) a prestatement is a set of transactions corresponding to a line of code, (3) chunks are frequently occurring configurations of prestatements corresponding to several lines of code.

Key words and phrases: BASIC, learning, instruction.

Introduction

This paper deals with two questions: What does a person know when he or she has acquired the ability to write and interpret simple BASIC computer programs? How should a teacher instruct a person so that he or she will most effectively acquire this knowledge?

In response to these questions, this paper suggests a technique for analyzing the knowledge that a learner may acquire in learning BASIC computer programming. The main problem is to determine a system for specifying the basic units of knowledge and the relations among them.

Several researchers have argued for the need to apply the analytic tools of cognitive psychology on an entire subject matter. For example, Greeno [3] has stated: "A further impediment to enthusiasm now is the fragmentary nature of the illustrations of detailed task analysis based on cognitive theory. A more reasonable evaluation may be possible when we can display a relatively complete analysis of the knowledge desired as the outcome of instruction in some subject..." This paper is an attempt to suggest how one might begin to attack this important problem in the subject area of BASIC programming.

BASIC programming was choosen for several reasons. First, it is well-defined subject area which is taught in schools and elsewhere. Second, it is a new subject matter -- less than 15 years old -- that has not been subjected to the intense instructional analysis of more established topics such as mathematics. Third, the users and learners of BASIC are increasingly turning out to be novices who will not become professional programmers. With micro-computers, programmable in BASIC, becoming a part of everyday business and home life, the demands for teaching BASIC to non-professionals will likely increase. Emphasis on instructional methods seems particularly important for learners who will not become professional programmers.

Levels of Knowledge

One of the first steps in attempting to describe the knowledge that one must possess to perform BASIC programming is to determine the unit of knowledge. Traditionally, the main units of knowledge used in instruction have been: (1) the statement such as READ, PRINT, IF, LET, etc., and (2) the program such as a specific program containing the just taught statements.

In order to teach these two units of knowledge, instructional sequences generally contain the following types of frames: (1) statement definition -- text devoted to presenting the format and formal definition of the statment, (2) statement grammar -- text devoted to the grammatical rules relating to a statement such as allowable address labels, etc., (3) program example -- a program that uses the statements and rules described in statement definition and statement grammar frames, (4) program exercises -- questions asking the learner to write or interpret a program containing statements discussed earlier.

It must be pointed out, however, that the statement and the program are just two levels of knowledge. The main thesis of this paper is that there are several possible levels that go below the statement as the unit of knowledge, and several possible levels about the statement. In general, instructional sequences for BASIC have not fully exploited these alternative levels; however, a careful analysis of "what is learned" in BASIC programming may indicate that learning occurs on levels other than those taught. These levels are suggested in Table 1. Each level of knowledge about BASIC programming will be discussed in turn.

- - - - - - - - - - - - - -

Insert Table 1 about here

- - - - - - - - - - - - - -

Machine level. The lowest level of knowledge suggested in Table 1 is the machine level. Presumably, a person could know each specific, single change that may occur within the actual hardware of the computer, as indicated by machine language statements. Many instructional manuals provide an introductory chapter that describes the nature of electromagnetic fields, logic circuits, octal code and the like. However, this level if knowledge is rarely used by novices as a meaningful context for further learning; more often, it is treated by student and instructor alike as extra technical information that is not necessary for learning programming. Indeed, BASIC programming does not require an understanding of electronics or hardware or even octal code. Thus this level of knowledge, while important to the potential professional, may be too detailed for the novice who is learning BASIC for the first time.

Transactions. An alternative to the machine level is the next level in Table 1, the transaction level. Transactions are not tied to the actual hardware of the computer, but are related to the general functions of the computer that underlie statements. A transaction is a unit of programming knowledge in which a general operation is applied to an object at a general location. A transaction consists of three parts:

(1) operation -- such as MOVE, FIND, CREATE, DESTROY, DECIDE, COMBINE

(2) object -- such as number, program, pointer, program line

(3) location -- such as input stack, memory address, program space, output screen, keyboard, file

An example of a transaction is: move the number that is on the top of the input stack to the finished stack (OP: MOVE, OB: number, LOC: input stack), or create a certain number in memory space A1 (OP: CREATE, OB: number, LOC: memory space), or move the line pointer to the next line of the program (OP: MOVE, OB: pointer, LOC: program space).

In general, the transaction level has not been exploited in instructional manuals. A notable exception is that many mannuals describe memory as a set of eraseable blackboards; this analogy helps clarify one of the locations. However, our research has suggested that the transactional level may be crucial in providing understanding for novices [5,6]. Transactions provide a means, for the novice, of "explaining" what is going on inside the computer when a particular statement is executed and of relating the new technical language that he or she must learn to general operations, locations and objects that he or she is already familiar with.

Table 2 presents a list of the major transactions involved in learning elementary BASIC programming. Note that these transactions are building blocks from which statements may be made.

- - - - - - - - - - - -

Insert Table 2 about here

- - - - - - - - - - - -

It is important to point out that transactions do not require an understanding of machine-level hardware and operations. In our studies we have used analogies to describe the locations involved in transactions. For example, some of the main locations can be successfully described as follows:

(1) ticket window -- data cards are placed in a pile outside a ticket window and are moved inside, one at a time, as each is processed,

(2) memory scoreboard -- memory is made up of many squares in an eraseable chalkboard with a label permanently attached to each,

(3) output pad -- messages are written on successive lines of a note pad,

(4) program list and pointer arrow -- the program is like a recipe or shopping list and the pointer arrow points to the current line on the program,

(5) file cabinet -- programs are stored in a file cabinet in alphabetical order by name.

Since the objects and operations are generally familiar to the learner, it may not be as important to provide analogies for them.

Any line of code may be translated into a series of transactions. By describing a statement in terms of transactions, in addition to giving its formal definition, the teacher gives the learner a way of understanding what is going on inside the "computer" when a statement is executed.

In addition, evaluation of learning based on transactions may provide a more effective way of locating areas for remedial work. For example, a test could ask a student to list the transactions for a statement or a simple program. For example, if the learner has not yet mastered the concept of destructive read in and non-destructive read out, the transactions listed for a READ or PRINT statement would indicate this problem.

In short, the underlying structure of BASIC may be made up of transactions. Transactions are powerful units of knowledge because they are few in number and yet can cover all of the elementary BASIC statements.

Prestatements. The third level in Table 1 concerns prestatements. Prestatements are sub-categories of statements; they are more specific that the general category of statements that all share the same name. For example, the LET statement really includes several quite distinct and different types of prestatements: to set a counter such as LET X = 1, to perform an arithmetic operation on a counter such as LET X = X/2, or to perform an arithmetic operation and store the answer such as LET X = 5/2, etc.

Each prestatement has its own unique series of transactions; thus the transactions for a counter set LET are not the same as the series of transactions for an arithmetic computation LET. However, for any prestatement, the

list of transactions is the same regardless of specific numbers or memory locations; thus the series of transactions is the same for any two counter set LETs such as LET C = 5 or LET X = 999.

One problem that many learners face is that the prestatements are not made clear at the onset of learning. Each statement may actually be a family of quite different prestatements, and the nature of the prestatements is often not clarified explicitly in the instruction. Some of the major prestatements are given in Table 3.

- - - - - - - - - - - - - - -

Insert Table 3 about here

- - - - - - - - - - - - - - -

Landa [4] has suggested that students should be given algorithms for learning different cases in Russian grammar. The same approach is possible with respect to learning prestatements; each has its own unique series of transactions and these could serve as a way generating an algorithm for locating prestatements.

Statements. The statement level is the fourth level shown in Table 1, and constitutes the traditionally lowest level. A statement is a class of one or more prestatements all sharing the same name. Table 4 presents the relationship between statements and prestatements; further, Table 3 presents the relation between prestatements and transactions.

While this is the most dominant level for instruction, full comprehension of BASIC may also involve the lower levels as well. The rules for dividing a statement into a set of unique prestatements, and of analyzing each prestatement into a series of transactions are skills that may underlie the statement level.

- - - - - - - - - - - - - - -

Insert Table 4 about here

- - - - - - - - - - - - - - -

Mandatory chunks. The next level after statements in Table 1 is the mandatory chunk level. A mandatory chunk is a series of two or more statements that must occur in some configuration. For example, a READ statement always requires a complementary DATA statement, or a FOR statement always requires a NEXT statement. Thus some statements may be learned as members of a larger chunk.

Learning may involve establishing a large repertoire of chunks; however, this chunking process should begin with mandatory chunks. Most textbooks and manuals do explicitly note the existence of mandatory chunks, but many do not build further on the concept of a chunk.

Examples of major mandatory chunks are given in Table 5.

- - - - - - - - - - - - -

Insert Table 5 about here

- - - - - - - - - - - - -

Basic non-mandatory chunks. The next level in Table 1 is for basic non-mandatory chunks. A non-mandatory chunk is a series or configuration of pre-statements that is often used in a variety of programs to accomplish some general goal -- i.e. some larger series of transactions than a single pre-statement. For example, IF and GOTO statements may be used in several different configurations. These are listed in Table 6. Certainly these chunks are not meant to be an exhaustive list but only as examples of how chunks may be built. Note that each serves as a sort of super-statement in the sense that a long list of transactions is called for.

Looping is one of the most difficult concepts underlying BASIC programming, and many of the non-mandatory chunks involve different loop configurations. By spelling out the different types of loop configurations, learners may be

better able to tell which one is involved in a given program. The main characteristics common to all loop configurations are: (1) initial conditions -- the state of the computer at the onset of the loop, (2) exit conditions -- the conditions under which the computer will shift out of the loop to another line on the program, (3) body of the loop -- the statements such as LET or PRINT etc., that are executed on each cycle through the loop, and (4) reset -- the statements such as GOTO that allow the computer to move from the end of the loop back to the beginning. Each of the loop configurations in Table 6 may be related to these four characteristics.

- - - - - - - - - - - - -

Insert Table 6 about here

- - - - - - - - - - - - -

Higher non-mandatory chunks. The next level in Table 1 calls for even higher chunks. An example is given in Table 7. This level is simply an extension of the one before it and has no hard boundary line with it. As a learner gains more experience the size and number of chunks (or "super-statements) he knows will grow. This approach is related to the "structured programming" revolution -- the idea of dividing a program in recognizable chunks that are separate and removable parts of a program. Because this technique has already received much attention, and because it is more related to advanced learning in programming, it will not be covered here.

- - - - - - - - - - - - -

Insert Table 7 about here

- - - - - - - - - - - - -

Programs. The highest level in Table 1 is the program level. However, the program level is directly analyzable into a set of chunks and statements. It must be noted that this paper does not deal with the important question of

what skills are required to generate a program, e.g. heuristics and algorithms. This component, of course, is required before a useful instructional science of BASIC programming may be established.

Implications for Theory and Instruction

This paper began by asking what knowledge is acquired by a novice learning BASIC programming, and how can this knowledge be most efficiently acquired.

In answering the first question, this paper has suggested that people acquire three basic skills that are not obvious either in instruction nor in traditional performance. These skills are: the ability to analyze each statement into a type of prestatement, the ability to enumerate the transactions involved for each prestatement, and the ability to chunk prestatements into general clusters or configurations. This paper has shown the psychological structures that may be involved in understanding transactions, relating transactions to prestatements, relating prestatements to statements, and relating restatements to chunks.

This psychological analysis of the basic concepts underlying performance in BASIC programming leads to some instructional implications stated in the second question. The traditional method of instruction for BASIC programming is to help the learner see the relation between a list of statements (i.e. program) and a certain output. Generally, statements are defined and examples are given; the learner is encouraged to engage in "hands on" experience such as typing in programs and seeing what output comes out. This approach does not emphasize the psychological concepts that underlie the relationship between program and output.

An alternative instructional approach that could be called the "transactional approach", involves emphasizing the underlying concepts of

transactions, prestatements, and chunks before emphasizing "hands on" learning. Once the student has acquired the relevant subsuming concepts [ ] then the relationship between program and output will be more meaningful. The two types of instructional procedures are summarized in Table 8. It should be noted, of course, that some students may form the internal concepts of transactions and chunks and prestatements even though they have not been formally presented. The transactional approach, however, makes sure they are then acquired by the novice.

- - - - - - - - - - - - - - -

Insert Table 8 about here

- - - - - - - - - - - - - - -

Traditionally, the statement level and the program level of knowledge have been emphasized. The other levels -- transactions, prestatements and chunks -- have not been as fully exploited as they could have been. The main implication of this report is that these three levels, and their relations to programs and statements should be more fully used in instruction for BASIC with novices. For example, learners must learn which transactions go with which prestatements, which prestatements go with which statements, which statements go with which chunks, and so on.

Based on the foregoing analysis, the following recommendations for instruction can be offered:

(1) Explicitly teach the basic transactions involved in elementary BASIC, including locations, objects and operations. To enhance learning in novices, a concrete or familiar model of the computer should be introduced early in learning and used throughout learning. The model should provide a familiar context for describing the basic locations, objects and operations in the system.

(2) Explicitly teach the sequence of transactions for each prestatement. Thus, in addition to formal definitions and grammatical rules, instruction should include the relationship beteen a prestatement and its underlying transactions.

(3) Explicitly distinguish among the different prestatements that share the same statement name. Students should receive practice in recognizing different types of prestatements for the same statement name. Since each type of prestatement has a different set of transactions, these should be made clear.

(4) Explicitly present mandatory chunks. This will allow novices to see the relation among key prestatements.

(5) Explicitly present basic non-mandatory chunks.

(6) Evaluate and remediate by asking learners to list transactions for a given code. Remediation may thus be based on correctly holes in knowledge at the transactional level, such as a failure to use "destructive memory read in", etc.

(7) Emphasize techniques for generating subroutines and structured programming. This will help the novice to develop additional chunks. Since there are already many fine, more advanced treatments of structured programming there is no need for an exa[illegible]e.

It should be noted that these recommendations are in addition to the many fine instructional procedures already in use, such as encouraging active participation from the learner, using humor, and clearly stating the grammatical rules.

Evidence

Will these recommendations result in improvements in the way novices learn to interact with computers? There are encouraging empirical results which suggest that the answer is yes [5, 6, 7, 8, 9].

For example, in a series of experiments [5], college students who had no prior experience with computer programming were taught a simplified version of BASIC, including the statements READ, PRINT, LET, GO TO, IF and END. For half of the learners, the instructional text referred to a concrete model of the computer and explained each statement as a set of transactions within the model. For the other half, the same basic information was presented but instead of giving the model or transactions the booklet emphasized the grammatical rules. Results indicated that the model group performed better than the rule group on problems requiring creative programming but not on easy problems. In particular, the model group excelled on interpreting what a program would do and on writing long looping programs. In addition, there was evidence that the model helped the low ability students the most.

A follow-up study [6] used an actual concrete model of the computer that the learners could view or operate. Students were encouraged to see how each statement would be translated into changes in the concrete model. Results of a series of studies showed that there were major differences between learners who were introduced to the model before learning (and encouraged to use it during learning) and those who were introduced to it after learning. The before group performed better than the after group, and especially performed better on problems requiring creative programming. Apparently, being able to understand each statement as a set of transactions within a familiar context resulted in broader, more meaningful learning.

More recently, Mayer & Bromage [9] asked college students who had no experience with computers to read a booklet similar to that used in the earlier studies. As in the above study, some learners were given an introduction that included a concrete model of the computer and hints for how to relate the to-be-learned statements to it, while other learners received the same information

after learning. As a test, the students were asked to write down all they could remember about selected statements. The before group remembered more of the conceptual ideas while the after group remembered more of the technical facts; in addition, the before group tended to add in more meaningful inferences and references to the model while the after group made more vague statements and inappropriate intrusions. These results are consistent with earlier results: the conceptual ideas and good inferences of the before group would be expected to support creative programming while the specific facts recalled by the after group would serve best in simple retention questions.

In another study [8], the same results were obtained using a different programming language, namely a file management system discussed by Gould & Asher [2]. In this case, a different model and different transactions were used, but the results were the same as with earlier studies.

Finally, a study [7] was conducted in which the instructional information was presented in logical order or in a random order. A group of learners who read about the model and how to relate it to transactions before they read the text performed better than a control group for the random passage but not the logical. Apparently, the use of transactions is most important when the material is poorly organized.

These studies may be summarized by saying that providing information about transactions (and using a concrete model to explain them) increases performance especially on creative programming tasks, especially for low ability subjects, and especially when material is not well organized.

References

1. Ausubel, D. P. Educational Psychology: A Cognitive View. New York: Rinehart & Winston, 1968.

2. Gould, J. D. & Ascher, R. N. Query by non-programmers. Paper presented at convention of American Psychological Association, 1974.

3. Greeno, J. G. Cognitive objectives of instruction: Theory of knowledge for solving problems and answering questions. In D. Klahr (Ed.), Cognition and Instruction. Hillsdale, N.J.: Erlbaum, 1976, p. 158.

4. Landa, L. N. Algorithmization in Learning and Instruction. Englewood Cliffs, N.J.: Educational Technology Publications, 1974.

5. Mayer, R. E. Different problem-solving competencies established in learning computer programming with and without meaningful models. Journal of Educational Psychology, 1975, 67, 725-734.

6. Mayer, R. E. Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. Journal of Educational Psychology, 1976, 68, 143-150.

7. Mayer, R. E. Advance organizers that compensate for the organization of text. Journal of Educational Psychology, 1979, 71, in press.

8. Mayer, R. E. Elaboration techniques and advance organizers that affect technical learning. Under editorial review.

9. Mayer, R. E. & Bromage, B. Different recall protocols for technical text due to sequencing of advance organizers: Assimilation versus addition encoding. Under editorial review.

Footnote

Preparation of this paper was supported by Grant SED77-19875 from the National Science Foundation. Requests for reprints and instructional materials should be sent to: Richard E. Mayer, Department of Psychology, University of California, Santa Barbara 93106.

Table 1

Levels of Knowledge for BASIC

1. MACHINE
2. TRANSACTION
3. PRESTATEMENT
4. STATEMENT
5. MANDATORY CHUNK
6. BASIC NON-MANDATORY CHUNK
7. HIGHER CHUNK
8. PROGRAM

Table 2

Some Basic Transactions

Operation	Object	Location	
FIND	Number	Input Stack	Find the next number waiting in the input stack.
FIND	Number	Memory Address	Find the number in a particular memory space.
FIND	Number	Keyboard	Find the number just entered in the keyboard.
FIND	Number	Program List	Find the number indicated in a particular place in the program.
FIND	Line	Program List	Find a particular line in the program.
FIND	Program	Program File	Find a particular program in the file.
FIND	Command	Keyboard	Find a particular word just entered in the keyboard.
FIND	Place in Line	Input Stack	Find the end of line at input stack.
MOVE	Number	Input Stack	Move the next number at the input window to the finished pile.
MOVE	Line Pointer	Program List	Move the line pointer to a particular line on the program.
MOVE	Program	Program List	Move a particular program to the program space.
MOVE	Printer Pointer	Output Screen	Move a printer pointer to a particular zone on the output screen.
CREATE	Number	Memory Address	Write a number in a particular memory space.
CREATE	Number	Input Stack	Put a number in line at the input window.
CREATE	Number	Output Screen	Write a number on the output screen.
CREATE	Program	Program List	Write a program into the program space.
CREATE	Word	Output Screen	Write a message (such as OUT OF DATA) on output screen.
CREATE	Question Mark	Output Screen	Write a ? on the output screen.
DESTROY	Number	Memory Address	Erase the contents of a particular memory space
DESTROY	Program	Program List	Destroy a particular program that is in the program space.
DESTROY	Program	Program File	Destroy a particular program that is in the file.
DECIDE	Number	Memory Address	Apply a particular logical operation to numbers in memory.
DECIDE	Numbers	Program List	Apply a particular operation to numbers in program.
COMBINE	Number	Memory Address	Apply a particular arithmetic operation to numbers in memory.
COMBINE	Numbers	Program List	Apply a particular arithmetic operation to numbers in program.
ALLOW	Command	Keyboard	Execute next statement entered on keyboard.
ALLOW	Command	Program List	Execute next statement pointed to by line pointer.
DISALLOW	Command	Program List	Do not execute next statement pointed to by line pointer.

Table 3

Transactions Involved in Selected Prestatements

Prestatement	Operation	Object	Location	English Translation
READ address (Single Address READ)	FIND	Number	Input Stack	Find the next number waiting at the input stack.
	DECIDE	Number	Input Stack	If there are no numbers in the input stack,
	CREATE	Words	Output Screen	If no more data print "OUT OF DATA" on screen,
	ALLOW	Command	Keyboard	and wait for a new command from the keyboard.
	MOVE	Number	Input Stock	Otherwise, move the number at the input window to the finished pile.
	FIND	Number	Memory	Find the number in the memory space indicated on the READ statement.
	DESTROY	Number	Memory	If more data Erase that number from the memory space.
	CREATE	Number	Memory	Write the new number into the memory space.
	MOVE	Line Pointer	Program	Go on to the next statement,
	ALLOW	Command	Program	and do what it says.
DATA number (Single Datum DATA)	FIND	Number	Program	Find the number indicated on the DATA statement.
	FIND	Place-in-line	Input Stock	Find the end of the line at the input window.
	CREATE	Number	Input Stock	Put the number at the end of the line at the input window.
	MOVE	Line Pointer	Program	Go on to the next statement,
	ALLOW	Command	Program	and do what it says.
PRINT address (Single Address PRINT)	FIND	Number	Memory	Find the number in the memory address indicated.
	CREATE	Number	Output Screen	Write that number on the next available space on the output screen.
	MOVE	Line Pointer	Program	Go on to the next statement,
	ALLOW	Command	Program	and do what it says.
END	CREATE	Word (READY)	Output Screen	Write "READY" on the output screen.
	ALLOW	Command	Keyboard	Wait for a new command from the keyboard.
LET address = number (Counter Set LET)	FIND	Number	Program	Find the number indicated on the right of the equals.
	FIND	Number	Memory	Find the number in the memory space indicated on the left of the equals.
	DESTROY	Number	Memory	Erase the number in that memory space.
	CREATE	Number	Memory	Write the new number in that memory space.
	MOVE	Line Pointer	Program	Go on to the next statement,
	ALLOW	Command	Program	and do what it says.

Table 3 (Continued)

Prestatement	Operation	Object	Location	English Translation
INPUT address (Single Number INPUT)	CREATE	Question Mark	Output Screen	Write a "?" on the output screen
	ALLOW	Command	Output Screen	Wait for a number to number to be entered from the keyboard, followed by depression of the RETURN key.
	CREATE	Number	Output Screen	Write the entered number on the output screen next to "?"
	FIND	Number	Output Screen	Find that number that was just entered.
	FIND	Number	Memory	Find the number in the memory space indicated in the input statement.
	DESTROY	Number	Memory	Erase the number in that memory space.
	CREATE	Number	Memory	Write in the new number.
	MOVE	Line Pointer	Program	Go on to the next statement,
	ALLOW	Command	Program	and do what it says.
GO TO line	FIND	Line	Program	Find the line on the program indicated in the GO TO statement.
	MOVE	Line Pointer	Program	Start working on that statement, ignoring all others
	ALLOW	Command	Program	inbetween, and do what it says.
IF address = number THEN line (Counter Match IF)	FIND	Number	Program	Find the number indicated to the right of the equals in the IF statement.
	FIND	Number	Memory	Find the number stored in the memory space indicated to the left of the equals
	DECIDE	Numbers	Program & Memory	If the numbers match,
	FIND	Line	Program (If condition satisfied)	find the line that has the number indicated after THEN,
	MOVE	Line Pointer	Program (If condition satisfied)	and start working on that statement, ignoring all
	ALLOW	Command	Program (If condition satisfied)	and do what it says.
	MOVE	Line Pointer	Program (If condition not satisfied)	Otherwise, go on to the next statement under this IF statement,
	ALLOW	Command	Program (If condition not satisfied)	and do what it says.
NEW	FIND	All Lines	Program	Find the program that is now in the computer.
	DESTROY	All Lines	Program	Erase that program.
	ALLOW	Commands	Keyboard	Wait for program to be typed in on keyboard.
	CREATE	Lines	Program	Copy each line of the program in the space inside the
	CREATE	Lines	Screen	computer, and on the screen.
	Allow	Command	Keyboard	Wait for an operating command.
RUN	FIND	Top Line	Program	Find the first line of the program in the computer
	MOVE	Pointer	Program	Move the pointer arrow to this line
	ALLOW	Command	Program	Execute this statement, and work down from there.
STOP	DISALLOW	Command	Program	Do not execute the next statement in the program.
Control/C	ALLOW	Command	Keyboard	Wait for an operating command from the keyboard.

Table 4

Prestatements Involved in Selected Statements

Statement	Type of Prestatement	Format of Prestatement	Prestatement
LET	Counter Set to Number	LET address = number	LET X = 2
LET	Counter Set to Address	LET address = address	LET X = Y
LET	Arithmetic	LET address = number operator number	LET X = 2*4
LET	Formula	LET address = address operator address	LET X = Y*Z
LET	Mixed Formula	LET address = address operation number	LET X = Y*4
LET	Increment/Decrement	LET address = same address operation number	LET X = X+1
PRINT	Single Message	PRINT "message"	PRINT "55"
PRINT	Single Address	PRINT address	PRINT A1
PRINT	Multiple Address	PRINT address list	PRINT A1,A2,A3
PRINT	Line Space	PRINT (blank)	PRINT

Table 5

Some Mandatory Chunks

READ-DATA Chunk

Rule: Every READ statement requires a corresponding DATA statement.

Example:

```
READ A1

DATA 5
```

Program-END Chunk

Rule: Every program should have an END statement.

Example:

```
Program

END
```

FOR-NEXT Chunk

Rule: Every FOR requires a NEXT somewhere beneath it, and every NEXT requires a corresponding FOR somewhere above it.

Example:

```
FOR X = 1 to 5 STEP 2

NEXT
```

IF-Line Chunk

Rule: Every IF statement requires that there be a line on the program corresponding to the number after them.

Example:

```
10 IF X=2 THEN 50

50 (Must have some statement)
```

Table 6

Some Basic Non-Mandatory Chunks for Loops

Repeat READ Loop

READ DATA	10 20	READ X DATA 6, 7, 8, 9, 10	The INITIAL CONDITION is that numbers are waiting at the input window and one is read in.
			The EXIT CONDITION is that there are no more numbers at the input window. If the computer tries to read but there are no more numbers left it will end the program and print OUT OF DATA.
(more statements such as LET or IF or PRINT)	30 40	LET Y = 6+2 PRINT X, Y	The BODY OF THE LOOP consists of a series of statements such as LET or IF or PRINT which operate on the number that is read in.
GO TO (line with READ)	50	GO TO 10	The RESET statement sends the pointer arrow back to the original state so the next number will be processed in the same loop.
END	60	END	

Branch Loop

READ DATA	10 20	READ X DATA 2, 99, 6, 32, 4	The INITIAL CONDITION is that numbers are waiting at the input window and one is read in.
IF-THEN (Branch 2)	30	IF X>30 THEN 60	The DECISION CONDITION is whether the number meets some some criteria. If so, the pointer shifts to Branch 2; if not, it goes on with Branch 1.
(statements for Branch 1 such as LET and PRINT)	40	PRINT X	The BODY OF THE LOOP has two parts: the body of the loop for Branch 1 and the body of the loop for Branch 2.
GO TO (line with READ)	50	GO TO 10	The RESET sends the pointer arrow back to the same point in the program for both branches.
(statements for Branch 2 such as LET and PRINT)	60 70	LET X = X/2 PRINT X	
GO TO (line with READ)	80	GO TO 10	
END	90	END	

29

Table 6 (continued)

Wait for a Data Number

READ DATA	10 20	READ X DATA 5, 12, 72, 6, -1	The INITIAL CONDITION is that numbers are waiting at the input window, and one is read in
IF-THEN (exit line with END)	30	IF X<0 THEN 60	The EXIT CONDITION is that if a certain number (e.g. a negative number) is read in, then the pointer arrow will shift out of the loop.
(more statements)	40	PRINT X	The BODY OF THE LOOP consists of one or more statements after the EXIT decision but before the RESET
GO TO (line with READ)	50	GO TO 10	The RESET puts the pointer arrow back to READ and allows another cycle through the loop
(more statements)	60	END	

Wait for a Counter

Let C = 0	20	LET C = 1	The INITIAL CONDITION is that the counter is set to zero or one (or some other value)
IF C = ___ THEN (exit line such as END)	30	IF C = 5 THEN 60	The EXIT CONDITION is that if the counter meets a certain criteria the pointer arrow will shift down out of the loop
(more statements)	40 50 60	READ X DATA 5, 20, 23, 6, 7, 10 PRINT X	The BODY OF THE LOOP contains statements such as LET and PRINT
LET C = C+1 GO TO (line with IF)	70 80	LET C = C+1 GO to 30	The RESET involves incrementing the counter and shifting back to the top of the loop
	90	END	

Table 7

Higher Order Chunks

READ-DATA - Operation Chain - PRINT - END

Rules: When a member is read it is usually either printed, or it is operated on with the result printed. Something is usually printed before the program ends.

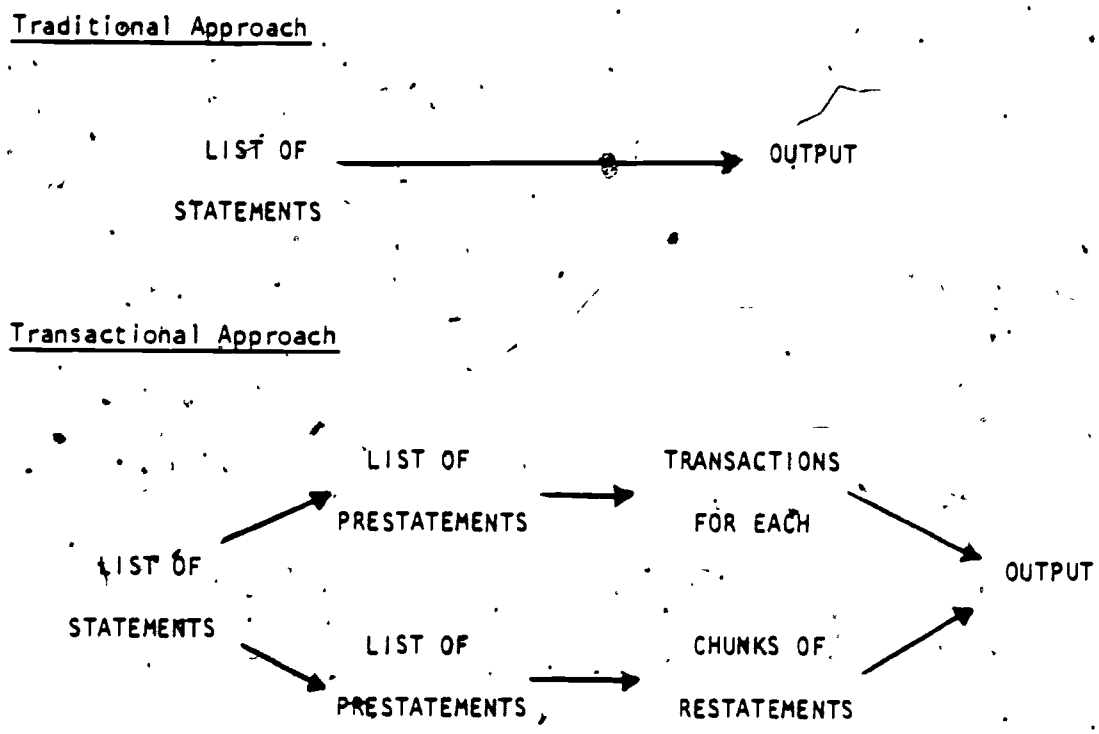
Examples:

1. READ X
2. DATA 4
3. LET X = X ÷ 2
4. IF X > 4 THEN 60
5. PRINT X
6. END

Table 8

Two Approaches to Teaching the Relationship

Between a Program and It's Output



TECHNICAL REPORT SERIES IN LEARNING AND COGNITION

Report No.	Authors and Title
78-1	Peper, R. J. and Mayer, R. E. Note-taking as a Generative Activity. (Journal of Educational Psychology, 1978, 70, 514-522.)
78-2	Mayer, R. E. & Bromage, B. Different Recall Protocols for Technical Text due to Advance Organizers. (Journal of Educational Psychology, 1980, 72, 209-225.)
79-1	Mayer, R. E. Twenty Years of Research on Advance Organizers: (Instructional Science, 1979, 8, 133-167.)
79-2	Mayer, R. E. Analysis of a Simple Computer Programming Language: Transactions, Prestatements and Chunks. (Communications of the ACM, 1979, 22, 589-593.)
79-3	Mayer, R. E. Elaboration techniques for Technical Text: An Experimental Test of the Learning Strategy Hypothesis. (Journal of Educational Psychology, 1980, 72, in press.)
80-1	Mayer, R. E. Cognitive Psychology and Mathematical Problem Solving. (Proceedings of the 4th International Congress on Mathematical Education, 1980.)
80-2	Mayer, R. E. Different Solution Procedures for Algebra Word and Equation Problems.
80-3	Mayer, R. E. Schemas for Algebra Story Problems.
80-4	Mayer, R. E. & Bayman, P. Analysis of Users' Intuitions About the Operation of Electronic Calculators.
80-5	Mayer, R. E. Recall of Algebra Story Problems.
80-6	Bromage, B. K. & Mayer, R. E. Aspects of the Structure of Memory for Technical Text that Affect Problem Solving Performance.
80-7	Klatzky, R. L. and Martin, G. L. Familiarity and Priming Effects in Picture Processing.
81-1	Mayer, R. E. Contributions of Cognitive Science and Related Research on Learning to the Design of Computer Literacy Curricula.
81-2	Mayer, R. E. Psychology of Computing Programming for Novices.
81-3	Mayer, R. E. Structural Analysis of Science Prose: Can We Increase Problem Solving Performance?
81-4	Mayer, R. E. What Have We Learned About Increasing the Meaningfulness of Science Prose?