DOCUMENT RESUME

ED 049 634 EM 008 881

AUTHOR	Friend, Jamesine
TITLE	INSTRUCT Coders' Manual.
INSTITUTION	Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY	National Aeronautics and Space Administration, Washington, D.C.; Office of Education (DHEW), Washington, D.C.
REPORT NO	TR-172
PUB DATE	1 May 71
GRANT	OEG-0-70-4797(607)
NOTE	111p.; Psychology Series
EDRS PRICE	EDRS Price MF-$0.65 HC-$6.58
DESCRIPTORS	*Computer Assisted Instruction, Curriculum Development, *Manuals, Programing, *Programing Languages
IDENTIFIERS	*INSTRUCT (Coding Language)

ABSTRACT

The coding language INSTRUCT is a high-level programing language designed for programing computer-assisted instruction lessons. As it is presently implemented on the PDP-10 computer, a "lesson processor" transforms the INSTRUCT lessons into a numeric code that can be understood by a teaching program called INST. INST controls the interaction between the student and the computer at the time the student is taking a programed lesson. The main steps in preparing an INSTRUCT lesson are: coding the lesson, assembling the lesson, correcting assembly errors, reassembling, loading, and debugging. This manual is designed both as an instructional manual for beginning coders and as a reference manual for the INSTRUCT coding language. It provides an overview of the language, a definition of the INSTRUCT commands, directions for processing and debugging INSTRUCT lessons, and instructions for advanced coding techniques which expand the routines available from the INST program. (JY)

INSTRUCT CODERS' MANUAL

BY

JAMESINE FRIEND

TECHNICAL REPORT 172

MAY 1, 1971

PSYCHOLOGY SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

STANFORD UNIVERSITY

STANFORD, CALIFORNIA

TECHNICAL REPORTS

PSYCHOLOGY SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

(Place of publication shown in parentheses; if published title is different from title of Technical Report, this is also shown in parentheses.)

(For reports no. 1 - 44, see Technical Report no. 125.)

50 R. C. Atkinson and R. C. Calfee. Mathematical learning theory. January 2, 1963. (In B. B. Wolman (Ed.), Scientific Psychology. New York: Basic Books, Inc., 1965. Pp. 254-275)

51 P. Suppes, E. Crothers, and R. Weir. Application of mathematical learning theory and linguistic analysis to vowel phoneme matching in Russian words. December 28, 1962.

52 R. C. Atkinson, R. Calfee, G. Sommer, W. Jeffrey and R. Shoemaker. A test of three models for stimulus compounding with children. January 29, 1963. (J. exp. Psychol., 1964, 67, 52-58)

53 E. Crothers. General Markov models for learning with inter-trial forgetting. April 8, 1963.

54 J. L. Myers and R. C. Atkinson. Choice behavior and reward structure. May 24, 1963. (Journal math. Psychol., 1964, 1, 170-203)

55 R. E. Robinson. A set-theoretical approach to empirical meaningfulness of measurement statements. June 10, 1963.

56 E. Crothers, R. Weir and P. Palmer. The role of transcription in the learning of the orthographic representations of Russian sounds. June 17, 1963.

57 P. Suppes. Problems of optimization in learning a list of simple items. July 22, 1963. (In Maynard W. Shelly, II and Glenn L. Bryan (Eds.), Human Judgments and Optimality. New York: Wiley. 1964. Pp. 116-126)

58 R. C. Atkinson and E. J. Crothers. Theoretical note: all-or-none learning and intertrial forgetting. July 24, 1963.

59 R. C. Calfee. Long-term behavior of rats under probabilistic reinforcement schedules. October 1, 1963.

60 R. C. Atkinson and E. J. Crothers. Tests of acquisition and retention, axioms for paired-associate learning. October 25, 1963. (A comparison of paired-associate learning models having different acquisition and retention axioms, J. math. Psychol., 1964, 1, 285-315)

61 W. J. McGill and J. Gibbon. The general-gamma distribution and reaction times. November 20, 1963. (J. math. Psychol., 1965, 2, 1-18)

62 M. F. Norman. Incremental learning on random trials. December 9, 1963. (J. math. Psychol., 1964, 1, 336-351)

63 P. Suppes. The development of mathematical concepts in children. February 25, 1964. (On the behavioral foundations of mathematical concepts. Monographs of the Society for Research in Child Development, 1965, 30, 60-96)

64 P. Suppes. Mathematical concept formation in children. April 10, 1964. (Amer. Psychologist, 1966, 21, 139-150)

65 R. C. Calfee, R. C. Atkinson, and T. Shelton, Jr. Mathematical models for verbal learning. August 21, 1964. (In N. Wiener and J. P. Schoda (Eds.), Cybernetics of the Nervous System: Progress in Brain Research. Amsterdam, The Netherlands: Elsevier Publishing Co., 1965. Pp. 333-349)

66 L. Keller, M. Cole, C. J. Burke, and W. K. Estes. Paired associate learning with differential rewards. August 20, 1964. (Reward and information values of trial outcomes in paired associate learning. (Psychol. Monogr., 1965, 79, 1-21)

67 M. F. Norman. A probabilistic model for free-responding. December 14, 1964.

68 W. K. Estes and H. A. Taylor. Visual detection in relation to display size and redundancy of critical elements. January 25, 1965, Revised 7-1-65. (Perception and Psychophysics, 1966, 1, 9-16)

69 P. Suppes and J. Donio. Foundations of stimulus-sampling theory for continuous-time processes. February 9, 1965. (J. math. Psychol., 1967, 4, 202-225)

70 R. C. Atkinson and R. A. Kinchla. A learning model for forced-choice detection experiments. February 10, 1965. (Br. J. math stat. Psychol., 1965, 18, 184-206)

71 E. J. Crothers. Presentation orders for items from different categories. March 10, 1965.

72 P. Suppes, G. Groen, and M. Schlag-Rey. Some models for response latency in paired-associates learning. May 5, 1965. (J. math. Psychol., 1966, 3, 99-128)

73 M. V. Levine. The generalization function in the probability learning experiment. June 3, 1965.

74 O. Hansen and T. S. Rodgers. An exploration of psycholinguistic units in initial reading. July 6, 1965.

75 B. C. Arnold. A correlated urn-scheme for a continuum of responses. July 20, 1965.

76 C. Izawa and W. K. Estes. Reinforcement-test sequences in paired-associate learning. August 1, 1965. (Psychol. Reports, 1966, 18, 879-919)

77 S. L. Biehart. Pattern discrimination learning with Rhesus monkeys. September 1, 1965. (Psychol. Reports, 1966, 19, 311-324)

78 J. L. Phillips and R. C. Atkinson. The effects of display size on short-term memory. August 31, 1965.

79 R. C. Atkinson and R. M. Shiffrin. Mathematical models for memory and learning. September 20, 1965.

80 P. Suppes. The psychological foundations of mathematics. October 25, 1965. (Colloques Internationaux du Centre National de la Recherche Scientifique. Editions du Centre National de la Recherche Scientifique. Paris: 1967. Pp. 213-242)

81 P. Suppes. Computer-assisted instruction in the schools: potentialities, problems, prospects. October 29, 1965.

82 R. A. Kinchla, J. Townsend, J. Yellott, Jr., and R. C. Atkinson. Influence of correlated visual cues on auditory signal detection. November 2, 1965. (Perception and Psychophysics, 1966, 1, 67-73)

83 P. Suppes, M. Jerman, and G. Groen. Arithmetic drills and review on a computer-based teletype. November 5, 1965. (Arithmetic Teacher, April 1966, 303-309.

84 P. Suppes and L. Hyman. Concept learning with non-verbal geometrical stimuli. November 15, 1968.

85 P. Holland. A variation on the minimum chi-square test. (J. math. Psychol., 1967, 3, 377-413).

86 P. Suppes. Accelerated program in elementary-school mathematics -- the second year. November 22, 1965. (Psychology in the Schools, 1966, 3, 294-307)

87 P. Lorenzen and F. Binford. Logic as a dialogical game. November 29, 1965.

88 L. Keller, W. J. Thomson, J. R. Tweedy, and R. C. Atkinson. The effects of reinforcement interval on the acquisition of paired-associate responses. December 10, 1965. (J. exp. Psychol., 1967, 73, 268-277)

89 J. I. Yellott, Jr. Some effects on noncontingent success in human probability learning. December 15, 1965.

90 P. Suppes and G. Groen. Some counting models for first-grade performance data on simple addition facts. January 14, 1966. (In J. M. Scandura (Ed.), Research in Mathematics Education. Washington, D. C.: NCTM, 1967. Pp. 35-43.

91 P. Suppes. Information processing and choice behavior. January 31, 1966.

92 G. Groen and R. C. Atkinson. Models for optimizing the learning process. February 11, 1966. (Psychol. Bulletin, 1966, 66, 309-320)

93 R. C. Atkinson and O. Hansen. Computer-assisted instruction in initial reading: Stanford project. March 17, 1966. (Reading Research Quarterly, 1966, 2, 5-25)

94 P. Suppes. Probabilistic inference and the concept of total evidence. March 23, 1966. (In J. Hintikka and P. Suppes (Eds.), Aspects of Inductive Logic. Amsterdam: North-Holland Publishing Co., 1966. Pp. 49-65.

95 P. Suppes. The axiomatic method in high-school mathematics. April 12, 1966. (The Role of Axiomatics and Problem Solving in Mathematics. The Conference Board of the Mathematical Sciences, Washington, D. C. Ginn and Co., 1966. Pp. 69-76.

(Continued on inside back cover)

INSTRUCT CODERS' MANUAL

by

Jamesine Friend

TECHNICAL REPORT 172
May 1, 1971

PSYCHOLOGY SERIES

This research has been supported by National Aeronautics and Space Administration Grant NGR-05-020-244 and U. S. Office of Education Grant OEG-0-70-4797(607)

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

STANFORD UNIVERSITY

STANFORD, CALIFORNIA

TABLE OF CONTENTS

I. INTRODUCTION

This manual is designed both as an instructional manual for beginning coders and as a reference manual of the coding language, INSTRUCT.

Part I (pp. 1 to 3) is a brief introduction; a better introduction would be to sign on as a student for one of the programming courses (Introduction to Programming: AID, or Introduction to Programming: BASIC) and to take a few lessons.

Part II is devoted to a description of the coding language. The first few sections (Sections IIA to IIC, pp. 4 to 26) give an overall picture of the coding language and should be read quite thoroughly, perhaps even read twice. The remainder of Part II (Sections IID to IIK, pp. 27 to 60) is a reference manual of the coding language; on first reading, this section should be skimmed rather than read thoroughly.

Part III discusses the major programs necessary to implement the teaching system and also lists the limitations of the current implementation on such things as number of lessons, number of exercises per lesson, and length of problems. The beginner need not concern himself too much with the details in Part III, since they are not likely to have immediate effect on his coding efforts. Therefore, on first reading, skim Part III and return to it later when questions arise.

Part IV, of utmost importance to the new coder, describes in detail exactly how to code a lesson, what buttons to push, and what utility programs to use.

Part V, the operator's instructions, must be thoroughly understood by everyone (machine operators, coders, writers, etc.) working on the project.

Part VI, the section on advanced coding techniques, is best read after considerable coding experience.

The coding language INSTRUCT is a high-level programming language designed for programming computer-assisted instruction. Instructions that must be given to the computer about how to present a lesson to students include:

1. what exercises to type and when to wait for an answer;
2. how to analyze the student's answer to determine whether or not it is correct;
3. how to respond to the different possible answers a student may give;
4. what exercise to present next.

These instructions to the computer must be written in a language the computer understands, and INSTRUCT is one such programming language. After the instructions are written, they must be entered into computer memory (see Part IV) where they will be stored until some student needs them. The student takes the lesson by using a computer program called INST which interprets the language INSTRUCT into machine language so that the computer can operate. The lessons are grouped into "courses" in computer storage.

Each course consists of a number of strands and contains a number of lessons. Each lesson consists of a series of exercises, written in the language INSTRUCT. In addition to interpreting the coded lessons, the program INST also keeps track of which students are enrolled for which courses, how far each student has progressed, and how well he is doing.

A number of optional features included in the INST program allow the student to request the correct answer to an exercise or allow him to do exercises in any order he wishes. Which optional features are used depends upon the "course type," and each course is identified as one of six possible types. Some course types give the student control over his own sequence of exercises, others allow him to request additional instruction before he responds to an exercise. Some permit an unlimited number of trials on each exercise, while still others restrict the number of allowable trials, etc. In addition to the content of certain "standard messages," the course type determines what characters are designated as "student control keys." For example, one student control key is the "tell" key; the student types the "tell" key to get an answer. In other instances, the student types the CTRL key and the letter G simultaneously or he types a slash (/) to get the answer. The end of a response may also be indicated by typing an "enter" character; in one course type the "enter" character may be the RETURN key; in another it may be a space or a period.

A complete list of student control characters, standard messages, etc., associated with each course type is given in Part VII.

II. THE CODING LANGUAGE

A. Format of Commands

A lesson is coded by using a series of commands or instructions that cause the computer to present problems, via teletype, to students. For example, commands cause a problem statement to be typed; commands cause student answers to be analyzed (checked for correctness); commands cause specified messages to be typed if the student is correct, etc.

Each command must begin with an "op code" followed by a space. Op codes, the vocabulary of the coding language, are mnemonic words, such as EXER, HINT, and EQ, and serve to specify the kind of command. EXER, for example, is the op code for a command that causes an exercise to be displayed. Of course, the text for the exercise must also be supplied by the coder, so an EXER op code is followed by a text string containing the text of the problem. The text must be enclosed in text delimiters, such as + or /. These text delimiters serve as "quotation marks"--in fact, quotation marks may be used as text delimiters.

EXER + WHAT IS THE SUM OF 15 AND 12? +

The above command causes the computer to type the following on the student's teletype:

WHAT IS THE SUM OF 15 AND 12?

and then to wait for a response from the student.

As another example, an EQ command is used to find out if the student's answer is equal to a certain number. The command

EQ /3.1416/

causes the teaching program to check the student response, which must be equal to 3.1416 in order to be correct.

The correct answer command

CA "GOOD THINKING."

causes the teaching program to type

GOOD THINKING.

on the student's teletype if he makes a correct answer.

All commands must follow these rules:

1. Each command must begin on a new line, although the command is not limited to a single line.

   Note: Blank lines are allowed between commands and should be used wherever necessary to improve the readability of the coding.

2. The first word in the command is an op code (op codes are listed in Section IIK, p. 57).

3. The op code must be followed by a space to separate it from the rest of the command.

4. The first text delimiter, if there is any text, must be on the first line of the command, although the text itself may start on any line.

All commands in the coding language have the same form. The first word of each command is an op code that defines the kind of command. Following the op code are "arguments" that serve to modify the op code by adding further specifications. The op code is separated from the following arguments by a space. Most op codes require only one argument, usually a text string. For example, in the command,

CA /VERY GOOD/

the text string /VERY GOOD/ is the argument for the CA op code. For many op codes (CA, WA, and other action op codes) the argument is

optional. For some op codes (YES, NO, TRUE, FALSE) no arguments are needed. Some op codes, such as BRCA, require more than one argument; the first argument for a BRCA is a strand identifier, the second a lesson number, the third a problem number, and the fourth an optional text string. The exact form and number of arguments for each op code are given in the following pages (Sections IID to IIG). If an op code requires more than one argument, the arguments must be separated by commas.

Text strings must always be contained between text delimiters. Text delimiters may be any characters not contained in the text string. For example, one may use quotation marks around a text string, if there are no quotation marks in the text itself:

HINT "TO FIND INTEREST, USE THE FORMULA I = P × R × T"

Usually special characters like + or % or / are easier to read than letters or numbers or standard punctuation marks. A few special characters like < > (: , ← cannot be used as text delimiters, for various obscure reasons. Letters within text strings may be typed in either upper or lower case, since the lesson processor translates all letters to upper case.

Text strings are handled by the computer in one of two ways, depending upon the kind of op code that has the text string as an argument. Some text strings specify what is typed <u>for</u> the student (if the op code is EXER, HINT, CA, etc.) and others specify what should be typed <u>by</u> the student (if the op code is EXACT, EQ, etc.).

Text strings used as arguments for the problem statement commands and the action commands specify the text typed on the teletype by the computer. In such text strings, the coder should pay close attention

to spacing and carriage returns, since the text typed on the teletype reflects the exact text put into the text string by the coder. In particular, if you want the type bar on the teletype positioned at the beginning of a new line before the message is typed, the first character in the text string should be a carriage return, i.e., start your text on the line after the op code; otherwise, the teletype will start typing from whatever position it was left, which may result in the message running over the end of a line and being unreadable. Also, if you want an empty line or two after the message, end the text string with a few carriage returns.

There is one exception to the rule about beginning a text string with a carriage return. Before an NEXER text string is sent to a teletype, three carriage returns and the problem number are displayed automatically on the teletype. Hence, the coder can assume that the NEXER text is sufficiently set off from the preceding text. (For more about problem numbering, see Section IIH, p. 54.)

A few examples of recommended ways to code text strings for teletype displays follow:

```
NEXER  /NAME A STATE EAST OF THE MISSISSIPPI AND WEST OF THE
ALLEGHENIES.
/

HINT /
HINT: WHICH STATE IS CHICAGO IN?
/

EXACT  /ILLINOIS/
CA /
GOOD/
```

WA /
WRONG. TRY AGAIN

/

After you have coded one lesson and tried it on a teletype, you will be in a better position to decide where to put spaces and carriage returns.

An additional cautionary word about the characters used in text strings. Although any character, except the character you are using as a text delimiter, may be used in a text string, quite a few characters on the Philco or IMLAC keyboards have no equivalents on the teletypes. If an untranslatable character is used in a text string, it is translated into a question mark (?) and may cause your text to look peculiar.

Also, a comment about upper-case and lower-case letters. Since there are no lower-case letters on Model-33 or 35 teletypes, both lower-case and upper-case letters are translated to upper-case letters. In other words, you may type your text in either capitals or small letters as you prefer. In fact, capital and lower-case letters are equivalent everywhere in the coding, so op code names, etc., may also be typed in lower case if desired. (Upper case is used for op codes throughout this manual only for the purpose of making examples more readable.) The form

exer /what is the sum of 1 and 2?/

is quite acceptable.

Not all text strings used in the coding specify text to be typed on a teletype by the computer. Text strings may also be used to specify what should be typed by the student. The text strings used as arguments for the analysis op codes (EXACT, MC, EQ, etc.) are all of this kind. In such text strings, do not use unnecessary spaces or carriage returns,

for though they are not prohibited, the computer must sort out and discard extra spaces and carriage returns before it checks the student answer.

Here are some examples of recommended ways to code text strings for analysis commands:

```
EXACT  /ELEPHANT/
MC  /AC/
EQ  /3.1416/
KW  /RICHARD NIXON/
```

Note that in the last example, the space between the two words "Richard" and "Nixon" is a necessary part of the text string and should not be omitted.

Caution: Text strings for MC, NOTMC, EQ, NOTEQ, KW, NOTKW, EXACT, and NOTEXACT must be on the same line as the op code; these commands must not use more than one line.

B. Classification and Order of Commands

The commands for problem coding are of four kinds:

1. Problem Statement Commands

The problem statement commands cause either a display of the problem statement or a display of additional instruction or information about the problem and must be the first command in a problem. The op codes for problem statement commands are EXER, LEXER, NEXER and SEXER.

EXER causes the coded text to be typed and then causes the computer to wait for a student response. For example, the command

```
EXER +
HAVE YOU EVER STUDIED PROGRAMMING BEFORE? +
```

causes the computer to type

```
HAVE YOU EVER STUDIED PROGRAMMING BEFORE?
```

and then to wait for a student response.

LEXER, NEXER and SEXER are all variants of the EXER command; they all cause a display of text and a pause for student response. The differences between these commands are described in Section IID, p. 27.

2. Analysis Commands

Analysis commands cause a student's response to be analyzed to determine whether it is correct. The analysis op codes are

EXACT	NOTEXACT	
MC	NOTMC	(MC means "multiple choice")
EQ	NOTEQ	(EQ means "equal number")
KW	NOTKW	(KW means "key word")
YES	NO	
TRUE	FALSE	

Some examples of analysis commands are:

(1) EXACT /COMPUTER/

(The student's response is marked correct if he types the word "computer" and marked wrong otherwise. The student response must match the coded answer exactly, character by character, space by space.)

(2) KW /COMPUTER/

(The student's response must contain the key word "computer.")

(3) TRUE

(The student's response must be either the word "true" or the letter "t.")

(4) MC /B D E/

(MC is used for multiple-choice problems. The student's response must be a list of the letters B, D, and E, in any order.)

(5) NOTEXACT /ELEPHANT/

(The student's response is wrong if it is the word "elephant." Any other response, such as "happy," is marked correct. This analysis command would be used in a problem such as "TYPE ANYTHING EXCEPT 'ELEPHANT'."! Less trivial uses of the NOT commands will be discussed later.)

The analysis commands work something like this: A counter in the computer called SCORE is capable of storing either positive or negative numbers. As soon as the student's response is checked for correctness, a number is put into the counter SCORE; if the student's response is correct, a positive number (usually +1) is put into SCORE, and if the response is incorrect a negative number (usually -1) is put into SCORE. This counter is later used by the computer to decide whether to type a "correct" message or a "wrong" message for the student.

Generally, the analysis commands cause the student's response to be checked and an appropriate number to be put into SCORE (this is all done "behind the scenes," i.e., the student sees no action whatsoever). However, some analysis commands also check the student's response to see if it is in the correct "form" and type an error message if it is not. For example, if the equal command

EQ /5/

is used in the coding, the correct response is the number 5. If the student types "5," SCORE is set to +1. If the student types "4," SCORE is set to -1. However, if the student types "five," the answer is not in the correct form, so an immediate error message is typed for the student:

WRONG, PLEASE TYPE A DECIMAL NUMBER TO ANSWER THIS PROBLEM.

The commands that send error messages if the student makes an error in form are

MC	NOTMC	(The answer must be a letter or list of letters.)
EQ	NOTEQ	(The answer must be in the form of a decimal number or in scientific notation.)

YES NO (The answer must be "YES," "Y," "NO," or "N.")
TRUE FALSE (The answer must be "TRUE," "T," "FALSE," or "F.")

If an analysis command causes an error message to be typed, it also causes another pause for a new response from the student.

The analysis commands come in pairs (EQ and NOTEQ, TRUE and FALSE, EXACT and NOTEXACT, YES and NO). The two related commands cause exactly the same analysis of a student's response, but if the op code has a NOT prefix, the last thing done is to negate the value of SCORE. For example, the command

NOTEQ /7.5/

checks the student response to find out if it is equal to 7.5. (If the student's response is not a number, an error message will be sent.) If the response is 7.5, SCORE is set to -1; otherwise it is set to +1. The NOT op codes are generally used in looking for expected wrong answers, so that if a student makes a specific mistake, he can be given a wrong-answer message related to the kind of mistake he makes.

If a NOT command is used to check for an expected wrong answer, other analysis commands are usually used to check for the correct answer.

The analysis commands are described in more detail in Section IIE, p. 31.

3. Action Commands

Since the analysis commands ordinarily give the student no visible indication of whether his answer is correct, a third type of command is needed. The "action command" tells the student the result of the analysis of his answer.

Some action commands are either for correct answers or wrong answers. These are

CA	WA	(C means "correct," W means "wrong")
C1	W1	
C2	W2	
C3	W3	
BRCA	BRWA	(BRCA means "BRanch if Correct Answer")
	WS	(WS means "Wrong but Skip to next problem")
	WR	(WR means "Wrong. Retype exercise")

These commands cause a specified message to be typed only if SCORE is set to the appropriate value. For example,

CA /GOOD./

causes the message

GOOD.

to be typed only if SCORE is a positive number.

The C1, C2, and C3 commands are similar to the CA command, except that action is taken only if SCORE is a specific positive number. A C2 command causes action only if SCORE is +2, a C1 command causes action if SCORE is +1, etc., whereas a CA command causes action if SCORE is any positive number.

The WA, W1, W2, and W3 commands are similar to CA, C1, C2, and C3, except that the action takes place only if the value of SCORE is an appropriate negative number.

The correct-answer commands, CA, C1, C2, and C3, cause a branch to the next problem in sequence after the correct-answer message is typed, whereas the wrong-answer commands, WA, W1, W2, and W3, cause a branch back to the part of the problem where the student was expected to make a response.

It is sometimes desirable to branch to problems other than the next ones in sequence. To accomplish this, a BRCA or BRWA command must be used. "BRCA" means "BRanch if Correct Answer" and "BRWA" means "BRanch if Wrong Answer." As an example, the command

```
BRCA  L,2,15,/O.K./
```

causes a display of the message

```
O.K.
```

followed by a branch to Strand L, Lesson 2, Problem 15.

The WS command is used if the student is to be given the next problem in sequence, even if he makes an incorrect response. "WS" stands for "Wrong but Skip to next problem."

The text string (message) that follows a C or W op code is optional. For each of these action commands, a "standard message" is used if the coder does not supply a message. The short form of the WA command, namely,

```
WA
```

causes the standard message

```
WRONG
```

to be typed if a student makes an incorrect response.

The short form of the W2 command

```
W2
```

causes the following standard message

```
SOME OF THOSE ARE WRONG.
```

to be typed if a student makes an incorrect response. The short form of an action command may be used whenever the standard message is satisfactory. (The standard messages are listed in Section IIF, p. 40).

Examples of short forms of action commands:

```
CA
WA
BRCA  L,2,12
W1
BRWA  L,15,7
```

Other action commands, besides the C and W kind, that do not depend upon the value of SCORE, are HINT, TELL, BRTELL, and REPET. All of these commands cause the text to be typed at the request of the student. TELL, for example, is used to code whatever text the coder wants typed for a student who asks the computer to "tell the answer." (Whether or not a student is allowed to do this, and exactly what he must type to get the answer, depends upon the course type.) A TELL command might look like this:

```
TELL  /THE CORRECT ANSWER IS 27./
```

A HINT command specifies the text that will be typed if a student asks for a hint, and a REPET command specifies the text that will be typed if a student asks to have the problem repeated; if a student requests a repeat, the original problem statement will also be retyped.

The action commands are described in detail in Section IIF, starting on p. 38.

4. Miscellaneous Commands

Besides the commands used in coding a problem (problem statement commands, analysis commands, action commands), the coding language also contains several miscellaneous commands:

```
LESSON
EOL
TYPE
JMPGE
JMPL
```

DEFINE
COMMENT
CRUNCH
NEXT

The two most important of these are LESSON and EOL. The LESSON command is the first command in a lesson and serves to identify the lesson by strand and by number like this:

LESSON T,5

The EOL (End of Lesson) command is the last command in a lesson.

The use of the DEFINE command is discussed in Section VI, p. 76, since it need not be used by the beginning coder, and a discussion of its use might be more confusing than enlightening. Suffice it to say that a DEFINE command is used to redefine a standard message or to define a macro. After the beginning coder has coded and processed a few lessons and has seen them from the viewpoint of a student, he may begin to feel some dissatisfaction with the standard messages; at that time, he should study Section VI , p. 76, rather thoroughly and then redefine the messages to suit himself.

The TYPE command is used for text display, and the JMP commands are used for branching, which is conditional upon the student's performance in a lesson. The COMMENT command is used to put comments into a lesson; the comments have no effect on how the lesson is presented to the students. The CRUNCH command removes all spaces from the student's response. NEXT sets the student's restart point at the next problem.

This brief discussion of miscellaneous commands has been inserted here only to round out the picture of the types of commands. In summary, the coding language contains four different types of commands:

Problem Statement Commands
Analysis Commands
Action Commands
Miscellaneous Commands

5. Ordering of Commands

There are only a few rules about the ordering of commands:

(1) Every lesson starts with a LESSON command.

(2) The second command in a lesson must be a TYPE command, an EXER command, an NEXER command, an LEXER command, or a COMMENT.

(3) The last command in a lesson must be an EOL command.

A "problem" is defined as any string of commands starting with an EXER, an LEXER, an NEXER, or an SEXER command; the end of a problem is signalled by one of these commands:

EXER, LEXER, NEXER, SEXER
TYPE, JMPGE, JMPL, EOL

Commands within a problem must obey these rules:

(1) Every problem must begin with a problem statement command (EXER, LEXER, NEXER, SEXER). Only one problem statement command is used in a problem.

(2) If there are several HINT commands, they must be coded in a group, with the first hint first, then the second, etc. (If other commands are used between two HINT commands, all of the hints except those in the first group will be lost.)

(3) The analysis and action commands may be given in any order, depending on the desired sequence of events.

Note: The only mandatory command in a problem is an EXER (or LEXER or NEXER or SEXER) command. Thus a problem could consist of a single command. For example, here is an entire lesson (containing just one problem):

```
LESSON  L,1
EXER  /TYPE ANYTHING TO START LESSON 2./
EOL
```

Since no analysis of the student response is coded, there will be no analysis, responses will not be considered either correct or wrong, and anything the student types will allow him to go on to the next exercise, which is the first problem in Lesson L2.

C. Examples of Coding

Example 1. A simple problem using an "EXACT" analysis.

```
EXER  /APOLLO AND .... WERE TWINS.

/
TELL  /DIANA/

HINT  /APOLLO'S TWIN WAS A GODDESS.  WHAT WAS HER NAME?/

HINT  /THE NAME OF APOLLO'S SISTER STARTED WITH THE LETTER D.
WHAT WAS HER NAME?/

EXACT  /DIANA/
CA
WA  /WRONG.  TRY AGAIN.
/
```

Notice that the short form of the CA command was used; this is equivalent to

```
CA  /CORRECT
/
```

Example 2. An arithmetic problem.

```
EXER  /5 - 2 =
/
TELL  /3/

HINT  /SUBTRACT 2 FROM 5./

NOTEQ  /7/
WA  /WRONG.  YOU MUST SUBTRACT, NOT ADD.  TRY AGAIN./

EQ  /3/
CA  /VERY GOOD/
WA
```

Notice that in example (2) a check for the expected wrong answer (7) was made before the check for the correct answer. It is generally easier to put the check for an expected wrong answer first. Notice also that there is no CA command between the NOTEQ and the EQ commands. If there were, and if the student typed the wrong answer 4, the NOTEQ command

would cause SCORE to be set to +1, since it is true that 4 is not equal to 7. Then the misplaced CA command would take action (because SCORE would be positive), causing the student to get a correct-answer message even though he typed the wrong answer.

Example 3. A problem with several correct answers.

```
EXER  /WHAT IS THE OPPOSITE OF "ABOVE?"
/
TELL  /CORRECT ANSWERS: BELOW, BENEATH, UNDER/

EXACT  /BELOW/
CA  /GOOD/

EXACT  /BENEATH/
CA  /VERY GOOD/

EXACT  /UNDER/
CA  /RIGHT/
WA  /WRONG.  TRY AGAIN./
```

Notice that there is a CA command after each EXACT command. Since a CA action includes an immediate branch to the following problem, the student will go on if he has typed any of the correct answers. Notice also that there is only one WA command, which comes after all of the EXACT analyses. If there were a misplaced WA after, say, the first EXACT command, then the student who typed the second correct answer, "beneath," would get a wrong-answer message.

No HINT command is used in the above problem; if a student requests a hint, he is given the standard hint message, "NO HINT WAS WRITTEN."

Example 4. An exercise using a YES command and a BRCA command.

The following problem from Lesson 2 of Strand L illustrates the use of the YES command and the BRCA command.

```
EXER  /THE REST OF THIS LESSON IS REVIEW.  DO YOU WANT TO DO
THE REVIEW?
/
YES
CA  /HERE ARE THE REVIEW PROBLEMS./
NO
BRCA  L,3,1,/O.K./
```

In the above problem "yes" is considered a "correct answer" and causes the student to branch to the next problem in sequence, namely, the beginning of the review. An answer of "no" causes a branch to the beginning of the next lesson, i.e., Lesson 3, Problem 1 in Strand L, skipping the review section.

Example 5. An exercise that allows the student response to be typed on the same line as the exercise.

```
EXER  /
2 + 3 =/

TELL  /
5/

EQ  /5/
CA
WA
```

Notice that the text string for EXER has no carriage return at the end. This means that the student response for the problem is displayed right after the symbol "=." Notice also that the text string for TELL contains no carriage return at the end. No argument is used with either CA or WA; the standard messages are used. There is no HINT command so the standard message, "NO HINT WAS WRITTEN," is used.

Example 6. A typical example of YES and NO coding.

```
SEXER /
DO YOU REMEMBER HOW TO GET A HINT?
/

HINT /
YES, I SEE YOU DO REMEMBER. SO TYPE "YES" TO ANSWER THE QUESTION.
/

YES
CA /GOOD/

NO
CA /
TO GET A HINT, HOLD DOWN THE CTRL KEY WHILE YOU TYPE "H."/
```

Notice that the TELL command is not used. Also, there is no WA command since all possible student responses are already accounted for.

If the student response is "yes," the YES routine sets SCORE to 1 and the first CA causes a display of "GOOD" and a branch to the next problem.

If the student response is "no," the YES routine sets SCORE to -1, the first CA is not executed because SCORE is not positive, and control passes to the next command, the NO command. The NO routine sets SCORE to 1, so the following CA is executed and there is a branch to the next problem.

If the student response is neither "yes" nor "no," the YES routine sends an error message and awaits another response.

The reason for using a NO command followed by a second CA command is to allow the student to go on to the next problem, regardless of whether he answers "yes" or "no." However, the student receives different messages for the different expected responses.

Example 7. An exercise that checks for an expected wrong answer and responds with a specific wrong-answer message.

```
NEXER /
COMPLETE THIS AID COMMAND TO ASSIGN THE VALUE OF PI TO THE
VARIABLE P.

      .... = 3.1416.
/

TELL /
SET P/

HINT /HINT: HOW DO YOU DEFINE A CONSTANT IN AID?
/

NOTKW /LET/
WA /
YES, THE "LET" COMMAND WILL WORK BUT THERE IS A MORE EFFICIENT
WAY. TRY AGAIN.
/

EXACT /SET P/
CA
WA
```

The text strings for NEXER, HINT, and WA all include a final carriage return in order to position the teletype at the left of a new line.

Note that a check for the expected wrong answer "LET" was made before any check for the correct answer.

Example 8. A problem that uses an EQ command with a specified tolerance.

```
EXER /
SUPPOSE YOU WANT TO INSERT A NEW LINE BETWEEN LINES 17.65 AND
17.9 IN AN AID PROGRAM. WHAT LINE NUMBER WOULD YOU USE?
/

TELL /
17.7 WOULD BE OK. OR 17.8. OR 17.77, ETC./

HINT /
PICK A NUMBER BETWEEN 17.65 AND 17.9.
/
```

```
EQ  /17.775,0.125/
CA
WA
```

Notice that the EQ command has two arguments, 17.775 and 0.125; this is interpreted as 17.775 $\pm$ 0.125, which includes all the numbers between 17.65 and 17.9. 17.775 is found by taking the average of 17.65 and 17.9.

Example 9. An exercise with a subsequent subexercise that is not given to all students.

Here is a series of two problems using a KW analysis and a BRCA. (The first problem is assumed to be L3-14 followed by a subexercise that has no number.)

```
EXER  /
WHO IS THE PRESIDENT OF THE UNITED STATES?
/

TELL  /RICHARD NIXON/

HINT  /
HIS NICKNAME IS "DICK."
/

KW  /NIXON/
WA  /WRONG.  TRY AGAIN.
/

KW  /RICHARD/
BRCA L,3,15,/CORRECT/

KW  /DICK/
BRCA L,3,15,/CORRECT/

SEXER  /HIS LAST NAME IS NIXON.  WHAT IS HIS FIRST NAME?/

TELL  /RICHARD/

HINT  /WHAT IS PRESIDENT NIXON'S FIRST NAME?
/
```

```
KW  /RICHARD/
CA

KW  /DICK/
CA
WA
```

The first KW command in the problem causes a check for the key word "Nixon." Next there is a check for either the word "Richard" or the word "Dick." In either case, there is a branch to the next numbered problem (caused by the BRCA commands). The effect of the BRCA commands is to allow the student who types the entire answer correctly to bypass the following subproblem which asks for the President's first name.

Example 10. A multiple-choice problem using the short form of W1, W2, and W3 action commands.

```
EXER  /
WHICH NUMBERS ARE GREATER THAN 7?
     A.  7 - 15
     B.  15 - 7
     C.  7 - 9
     D.  7 - (-2)
/

MC /B  D/
CA
W1
W2
W3
```

The standard messages are used for the CA, W1, W2, and W3 action commands. The standard message for W1 is "WRONG," the message for W2 is "SOME OF THOSE ARE WRONG," and the standard message for W3 is "YOU HAVEN'T FOUND ALL OF THEM."

D. Problem Statement Commands

The problem statement commands are:

EXER	(exercises)
LEXER	(long exercise)
NEXER	(numbered exercise)
SEXER	(subexercise)

The only command required in a problem is one of the problem statement commands. The four problem statement commands all have the same form

(op code)(space)(text string).

Problems are numbered automatically for the coder. Each EXER, LEXER, NEXER and TYPE command is assigned an internal problem number; the first one in a lesson becomes Problem 1, the second becomes Problem 2, etc. A problem that begins with an SEXER command does not receive a problem number. The implications of this are these: first, because an SEXER (subexercise) has no number, there is no way for a student to specify the problem number when he is at the choice point, i.e., when the computer types the "WHERE TO?" message, the student cannot request a subproblem by number. Second, for an SEXER, a number cannot be stored in the student's restart record (the record of his current location in each of the strands), and consequently, if a student signs off during a subexercise, his restart point on that strand will be the preceding numbered problem; the next time the student signs on and asks to continue his lessons, he will backtrack to the last numbered problem rather than start at the subexercise he was last working on.

With a judicious use of SEXER's the student may be asked to make multiple responses, such as constructing tables, using more than one

line for his response. The entire set (one EXER and any number of SEXER's) appears as a single problem to the student.

The difference between EXER and NEXER is that although they are both numbered problems the problem number is displayed automatically if an NEXER (numbered exercise) op code is used, but it is not displayed if an EXER op code is used.

1. The EXER Command

The form of the EXER command is

EXER /the problem statement is given here./

Several lines may be used for this statement.

This command causes these actions:

(1) The text is typed on the teletype.

(2) The "ready" character is typed. For most course types the ready character is an asterisk. This signals the student that the computer is ready for him to type a response.

(3) The computer waits for the student response. No further action is taken until the student finishes his response and indicates that he is finished by typing the "enter" character. For most course types the enter character is the RETURN key.

(4) The student's response is edited by removing all spaces at the beginning and end of the response and deleting all invisible characters in the response. At this time there is a check for control characters, such as hint or erase requests, and the appropriate action is taken.

2. The NEXER Command

The form of the NEXER command is just like the form of the EXER command:

```
NEXER /the exercise is written here, using as many lines
as needed./
```

Any problem that begins with an EXER, an LEXER or an NEXER command is automatically numbered by the system (see Section IID, pp. 27 - 30). When an NEXER command is encountered, the first action is a display of the problem number.

First, there are 3 carriage returns (CR's), resulting in 3 blank lines.

Second, the strand identification letter is typed.

Third, the lesson number is typed.

Fourth, a dash is typed.

Fifth, the problem number is typed.

Sixth, a colon and one space are typed.

Seventh, the text string following the NEXER op code is typed exactly as specified.

For example, if the third problem in Lesson 12 of Strand T starts with this command

```
NEXER / WHAT IS THE VALUE OF X SQUARED PLUS Y?
/
```

the display appears as

```
T12-3: WHAT IS THE VALUE OF X SQUARED PLUS Y?
```

If the text for the NEXER command begins on the second line, like this,

```
NEXER /

WHAT IS THE VALUE OF X SQUARED PLUS Y?
/
```

the display is

T12-3:

WHAT IS THE VALUE OF X SQUARED PLUS Y?

3. The LEXER Command

The LEXER (long exercise) command is used for exercises with an exceptional amount of text. The form is the same as for EXER and NEXER. This command is rarely used; see Section IIIC, p. 65, for further explanation.

4. The SEXER Command

The SEXER (subexercise) command is identical to the EXER command. The form is

SEXER /Problem statement is put here, using several lines if necessary./

There is no automatic problem number or blank lines before the text of an SEXER is displayed. If blank lines are desired, they must be put in the text string by the coder, like this

SEXER /

To ensure several blank lines between this problem and the last problem, put blank lines at the beginning of this text string./

As mentioned before, an SEXER is not numbered internally (and, of course, no number is typed for the students). This means there is no way to get to an SEXER other than by an automatic branch from the preceding exercises; there can be no branch command directly to an SEXER, nor can the student request an SEXER as he can other kinds of exercises.

Since an SEXER can be reached only by going through the previous exercise, it cannot be the first exercise in a lesson. It must always follow an EXER, an LEXER, an NEXER, a TYPE, or another SEXER. Any number of SEXERS may be used in a string.

E. Analysis Commands

In a problem the analysis commands must come somewhere after the problem statement command. Analysis and action commands may be combined in any order within the problem.

None of the analysis or action commands are used until after the student completes his response. The duties of the analysis commands are as follows.

First, the student response is analyzed for correctness. If the response is correct, as specified by that analysis command, a positive number is put into the counter SCORE; if the student response is wrong, a negative number is put into SCORE. Note: the number is not added to the existing number in SCORE; rather, the existing value of SCORE is replaced by the new value.

Second, if the analysis routine includes a check on the form of the answer, and if the student's response is in the wrong form, an error message is sent and there is a branch back to the part of the problem that pauses for a student response.

Third, if no error message is sent, control passes to the next command in sequence.

The analysis commands come in pairs (EXACT and NOTEXACT). In general, both related commands do the same analysis, i.e., send the same "error-in-form" messages. However, the command with the NOT-prefix sets the value of SCORE to the negative of the value which is set by the unprefixed command; if EXACT causes SCORE to be set to 1, then NOTEXACT puts -1 in SCORE, and vice versa.

1. EXACT and NOTEXACT Commands

EXACT and NOTEXACT are analysis op codes that require text strings as arguments. These commands, like the other analysis commands, are not executed until after the student completes his response.

EXACT (NOTEXACT) determines whether the student response matches the coded text string and sets the counter SCORE to +1 (-1) if there is a match, and -1 (+1) otherwise. Leading and following spaces are ignored in the student response.

The form of the commands is:

EXACT /Correct answer is coded here./

NOTEXACT /Put expected wrong answer here./

The text string in an EXACT command may not contain the "enter" character; otherwise, there are no restrictions. The EXACT and NOTEXACT commands do not cause any analysis of the form of a student response. No "error-in-form" messages are sent.

2. MC and NOTMC Commands

MC is an analysis op code ordinarily used for multiple-choice problems. The argument for MC or NOTMC is a text string containing one or more letters, which may be separated by commas or spaces.

MC compares the set of student responses to the set of coded letters. The counter SCORE is set to 1 if the student response is completely correct, -1 if completely wrong, -2 if partially wrong, -3 if partially correct. NOTMC, of course, causes these values to be negated.

The form of the MC and NOTMC commands is

MC /List of correct choices coded here./

NOTMC /Incorrect choice coded here./

The student, if he wishes, may use spaces or commas to separate the letters in his response. The letters in the student response may be given in any order.

The MC and NOTMC commands send an error message if the student types anything other than letters, spaces, or commas. There are five possible results of an MC analysis:

(1) There is an error in form. An error message is sent. (The exact content of the error message depends on the course type. See Part VII.)

(2) The student response is completely correct. SCORE is set to +1.

(3) The student response is completely wrong, i.e., not one of the letters he typed is correct. SCORE is set to -1.

(4) The student response is partially wrong, i.e., he typed some and possibly all the correct answers, but also some incorrect answers. SCORE is set to -2.

(5) The student response is partially correct, i.e., he typed some, but not all, of the correct answers. SCORE is set to -3.

To inform the student whether he was partially correct, completely wrong, etc., an MC command must be followed by W1, W2, and W3 commands, as well as a CA command.

Generally, NOTMC is used to look for expected wrong answers to multiple-choice questions and has only one letter in the text string, e.g.,

```
NOTMC /D/
WA  /NO, D IS NOT CORRECT BECAUSE .../
```

If you use more than one letter in the text string for a NOTMC command, be sure you know the meaning of the various possible values for SCORE: -1, +1, +2, +3. (This is left as an exercise in logic for the adventurous coder.)

Caution: For MC and NOTMC, the entire command must be on one line, unlike most other commands, which may use any number of lines. (This is not much of a restriction, since all 26 letters of the alphabet may be typed on one line!)

3. EQ and NOTEQ Commands

The argument for the EQ and NOTEQ op codes is a text string that may contain either one or two decimal numbers.

```
EQ  /7.56/
EQ  /7.56..04/
NOTEQ  /10,5/
```

If the text string for an EQ command contains only one number, then the correct response must be a number exactly equal to the number in the coded text string.

If two numbers are coded in the text string, the second number is used as the tolerance, i.e., the allowable difference between the student response and the first coded number.

The command

```
EQ  /10,2/
```

defines the correct answer as a number different from 10 by no more than 2, i.e., any number between 8 and 12, inclusive.

If the second number is omitted in the coding, it is assumed to be 0, i.e., there is no tolerance allowed; thus, these two commands are equivalent,

```
EQ  /57.5,0/
EQ  /57.5/
```

If two numbers are in the text string, they must be separated by a comma.

As an example of NOTEQ, the command

```
NOTEQ  /100,2/
```

means that any number not between 98 and 102 is a correct answer.

The EQ and NOTEQ commands cause a check on the form of the student response. If the response is not an acceptably formed number, an error message is sent.

In the argument for an EQ command, any of the usual ways of writing a decimal number are acceptable (no fractions, however). All of the following are equivalent decimal numbers:

```
 .5
0.5
 .50
+.5
+.500.
```

Negative numbers, of course, are indicated by a preceding minus sign:

```
-5
-5.0
-5.
```

Scientific notation may also be used for numbers:

2.3*10↑5 (meaning 2.3 times 10 to the power 5).

Thus, all of the following are equivalent:

```
-3.156*10↑4
-.3156*10↑5
-31560
-31560.0.
```

One restriction on decimal numbers is that they must be limited to nine significant digits. Thus

```
.00000000000123
```

is an acceptable decimal number, since it contains only three significant digits, but

1234.567891

is unacceptable.

Caution. For EQ and NOTEQ, the entire command must be on one line. The same restriction applies to MC, NOTMC, KW, NOTKW, EXACT, and NOTEXACT.

4. KW and NOTKW Commands

KW and NOTKW are analysis op codes requiring text strings as arguments. The text string may contain carriage returns.

The KW routine determines whether the student response contains the coded character string; if it does, SCORE is set to 1, if it does not, SCORE is set to -1. NOTKW negates the value of SCORE.

The form of the commands is

KW /keyword or phrase/

NOTKW /undesired word or phrase/

These commands cause no analysis of the form of a student response. No error messages are sent.

Caution: Spaces may be used as a meaningful part of the text string. The command

KW /under/

classifies responses, such as "UNDERHANDED," "UNDERDOG," and "WUNDERFUL," as correct. If you want to look for the word "UNDER" surrounded by spaces, use spaces in the command

KW / under /

If there is a possibility that the words you are looking for will occur at the end of a sentence, you may want to use an additional analysis for

the key word followed by a period:

```
KW / UNDER./
```

5. YES and NO Commands

There are no arguments for the YES and NO op codes. The forms of the commands are

```
YES
NO
```

The YES command causes SCORE to be +1 if the student response is "y" or "yes" and -1 if the response is "n" or "no." NO does the opposite.

If the student response is in the wrong form, i.e., anything other than "y" or "yes" or "n" or "no," an error message is sent.

6. TRUE and FALSE Commands

The TRUE and FALSE commands are similar to YES and NO, except that TRUE defines the correct answer as "t" or "true." FALSE is used if the correct answer is "f" or "false."

An error message is sent for any other response.

F. Action Commands

The action commands, which may be interspersed with analysis commands, are used to tell the student the result of the analysis of his response and to branch him to other problems if appropriate.

There are two classes of action commands: those which are contingent upon whether a student response is correct or wrong, and those which act when the student types a student control character. The first class, the contingent action commands, are executed, i.e., take action, only if the value of SCORE is appropriate. For example, a C2 command is executed only if the value of SCORE is +2; if the value of SCORE is not +2, the command is simply skipped.

The usual order for analysis and action commands is one analysis command followed by one or two contingent action commands, followed by a second analysis command, etc. Any order that achieves the desired result is acceptable. The coder, however, must be aware that after any action command is executed, there is an immediate branch to the next problem, or back to the pause for student response, or to some other specified problem.

Summary of when contingent action commands are executed:

Op code	Command executed if SCORE =
CA	any positive number
WA	any negative number
C1	+1
W1	-1
C2	+2
W2	-2
C3	+3
W3	-3
BRCA	any positive number
BRWA	any negative number
WS	any negative number
WR	any negative number

All of the above action commands have two forms: a long form (with a text string) and a short form (no coded text string). For example, the long form of a WA might look like this:

```
WA +I'M SORRY BUT YOUR ANSWER IS WRONG.+
```

The short form would be

```
WA
```

with no text string. Whenever the short form is used, a standard message is automatically inserted. The standard message for the WA is WRONG so the short form

```
WA
```

is equivalent to

```
WA +WRONG
+
```

There are two kinds of standard messages: those used in the action command (CA, WA, etc.) and those which are dependent upon the course type. The standard action messages are added by the lesson processor wherever a short form of an action command is used; for example, the command

```
WA
```

is transformed into

```
WA  /WRONG
/
```

by the lesson processor.

Standard Action Messages

Action Command	Content of Message
CA BRCA	"CORRECT"
WA BRWA WS WR	"WRONG"
C1	"CORRECT"
C2	"CORRECT"
C3	"CORRECT"
W1	"WRONG"
W2	"SOME OF THOSE ARE WRONG"
W3	"YOU HAVE NOT FOUND ALL THE CORRECT ANSWERS"

The content of the standard action messages may be changed by the coder by use of the DEFINE command (see Section VI, Advanced Coding Techniques).

Standard messages determined by the course type are not inserted into the lesson coding by the lesson processor and cannot be changed by the coder. A complete list of fixed standard messages is given in Part VII.

Besides the action commands discussed above, a second group of action commands are independent of the value of SCORE. Action commands HINT, TELL, BRTELL, and REPET are used only if the student requests the specific action by typing the appropriate student control character (the "hint" character, the "tell" character, or the "repeat" character). These commands, like the CA and WA type of action commands, cause a text display and a branch, either to another exercise or back to the beginning of the same exercise.

Summary of the kinds of branching done by action commands:

Op code	If executed causes a branch to
CA, C1, C2, C3, WS	next problem in sequence
WA, W1, W2, W3	same problem, pause for student response
WR	same problem, with REPET text, if any, and repeat of problem statement
BRCA BRWA	whatever problem is specified by the coder
HINT	same problem, pause for student response
TELL	next problem in sequence
BRTELL	whatever problem is specified by the coder
REPET	same problem, with REPET text, if any, and repeat of problem statement

1. CA Command

CA is an action op code with one optional argument that is a text string. The forms of the CA command are

```
CA /GOOD/
```

and

```
CA
```

The CA command is executed only if SCORE is positive and causes a display of the message in the text string, followed by a branch to the next problem. If there is no argument, the CA routine displays the standard CA message "CORRECT" before branching to the next problem.

The text string for a CA command may contain carriage returns, i.e., may take several lines.

2. C1, C2, and C3 Commands

C1, C2, and C3 are similar to CA except that C1 is executed only if SCORE = 1, C2 is executed only if SCORE = 2, and C3 is executed only if SCORE = 3.

3. WA, W1, W2, and W3 Commands

The wrong-answer action commands are similar to the correct-answer action commands, except that they cause a branch to the pause for student response after the coded message is displayed. If the course type has specified a small number of permitted trials and the student gets a wrong answer on his last trial, he is branched to the TELL routine instead. (See Section VII, p. 93.)

WA is executed if SCORE < 0.
W1 is executed if SCORE = -1.
W2 is executed if SCORE = -2.
W3 is executed if SCORE = -3.

4. BRCA and BRWA Commands

BRCA has four arguments: the first is a strand identifier, the second a lesson number, the third a problem identifier, and the fourth an optional text string. The arguments must be separated by commas. The BRCA command is executed if SCORE is any positive number.

```
BRCA  L,3,15,/VERY GOOD/
```

causes a branch to Problem 15 of Lesson 3 in Strand L after the message "VERY GOOD" is displayed.

```
BRCA  L,3,15
```

also causes a branch to Problem 15 of Lesson 3 in Strand L; however, the standard CA message "CORRECT" is displayed first.

There are two special forms of the BRCA command. The command

BRCA 0,0,0,/optional message/

causes a branch to the choice point after the message is displayed. The command

BRCA L,0,0,/optional message/

causes a branch to the student's restart point in Strand L.

The form and effects of a BRWA command are exactly like those of a BRCA, except that a BRWA is executed only if SCORE is negative.

5. WS Command

The WS command (Wrong, but Skip to next problem) is executed if SCORE is negative. The message is displayed and there is a branch to the next problem; if no text string is coded, the standard wrong answer message is used. Example:

WS /WRONG. THE CORRECT ANSWER IS 5./

6. The WR Command

The WR command (Wrong: Repeat) is executed if SCORE is negative. The coded message is typed and there is a branch to the REPET routine, which causes the REPET text to be typed and the problem statement to be retyped.

The form of the command is the same as other W commands:

WR /YOUR ANSWER IS WRONG./

If the short form (without text string) is used, the standard wrong-answer message is used.

7. TELL and BRTELL Commands

TELL has one argument that cannot be omitted; the argument is a text string.

```
TELL  /The correct answer is written here, using several
lines if needed./
```

A TELL command causes the following action if a student types the "tell" key:

First, the coded message is displayed.

Second, there is a branch to the next problem in sequence.

The TELL command is optional. If it is omitted, the following actions take place, if the student types the "tell" key:

First, the standard TELL message "NO ANSWER WRITTEN" is displayed.

Second, there is a branch back to the pause for student response.

Notice that a branch to the next problem occurs only if a TELL command is specified in the coding.

The BRTELL command is similar in format and action to the BRCA and BRWA commands.

```
BRTELL  T,5,2,/
THE CORRECT ANSWER IS 23.7./
```

The above command causes the text to be typed; then there is a branch to problem T5-2.

Caution: Only one TELL or BRTELL may be used in a problem.

8. HINT Command

The HINT commands are optional and must follow one another.

HINT has one argument that is not optional; the argument is a text string containing a message that is displayed on the teletype if called

by the student (using the "hint" key).

HINT /Put the first hint here./

HINT /A second hint may be coded after the first hint./

After the text is displayed there is a pause for a student response. The second time a student requests a hint he is given the second hint, etc. In no case is there a branch to the next problem in sequence.

If no HINT command is given, the student who requests a hint gets a standard message "NO HINTS WERE WRITTEN."

If HINT commands are given, but a student requests more hints than are available, he gets a standard message "THERE ARE NO MORE HINTS."

9. The REPET Command

The REPET command, like HINT and TELL, is optional. The form of the command is

REPET /READ CAREFULLY./

If a student requests a repeat of the exercise, the text from the REPET command is displayed, followed by the text from the problem statement. If a student requests a repeat and no REPET command was coded, only the problem statement is typed.

The REPET text is also used if a WR action command is executed. (See WR, Section IIF, p. 43.)

G. Miscellaneous Commands

All commands discussed so far in Part II (problem statement commands, analysis commands, action commands) are used in coding an individual exercise. A lesson, however, also contains commands which are not properly part of any exercise in the lesson. These commands are

LESSON, EOL
TYPE
JMPGE, JMPL
DEFINE
COMMENT
CRUNCH
NEXT

Only the LESSON and EOL commands are required in a lesson; all others are optional.

1. The LESSON and EOL Commands

A course consists of several strands, each of which is divided into lessons; strands are simply a device for organizing the lessons into different categories with a provision for duplicate lesson numbers, e.g., there may be a Lesson 1 in each strand.

Each course may be divided into many strands. The strands are identified by a "strand identifier," a word of one to six letters. For example, a strand may be named "INTRO" or "TEST" or simply "L" or "T."

Lessons within a strand are identified by the strand identifier, followed by a lesson number. For example, if a strand is identified by the letter "L," then the eighth lesson in the strand would be identified as "L8." The lesson numbers must be natural numbers (1,2,3,...,999), but the lessons need not be numbered consecutively.

When a lesson is coded, it must be identified by strand and lesson number. This is accomplished by putting a lesson command at the beginning of the coded lesson. For example, to code Lesson 25 in Strand T, start with this command:

```
LESSON T,25
```

The problem coding (described in the preceding sections) in a lesson starts immediately after the identifier command for the lesson. After all the problems for the lesson are coded, the lesson ends with the "end of lesson" command:

```
EOL
```

2. The TYPE Command

TYPE commands are used for text display only and are similar to the EXER command, except that there is no pause for student response. For example, the command

```
TYPE +
          LESSON 1
    INTRODUCTION TO PROGRAMMING +
```

causes the following to be typed on the student's teletype.

```
          LESSON 1
    INTRODUCTION TO PROGRAMMING
```

There is no pause for a student response.

3. The JMP Commands

The commands JMPGE (Jump if Greater than or Equal to) and JMPL (Jump if Less than) are used to specify branching contingent upon the student's performance in the lesson. For example,

```
JMPGE 75,T,5,1,+
     END OF LESSON
     GOOD WORK+
```

checks the student's performance record. If he has a score of greater than 75 percent on the lesson, the message

```
END OF LESSON
GOOD WORK
```

is typed and the next exercise given is Strand T, Lesson 5, Problem 1.

A JMP command may be used after any exercise in a lesson, not necessarily just at the end. For example, if the following command is used after the fourth exercise in a lesson, it is executed if the student has achieved 60 percent or better for the first four exercises.

```
JMPGE 60,CALC,6,1,+GOOD WORK.+
```

Notice that the JMP commands are similar in format and action to the BR commands. The main difference is that BRCA and BRWA depend only upon one student response whereas JMP depends upon a cumulative record.

Several JMP commands may be given in sequence. Suppose, for example, that a certain lesson is used as a pretest and that students take Lesson P1 if they scored less than 50 percent correct, they take Lesson P2 if they scored between 50 percent and 80 percent correct, and they take Lesson P3 if they scored better than 80 percent correct. The following commands accomplish the desired result.

```
JMPL  50,P,1,1,+
YOU SCORED LESS THAN 50%
HERE ARE SOME PRACTICE PROBLEMS FOR YOU.+

JMPL  80,P,2,1,+
YOU DID QUITE WELL BUT YOU NEED A LITTLE MORE PRACTICE.+
```

```
JMPGE  80,P,3,1,+
 EXCELLENT WORK.+
```

A detailed explanation of how the student's percentage score is calculated is given in Section IIH, p. 51.

4. The DEFINE Command

The DEFINE command is used by the coder to define new op codes. Any combination of commands may be grouped together and given a single name, which may then be used as a new op code. DEFINE is actually an assembly language op code used as a command to the lesson processor. The use of DEFINE is relatively complex and is discussed in detail, with examples, in Section III, Advanced Coding Techniques.

5. The COMMENT Command

The COMMENT command is used to insert comments or notes to yourself. They do not affect the way the lesson is presented to students. For example, you might want to use comments like this:

```
COMMENT /

     LESSON CODED    JAN. 1, 1929
     REVISED         DEC. 31, 1940
     REVISED         JULY 4, 1980 /
```

COMMENT commands may be used anywhere in a lesson including before the LESSON command (a most useful place) or after an EOL.

6. The CRUNCH Command

The CRUNCH command is an editing command and is ordinarily executed before any analysis commands. All it does is request INST to remove spaces from the student response. For example, the following student

responses might all be considered as reasonably correct responses to some exercise.

```
2+3  = 5
2+  3= 5
2  +  3  =5
2   +  3  = 5
```

To facilitate the analysis of such responses, code the exercise like this:

```
EXER / WRITE IN SYMBOLS:
    TWO PLUS THREE EQUALS FIVE.
/
CRUNCH
EXACT /2+3=5/
CA
WA
```

7. The NEXT Command

The NEXT command is used to manipulate a student's restart record. Usually, when a student starts a new day's work, he restarts at the same exercise he was last working on. In some cases, it is best to start the student at the following exercise. To accomplish this, use a NEXT op code in the exercise, like this:

```
EXER / DO YOU WANT TO START A NEW LESSON NOW?
/
YES
CA /OK/
WA /TYPE CTRL-Z TO STOP FOR TODAY/
NEXT
```

The NEXT command may be placed anywhere in the problem coding with the same effect.

H. Lesson Score Counters and X Problem Statement Commands

As mentioned before, the INST program which interprets lessons coded in the INSTRUCT language also keeps a record of how well each student is doing on each lesson. JMP commands are used to compare a student's performance to some specified criterion and to decide what lesson (or exercise) he should take next depending upon whether or not he met the specified criterion. For example, the command

JMPGE 70,T,5,1,/GOOD WORK!/

causes the following actions.

First, LESCOR, the student's percentage score in the lesson, is calculated. Then his score is compared to the specified criterion, in this case 70 percent. If the student's score is greater than or equal to 70 percent, he is branched to Exercise T5-1 after seeing the message

GOOD WORK!

If the student's score is less than 70 percent, no action is taken; he simply continues with the same lesson (of course, there may be a JMPL command or another JMPGE command right after the first one, in which case the student might be branched to elsewhere and not really continue the same lesson).

To understand exactly how the student's percentage score is calculated, one must know the kind of student performance record kept by the INST program. Basically, the percentage score is calculated by dividing the number of exercises correct by the number of exercises done. The complications arise when one asks exactly what constitutes an exercise "done," or an exercise "correct." If the student's first response is a

request for the correct answer, was the exercise "done"? If the student responds incorrectly the first time and then makes a correct response, is the exercise counted as "correct"? Or must the first response be correct? If the student's response is a request for a repeat, and his second response is correct, should not the response be considered correct?

The INST program makes these decisions by considering some student actions to be unresponsive, i.e., not genuine responses; in particular, all uses of control commands are considered to be unresponsive, so a student is not penalized if he asks for a repeat, or a hint.

If the student makes any genuine response to an exercise, then the exercise is counted. If his first response is correct, it is counted as an exercise correct, otherwise the exercise is marked wrong, even if the student eventually gives a correct response.

The lesson score counter is used for only one lesson at a time and may be checked (by using JMP commands) at any time within the lesson. The JMP commands do not disturb the value of the counter, so any number of JMP commands may be used at any desired places. The counter is reset to zero whenever a student changes lessons, or even if he starts the lesson again from the beginning; however, if the student recycles through part of the lesson without going all the way back to the first exercise, the counter is not reset to zero, so the score simply accumulates.

There may be times when you do not care what a student responds to an exercise. For example, if you want to ask the student's opinion with a question like

DID YOU LIKE THIS LESSON?

and you want any answer he makes to be ignored as far as scoring is concerned, use an X in front of the op code, like this:

```
XEXER /
DID YOU LIKE THIS LESSON?
/
```

The X is a signal to the INST program to leave the value of the lesson score counter unchanged. Any of the EXER op codes may be prefixed with an X: XEXER, XNEXER, XLEXER, XSEXER.

Summary of Numbered Op Codes

	Display problem number?	Internally numbered?	Wait for student response?	Changes LESCOR?
TYPE	No	Yes	No	No*
EXER	No	Yes	Yes	Yes
XEXER	No	Yes	Yes	No
NEXER	Yes	Yes	Yes	Yes
XNEXER	Yes	Yes	Yes	No
LEXER	No	Yes	Yes	Yes
XLEXER	No	Yes	Yes	No
SEXER	No	No	Yes	Yes
XSEXER	No	No	Yes	No

*If TYPE is the first command in a lesson, LESCOR will be set at zero.

J. Top-level Commands and Order of Execution of Commands

Certain commands are known as top-level commands:

TYPE
EXER, XEXER
SEXER, XSEXER
LEXER, XLFXER
NEXER, XNEXER
JMPGE, JMPL
EOL

In the ordinary course of events, each command in a lesson is executed in order. There are a number of exceptions, such as WA, which causes a branch back to a previous section of code, and BRCA, which causes a branch to a specified problem. In some cases, a branch to the next top-level command bypasses all intervening commands. Commands that cause a branch to a following top-level command are

CA, C1, C2, C3
WS
TELL
TYPE

JMPGE and JMPL also cause a branch to a top-level command, but not necessarily the immediately following one. The branch is to a command specified by the coder.

Summary of Top-level Commands

	Displays text?	Wait for student response?	Branches to where?
TYPE	Yes	No	Next top-level command.
EXER, XEXER	Yes	Yes	Next command, either top level or low level.
SEXER, XSEXER	Yes	Yes	Next command, either top level or low level.
LEXER, XLEXER	Yes	Yes	Next command, either top level or low level.
NEXER, XNEXER	Yes	Yes	Next command, either top level or low level.
JMPGE	Yes (optional)	No	If criterion is met, goes to specified problem. Else, goes to next top-level command.
JMPL	Yes (optional)	No	If criterion is met, goes to specified problem. Else, goes to next top-level command.
EOL	No	No	Goes to first problem of next lesson on same strand. If none, goes to "end of strand" routine.

K. Summary of Commands

Top-level commands are marked with an asterisk *.

Op code	Number of arguments	Kind of argument	Comments
LESSON	2	Strand identifier (1 to 6 letters). Lesson number.	Pseudo op code. Marks beginning of a lesson.
*EOL	none		Pseudo op code. Marks end of lesson.
*EXER *LEXER *SEXER	1	Text string.	Displays problem text. Pauses for student response.
*NEXER	1	Text string.	Displays problem number and problem text. Pauses for student response.
*TYPE	1	Text string.	Displays text. Branches to next top-level command.
TELL	1	Text string.	Displays text of correct answer when requested by student. Branches to next top-level command. Default routine causes branch to pause for student response.
BRTELL	4	Strand identifier. Lesson number. Problem number. Text string.	Displays text when requested by student. Branches to specified problem.
REPET	1	Text string.	Displays text when student requests a repeat. Branches to beginning of same exercise.
HINT	1	Text string.	Displays text for hint when requested by student. Pauses for student response.

Op code	Number of arguments	Kind of argument	Comments
EXACT	1	Text string.	Analyzes student response for exact match. Sets SCORE.
MC	1	Text string containing list of letters.	Analyzes response to multiple-choice problems. Sets SCORE to 1 if completely correct, -1 if completely wrong, -2 if partially wrong, -3 if partially correct. Checks form of response.
EQ	1	Text string containing: number and optional number, giving tolerance.	Analyzes response for equality with coded number, within tolerance specified by second number. Sets SCORE. Checks form of response.
KW	1	Text string.	Analyzes response for existence of coded text string. Sets SCORE.
NO	0		Analyzes response for "no" or "n." Sets SCORE. Checks form of response.
YES	0		Similar to NO.
TRUE	0		Checks for "true" or "t." Sets SCORE. Checks form of response.
FALSE	0		Similar to TRUE.
LIST	*undefined*		
SET	*undefined*		
NOTEXACT . . .		Similar to op codes described above, with negation of SCORE.	
NOTKW			

Op code	Number of arguments	Kind of argument	Comments
CA	1	Optional text string.	Executes only if SCORE > 0. Displays message. Branches to next top-level command.
C1	1	Optional text string.	Executes only if SCORE = 1. As for CA.
C2	1	Optional text string.	Executes only if SCORE = 2. As for CA.
C3	1	Optional text string.	Executes only if SCORE = 3. As for CA.
WA	1	Optional text string.	Executes only if SCORE < 0. Branches to pause for student response.
W1	1	Optional text string.	Executes only if SCORE = -1. As for WA.
W2	1	Optional text string.	Executes only if SCORE = -2. As for WA.
W3	1	Optional text string.	Executes only if SCORE = -3. As for WA.
BRCA	4	Strand identifier. Lesson number. Problem number. Optional text string.	Executes only if SCORE > 0. Displays message. Branches to specified problem.
BRWA	4	Strand identifier. Lesson number. Problem number. Optional text string.	Executes only if SCORE < 0. Displays message. Branches to specified problem.
WS	1	Optional text string.	Executes only if SCORE < 0. Displays message. Branches to next top-level command.
WR	1	Optional text string.	Executes only if SCORE < 0. Displays messages. Branches to REPET routine.

Op code	Number of arguments	Kind of argument	Comments
*JMPGE *JMPL	5	Percentage criterion. Strand identifier. Lesson number. Problem number. Optional text string.	Compares student score with criterion. If condition met, branches to specified problem. Else, branches to next top-level command.
CRUNCH	0		Removes spaces from student response.
NEXT	0		Sets student restart point to next problem.
DEFINE	variable		Defines macros.
COMMENT	1	Text string.	Allows insertion of notes to coder. No effect on lesson presentation.

III. THE PDP-10 IMPLEMENTATION

Two major programs are needed for the implementation of the INSTRUCT coding language. One of these, the teaching program, controls the interaction between the student and the computer at the time the student is taking a programmed lesson. The teaching program is actually an interpreter that interprets the problem coding and interacts with the student in accordance with coded instructions. This program is called INST.

The teaching program is not equipped, however, to interpret problem coding as originally coded in the language described previously. The coding as written by the coder must first be transformed into a numeric code that can be understood by the teaching program. This transformation into a numeric code is known as "processing" and is done by a program called the "lesson processor." Processing takes place before a lesson is used by a student and a lesson is processed only once, whereas it is interpreted (by INST) every time the student takes the lesson.

A. The Lesson Processor

After a lesson is coded, the lesson processor must be used to put the coding into a form that can be used by the teaching program (for detailed instructions on how to use the lesson processor see Section IV, pp. 66-67). Each op code is translated into a numeric code; KW, for example, becomes 13,* NOTEQ becomes 22, and so on. Each character in a text string is translated into a teletype character code; the letter A becomes 101, B becomes 102, etc.

In addition to making the relatively straightforward translation described above, the processor also performs the task of inserting standard messages in all appropriate places, that is, wherever the coder has used the short form of an action command.

The lesson processor also makes a directory of each lesson, giving the exact location of each numbered problem, and makes a few necessary calculations, such as the length of each text string and the length of each problem.

The processor then creates a new file containing the processed code for the lesson. The new file is named with the strand identifier and lesson number; if a lesson begins with the command

```
LESSON TEST,29
```

the processed code is put on a file named:

```
TEST.029
```

As for nomenclature, before processing the coded lesson is known as a "text file"; after processing into numeric code, the lesson is known as a "binary file."

---

*For those who are interested, the numeric codes used as examples here are octal numbers. The numeric code produced by the lesson preprocessor is nevertheless known as "binary code."

B. INST: The Teaching Program

After a coded lesson is processed, it can be used by INST as instructions for interacting with students. As soon as a lesson is put on the lesson file, it becomes available to the students.

The purpose and branching structure of the teaching program have been described in detail in previous sections. The teaching program is actually an interpreter that acts in real-time to interpret lesson coding in order to interact with students in the desired way.

C. Limitations Imposed by the Implementation

The implementation of any programming language necessarily imposes some restraints that are not a logical result of the language itself. Rather, the restraints result from considerations of space and time, which present themselves to any programmer working with a real machine. The implementation of the coding language described in this manual is no exception. Every system designer hopes, of course, to provide a system in which the limitations are as innocuous as possible, and here again there is no exception. Following is a list of those limitations that are of interest to the coder.

1. There is no restriction in the number of strands, but restart information is permanently saved for no more than six strands. (The "restart information" simply tells where each student is on each strand.)
2. There are no more than 128 numbered problems per lesson (but there is no restriction on the number of SEXERs).
3. Lesson numbers must be between 1 and 999.
4. Lessons need not be consecutively numbered, but an empty lesson is interpreted as the end of the strand if encountered during an automatic skip from the last problem of the previous lesson. Lessons that follow an empty lesson may be requested by number by the student or may be accessed by a branch command in the coding.
5. The student response is limited to 80 characters.
6. Strand identifiers must be one to six letters.

7. The amount of coding used in an exercise is restricted to 400 computer words of processed code. In practice, only unusually long exercises will exceed this limit, and since it is impossible for the coder to calculate how long his processed code is, the lesson processor checks the length and gives an error message if the exercise is too long. If the error is caused by an exceptional amount of text in the EXER command, you may be able to get around the restriction by using an LEXER (long exercise) command in place of the EXER. The LEXER allows an unlimited amount of text in the problem statement itself, but is more inefficient in operation than other EXER commands. Thus it should be used only when needed.

IV. HOW TO PROCESS A LESSON

The main steps in coding a lesson are:

(1) Coding the lesson, using TVEDIT or ED, the PDP-10 text editors.

(2) Assembling the lesson (the first stage of processing).

(3) Correcting assembly errors, using TVEDIT or ED, and assembling again.

(4) Loading the lesson (the second stage of processing).

(5) Debugging, using INST.

The coder must be able to sign on and off the PDP-10, use TVEDIT or ED, use the PDP-10 assembler for lessons, use the LOADER program, list lesson files, use INST, and use PIP. Manuals are available for TVEDIT, ED, and PIP. Instructions for signing on and off are best given by an experienced person.

Once a lesson has been coded, using TVEDIT or ED, the lesson must be assembled by the following method.

You type	Explanation
R FAIL 30 (CR)	Start the FAIL assembler.
T ← PRO,L15(CR)	Assemble Lesson L15. It will be put on a temporary file named T. If there are syntax errors in the lesson, error messages will be printed. The error must be corrected and the lesson assembled again before proceeding. If there are no syntax errors (unlikely), the computer will print another asterisk.
CTRL-C	Stop the FAIL assembler. At this point, the first stage of processing is complete unless there were error messages. If there were errors, you must correct them and assemble the lesson again before you proceed.

After the first stage of processing is completed, the processed lesson must be loaded as follows:

You type	Explanation
LOAD (CR)	Start the loader program.
T$	Type T followed by alt-mode (alt-mode is Ctrl-Shift-K on the teletypes). If there are no errors, the computer will respond with LOADER nKCORE m+nK MAX xxxx WORDS FREE EXIT ↑C . which indicates that the lesson is loaded into core.
S (CR)	Type S to save a lesson file. When the computer types LESSON SAVED ↑C . the lesson is completely processed and ready for student use.

After the lesson is completely processed, you may get a short form of the lesson listed by using the PRINT program; the PRINT program lists just the problem numbers and the text of the exercise statements without all the other coding.

V. DEBUGGING A LESSON

The first stage in debugging a lesson is to remove syntax errors detected by the FAIL assembler or syntax errors found by the coder trying to save the lesson by typing S.

These are some common syntax errors which will be found by FAIL:

Error	FAIL prints
no LESSON command	UNDEFINED VALUE AFTER _____ TOO MANY BENDS ⋮ UNREC SPC CHR TWO ADDRESS FIELDS OR UNDEF OPCODE UNBAL PARENS ILLEGAL CHAR STARTS EXPRESSION ⋮ FATAL END OF FILE & NO END STMT
no EOL command	FATAL END OF FILE & NO END STMT
missing comma in BR command (e.g., BRCA L,11,8/VERY GOOD/)	FAIL prints the 8 and then prints the text string. Also, TWO ADDRESS FIELDS OR UNDEF OPCODE ILLEGAL CHR AFTER OPERATOR Then each word of the text string is listed separately like VERY UNDEF 000000 GOOD UNDEF 000151
misspelled op code (e.g., HNT + USE THE...)	FAIL prints the misspelled op code and the delimiter if there is one. Then --

```
ILLEGAL CHR AFTER OPERATOR
        :
TWO ADDRESS FIELDS OR UNDEF OPCODE
        :
UNREC SPC CHR
     HNT  UNDEF  000000
     USE  UNDEF  000517
     THE  UNDEF  000465
        :
```

One other error that FAIL will detect is an error in using FAIL. If you do not specify a large enough number in your command

R FAIL 30

FAIL will stop processing the lesson and print

ILL MEM REF AT USER nnnnn
^C
.

Simply increase the number so the command reads

R FAIL 35 (or R FAIL 40, etc.)

35 or 40 should be large enough for almost all lessons and it is best to use the smallest number possible.

You may see these error messages after you type S to start the lesson file:

Message	What to do
PROBLEM NUMBER XXX IS TOO LONG.	Change the EXER or NEXER to an LEXER.
LESSON NUMBER TOO BIG. LIMIT IS 999.	Either revise your numbering scheme if the number is too large or else check the LESSON command to make sure you have a comma between the strand name and lesson number; there must be no extra spaces in the LESSON command.

Message	What to do
ERROR: TOO MANY CHARACTERS IN STRAND NAME. LIMIT IS 6.	Change your strand name to six letters or fewer; if it already is, check the commas and spacing in the LESSON command.
DISK ERROR	Not much can be done about this except to try again, including starting over with the processing if necessary.

After a lesson has been completely processed without errors, it must be carefully debugged before it is ready for student use. There is a variety of ways that this can be done, but we suggest only one method that has proved useful.

To debug a lesson thoroughly, the coder should go through the lesson as a student several times, checking each time for different things. To take a lesson, sign on to the computer and wait for it to print

```
∧ C
.
```

You type	Explanation
. L INST (CR)	Load the INST program.
WHAT COURSE? * G,COD (CR)	The computer will only ask coders and programmers "What course?" Real students will never see this message. Answer by typing a carriage return if you are signed on to the same user number used when coding and processing the lesson. Otherwise, answer by typing the user number where the lessons were coded.
WHAT LESSON DO YOU WANT? (or "WHERE TO?") * L15 (CR)	Type the strand name and lesson number of your lesson.

At this point the computer will type out the first problem of your lesson, and you are ready to begin debugging. These steps are a useful approach to lesson debugging.

(1) Take the lesson as if you were a student, making a few reasonable mistakes, asking for a few hints, repeats, and answers. This is a good way to see how the lesson would look to a student and to see if the overall lesson is reasonable and consistent.

(2) Take the lesson again asking for all repeats and hints and giving all correct answers.

(3) Take the lesson a third time to check all the wrong answers for which there was special coding and then check each TELL.

(4) If necessary, take the lesson once more to check anything else, such as branching with JMP commands.

At any point in your lesson INST may detect some kind of coding error and print an error message like this:

```
I'M SORRY, THERE IS SOMETHING WRONG WITH THIS LESSON.
PLEASE TRY ANOTHER LESSON.
L15-4 --- 3-18
```

On the last line INST, if possible, identifies the specific problem where the error occurs, then gives the error type (see list of error numbers on p. 74), and finally the op code number (see p. 75). So the above error in problem L15-4 is of type 3 in op code 18. After printing the message the program will branch to the choice point.

You may also get an error message that looks like this:

```
I'M SORRY, THERE IS SOMETHING WRONG.
9
```

The numbers will range from 1 to 10. The program will stop. Errors which cause this message to be printed are either system errors, machine errors, or program errors. There is very little you can do about these errors except reprocess your lesson and try again.

After you have debugged your lesson, correct your errors using TVEDIT or ED. Then process the lesson again. This time when you type S to save the lesson the processor will print

```
TO REPLACE L15 TYPE "REE."
↑C
```

After you have typed "REE(CR)" and the program has printed

```
LESSON SAVED.
↑C
.
```

you are ready to look at your lesson once more to see that you made all the changes you intended.

A. Location of Lessons and Coding

Lessons should be coded, processed, and debugged on a user number different from the user number available for student use. Thus a student will not accidentally get a lesson which has not been debugged.

In order for the lessons to be available to students, they must be in the directory for the LES user number (e.g., Q, ES) for your project. Use PIP to transfer the final binary files to this user number.

If the user number used for coding and debugging lessons has the same project number as the LES user number (e.g., A,LES and A,COD), the same options and messages will be available to the students. Otherwise, the general set of options specified for course type Ø (see Course Types, Section VII) will be in effect.

Coding Error Types

	Error	Possible Remedy
1	Problem too long	Use an LEXER for the problem statement.
2	Error in MC argument	Check format. Are all arguments there? Are there unnecessary spaces?
3	Error in EQ argument	
4	Error in form of branch command	
5	Branch to non-existent problem	Check to see that problem is really there, or that proper problem is named.
6	Error in op code number	Probably caused by the processor, so process the lesson again.
777	Unidentifiable coding error	First try processing the lesson again. If that does not help, look over the text of your coding until you find the error.

Op Code Numbers

1	NEXER	26	C1
2	SEXER	27	C2
3	HINT	28	BRCA
4	TELL	29	C3
5	CRUNCH	30	--
6	LEXER, TYPE, EXER	31	--
7	BRTELL	32	--
8	--	33	WA
9	MC	34	W1
10	EQ	35	W2
11	KW	36	BRWA
12	EXACT	37	WS
13	--	38	NEXT
14	--	39	W3
15	YES	40	WR
16	TRUE	41	--
17	NOTMC	42	--
18	NOTEQ	43	--
19	NOTKW	44	JMPGE
20	NOTEXACT	45	JMPL
21	NOTMN*	46	REPET
22	--	47	--
23	NO	48	--
24	FALSE	49	--
25	CA	50	--

*Not yet implemented.

VI. ADVANCED CODING TECHNIQUES

In addition to the op codes used in INST another available coding device is the "macro." A macro is a string of text with a name and optional substitutable arguments. Macros are part of the FAIL assembler language and are processed in the same way as op codes and other FAIL commands. Coders may regularly use three different kinds of macros.

(1) The first kind is not actually part of the coding, but may be used to redefine the standard action (CA, WA, etc.) messages. Using one of these macros saves a coder from having to code a special message for each action code.

(2) The second kind is one without any arguments. It is used to duplicate a section of coding without having to retype the section each time it is used.

(3) The third kind may have a variable number of arguments. This kind of macro is effective where a set of exercises uses the same format, instructions, etc., and where the content varies only in minor ways.

DEFINE is the command used to generate macros. The most basic form of a macro is

```
DEFINE  now   <
   text
>
```

The symbols < and > are used as delimiters. Braces, { and } , may be used instead, but since they are not available on teletype keyboards, it seems better to use < and > .

Every macro is given an identifying name which must begin with a letter. In the example the macro name is "now." The macro name is limited to six characters and may consist of any combination of letters and numbers beginning with a letter. The text may be any string of characters, but for our purposes it is generally either the definition of a standard message or a segment of lesson coding.

A. Changing the Standard Messages

You may perhaps wish to change the standard WA message from "WRONG" to "NO. TRY AGAIN." To do this, write a macro like this:

```
DEFINE   STDWA  <
ASCIZ  / NO.  TRY AGAIN.
/  >
```

"STDWA" is the macro name for "standard wrong answer," and < and > are the macro delimiters. The INSTRUCT delimiter is /; since you may use almost any character as a delimiter in INSTRUCT, the same is true here. However, since > is the closing delimiter for macros, do not use it as an INSTRUCT delimiter within a macro. ASCIZ is a special command to the FAIL assembler which must be included, so that the message is translated into the proper teletype code.

The standard messages for the following action codes may be redefined by the coder:

Op codes	Names for macros
CA	STDCA
C1	STDC1
C2	STDC2
C3	STDC3
WA	STDWA
WS	STDWS
W1	STDW1
W2	STDW2
W3	STDW3

The WR standard message is always the same as the WA standard message.

After the standard message has been redefined for an action code, the lesson processor will insert the new standard message every time

the short form of that code is used. No change will be made in the standard messages of lessons processed earlier. These standard message definitions should be put on a TVEDIT or ED file that is different from your coding files and the file used in your processing (see Section VII, Part F, Storage and Processing of Macros).

B. Macros Without Arguments

In coding you may find that certain segments of code are frequently used. For example, in the programming courses problems often ask a student to stop the INST program and use either the AID interpreter or the BASIC compiler to do a program. For these problems a macro such as this may be useful:

```
DEFINE USEAID <
EXACT ++
WA + USE AID FOR THIS PROBLEM.
+
NOTEXACT ++
WA + USE AID FOR THIS PROBLEM.
+
NEXT >
```

To use the macro, insert its name into your coding in the appropriate place:

```
XEXER + WRITE A PROGRAM THAT WILL PRINT THE CIRCUMFERENCE
AND AREA OF A CIRCLE GIVEN THE RADIUS.  TEST YOUR PROGRAM
ON AID. +
USEAID
```

When the lesson is processed, the code will be expanded to this:

```
XEXER + WRITE A PROGRAM THAT WILL PRINT THE CIRCUMFERENCE
AND THE AREA OF A CIRCLE, GIVEN THE RADIUS.  TEST YOUR
PROGRAM ON AID. +
EXACT ++
WA + USE AID FOR THIS PROBLEM.
+
NOTEXACT ++
WA + USE AID FOR THIS PROBLEM
+
NEXT
```

By using EXACT and NOTEXACT with empty arguments and WA's, we insure that the student cannot go directly on to the next problem. If he types anything besides what is necessary to use AID, he will be reminded to use AID. Since programming problems like this occur frequently in the AID and BASIC courses, a macro like this is used regularly.

Another example of a macro with no arguments is the ENDHW (END Home Work) macro used in other courses.

```
DEFINE ENDHW <
EXACT ++
BRCA 0,0,0,+ GIVE YOUR HOMEWORK TO YOUR TEACHER WHEN
YOU HAVE FINISHED. +
NOTEXACT ++
BRCA 0,0,0,+ GIVE YOUR HOMEWORK TO YOUR TEACHER WHEN
YOU HAVE FINISHED. +
EOL
>
```

The above macro is used only for exercises that a student is to tear off and take home to do. For example:

```
LESSON HW,1

XEXER + THIS IS YOUR HOMEWORK.  TEAR ON THE DOTTED LINE
AND TYPE RETURN.

- - - - - - - -
+
XSEXER  +          HOMEWORK
USE THESE WORDS IN SENTENCES THAT SHOW YOU UNDERSTAND
THEIR MEANINGS.
READY
ALWAYS
ALONE

- - - - - - - -
```

TEAR ON THE DOTTED LINE AND TYPE RETURN.

+

ENDHW

The lesson consists of only two problems. Since the first problem (XEXER) has no analysis or action commands, anything the student types causes the program to go immediately to the XSEXER. The EXACT and NOTEXACT in the macro are part of the XSEXER problem. Anything a student types for that problem causes the program to print

GIVE YOUR HOMEWORK TO YOUR TEACHER WHEN YOU HAVE FINISHED.

and then to branch to the choice point. (Remember: any branch command which gives strand, lesson, problem as 0,0,0 causes a branch to the choice point.)

C. Macros with Arguments

Some portions of coding may be basically the same with only minor text variations. Macros may be used for this kind of coding, too, using arguments to substitute for the variations. In the following example TEXT is the argument name. Whatever is substituted for TEXT is the actual argument.

```
DEFINE NP2 (TEXT)  <
EXER +
     THE NOUN PHRASES IN THIS SENTENCE ARE UNDERLINED.
     TYPE THE SECOND NOUN PHRASE.

TEXT
+
>
```

To use the macro, type the macro name, followed by a space, followed by the argument. For example,

```
NP2  <
OUR PARENTS ARE TAKING A VACATION
>
```

This is processed as

```
EXER +
     THE NOUN PHRASES IN THIS SENTENCE ARE UNDERLINED.
     TYPE THE SECOND NOUN PHRASE.

OUR PARENTS ARE TAKING A VACATION.

+
```

Notice that carriage returns are an important part of the argument. If a macro argument requires more than one line it must be enclosed by the delimiters.

A macro may use the same argument in more than one place. That is, an argument will be inserted wherever the argument name is specified in the macro. Further, there may be more than one argument in a macro.

For an argument name to be identifiable to the processor, it must be set off by spaces. If an argument is used which should not be set off by spaces (for example, the text string for an EXACT), identify it to the processor by a special character called the concatenation character, which is chosen by the coder. The character must be specified in the macro definition between the macro name and the argument list.

```
DEFINE CHECK $ (ANS,MISSP)  <
NOTEXACT +$MISSP$+
WS +NO.  YOU MISSPELLED THE ANSWER.
THE CORRECT SPELLING IS
   ANS
+
EXACT +$ANS$+
CA
WA +WRONG.  TRY AGAIN.
+
>
```

CHECK is a macro with two arguments, ANS and MISSP. ANS is used twice, in the WS message and in the EXACT. $ is the concatenation character. Since it is used when MISSP is the NOTEXACT text string, the processor recognizes that the argument should be substituted there without spaces. But in the WS message MISSP also occurs as part of the message in the word MISSPELLED. Since MISSP is not surrounded on both sides by either a space and/or a $ it is not confused with the argument. The same is true for ANS and ANSWER. Notice that ANS in the last line of the WS has no $ on either side. Since it has places on both sides,

it is clearly an argument. Here is an example of the macro's use:

```
CHECK    PLANNER,PLANER
```

Since neither argument has a carriage return, no delimiters are necessary. The arguments must be separated by commas, and spaces are still important. The above example would be expanded by the processor to this:

```
NOTEXACT +PLANER+
WS +NO.  YOU MISSPELLED THE ANSWER.
THE CORRECT SPELLING IS
   PLANNER
+
EXACT  +PLANNER+
WA +WRONG.  TRY AGAIN.
+
```

Since spacing is important in analysis codes, dc not code the above like this:

```
CHECK  PLANNER,  PLANER
```

You can, of course, write macros which include all the coding for a problem, such as this one for irregular plurals:

```
DEFINE IRRPLU $ (TEXT,WANS,ANS,REG) <
EXER +
TYPE THE NUMBER UNDER THE IRREGULAR PLURAL
TEXT$+
TELL +
THE CORRECT ANSWER IS "$ANS$."
+
NOTEQ +$WANS$+
WA +WRONG.  "$REG$" IS A REGULAR PLURAL.
+
EQ +$ANS$+
CA
WA
>
```

This would be coded as

```
IRRPLU  <
HIS PARENTS HAD FIVE CHILDREN.
 1     2     3   4      5
>,2,5,PARENTS
```

Notice that the first argument requires delimiters, since it contains carriage returns, but commas are sufficient to separate the other arguments.

D. Macros Using IFIDN and IFDIF

Two macro commands may be used to express conditions that must or must not be fulfilled before the code will be processed, and these are contained within the macro definition. IFIDN means "if identical," and IFDIF means "if different." Each of these has three arguments:

IFIDN < macro argument >, < value >, < code to be processed >

"Macro argument" is the name of one of the arguments of the macro. "Value" is a possible value of that argument, and "code to be processed" is the last argument if the specified argument has the specified value. IFDIF is similar, but the code is processed only if the value of the argument is different from the value specified here.

```
IFIDN  <LETTER>, <VOWEL>, < EXACT +$ANS$+
CA
WA + WRONG.  REMEMBER THE VOWELS ARE A, E, I, O, U.
+
>
```

The EXACT, CA, and WA are processed only if the macro call has specified the value of "letter" to be "vowel." Otherwise, that section of code is ignored.

Remember that IFDIF's and IFIDN's are embedded within the macro definition (they may also be embedded within other IFIDN's and IFDIF's. Be careful when you do this; it becomes complicated quickly). Here is a complete sample macro showing an IFDIF and the use of an empty argument.

```
DEFINE  DRILL $ (TEXT,ANS,BR)  <
EXER +
     TYPE THE NUMBER OF THE WORD THAT IS CLOSEST TO THE
     FIRST WORD.
```

```
TEXT
+
IFIDN <BR>, < >, <TELL +
THE ANSWER IS "$ANS$."
+
EQ +$ANS$+
CA
WA  >

IFDIF  <BR>, < >, <BRTELL BR$,+
THE ANSWER IS "$ANS$."
+
EQ +$ANS$+
BRCA BR
WA  >
>
```

Coded with the DRILL macro, a problem would look like this:

```
DRILL  <
SIBLING

1.  FATHER
2.  COUSIN
3.  BROTHER
>,3
```

The above problem has no value specified for the BR argument so BR is considered empty. The conditions for the IFIDN are fulfilled; i.e., BR is empty, so the code for the IFIDN is processed. The condition for the IFDIF is not fulfilled; i.e., BR is not different from the empty argument, so the code for the IFDIF is not assembled.

```
DRILL <
PARENT
1. SISTER
2. MOTHER
3. UNCLE
>,2,<WORDS,2,0>
```

This problem specifies a value for the BR argument so the condition for the IFIDN is not fulfilled, but the condition for the IFDIF is fulfilled: BR is different from the empty argument. The BR specifies the arguments for the BRTELL and BRCA commands. In the macro call it is enclosed in delimiters, since the commas would otherwise be interpreted as separating macro arguments.

Other macro commands may be found in the PDP-10 MACRO manual.

E. Storage and Processing of Macros

Some macros, such as the USEAID and ENDHW macros mentioned in the previous two sections, may be used frequently in your coding. These and any redefinitions of standard messages should be put on a special TVEDIT or ED file (perhaps with a name like DEFS). When you process your lessons you must also process this extra file. The commands are

```
.R  FAIL 35                  This tells FAIL to process the DEFS
*T ← PRO,DEFS,L15            file and then L15 so that macros on
*^C                          the DEFS file will be available for
.                            the lesson text file, L15.
```

It is not necessary, and is in fact wasteful, to keep macros on the DEFS file which are not regularly used. Any macros on the DEFS file are processed every time the DEFS file is used, even if the macros themselves are never called. One way to avoid this, but to still keep the macros available, is to put all rarely used macros on a separate file, perhaps called MACROS. When a macro is needed in a lesson, use the MERGE program to copy it onto the lesson file. After creating or copying all the macros needed onto the text file, you can begin coding the lesson. Then when you process your lesson, including the DEFS file if necessary, the macros will be processed just before the actual lesson.

F. Other Notes for the Advanced Coder

The more experienced coder should remember the less frequently used options of certain op codes to permit more answers or branching schemes. For example, in many situations an EQ with one argument is sufficient. But always keep in mind the optional argument which specifies a permitted range of answers (see Section IIE, p. 34).

The BR and JMP commands have two special forms which should be remembered.

```
JMPL 50,DRILL,0,0
```

will cause the student to be branched to his restart point in the DRILL strand. For example, if a student is in a TEST strand and gets less than 50 percent correct, he should go back to the DRILL strand for more review on that concept, starting wherever he left off before he took the test.

If a student has never worked on the DRILL strand before, the above command uses the first lesson and the first problem of the strand as his restart point.

The other branching commands which can use this "branch to restart point" form are BRCA, BRWA, and JMPGE.

The same branching commands have a form which specifies a branch to the choice point in this way:

```
BRCA 0,0,0
```

Here is an example of the use of this command in the last problem of a lesson:

```
XEXER +
DO YOU WANT TO GO TO THE NEXT LESSON NOW?
+
YES
CA
NO
BRCA 0,0,0,+O.K.+
EOL
```

When setting up more intricate branching schemes based on percentages, the coder should remember several points about the operation of counters in INST. The SCORE counter is changed for every trial of a problem and is only used by the action commands.

The JMP commands use LESCOR. LESCOR has two counters: one contains the total number of problems the student has attempted in the lesson and the other contains the number of problems in that lesson for which he gave a correct answer on the first trial. These counters are reset only after all trials on a problem have been completed. The two counters are carried over from day to day. Both counters are reset to zero when a student changes to a different lesson. Both are also set to zero if the student is branched to the first problem of the current lesson. Any other branching within a lesson maintains the counters.

Finally, the advanced coder may want to look in the FAIL manual and review other macro commands which are available to him. In particular, there are a variety of other conditional commands besides IFIDN and IFDIF.

VII. COURSE TYPES

In order to allow different control characters, different standard messages, etc., INST provides for different course types. Each course type determines a set of definitions for control characters, for standard messages, and for a few other variables. Each course has a letter identifier (the same as the project number) and a course type number. For example:

Course type	Letter ID	Course
1	Q	AID
2	W	BASIC
3	G	Grammar
4	K	Kendall language arts
5	A	Algebra
Ø	none	all unassigned courses

New courses require new course types.

A. Control Characters

There are nine control characters. The sign-off character for all courses is CTRL-Z. Each course must define an "enter" character. All other control characters are optional.

The "enter" character terminates a student's response so that INST will know it is ready to be checked.

The "hint" character calls the HINT routine in INST to print the coded HINT message or the appropriate standard HINT message.

The "tell" character calls the TELL routine to print the coded message or the standard no-tell message.

The "skip" character allows the student to skip to the next problem without doing the current problem.

The "go" character causes a branch to the choice point and prints the standard choice point message.

The "erase" character erases single characters and prints a slash (/) for each character erased.

The "zap" character erases the entire line, prints the standard message, "....ERASED," and gives a new line.

The "repeat" character calls the REPET routine.

If a new course type has not been defined for your course, use the control characters and standard messages for course type Ø. The chart on page 95 gives the definitions of control characters for the current course types.

Definitions of Control Characters

	Course Type					
	0	1	2	3	4	5
Enter	Return	Return	Return	Return	Return	Return
Hint	?	?	?	None	None	?
Tell	Ctrl-G	Ctrl-T	Ctrl-G	Ctrl-G	None	None
Skip	Ctrl-H	Ctrl-H	Ctrl-H	Ctrl-H	Ctrl-L	Ctrl-H
Go	Ctrl-J	Ctrl-G	Ctrl-J	Ctrl-J	Ctrl-G	Ctrl-J
Erase (single characters)	Rubout	Rubout	Rubout	None	None	Rubout
Zap (line erase)	Ctrl-U	Ctrl-U	Ctrl-U	Rubout	Rubout	Ctrl-U
Repeat	Ctrl-A	Ctrl-A	Ctrl-A	Ctrl-A	Ctrl-A	Ctrl-A
Sign-off	Ctrl-Z	Ctrl-Z	Ctrl-Z	Ctrl-Z	Ctrl-Z	Ctrl-Z

B. Standard Messages

Standard action messages, the messages defined for CA, WA, etc., have already been defined for all users of INST (see Section IIF, p. 40). Another set of fixed standard messages that cannot be changed is defined for all courses.

INST message ID	Function	Message
MM4	MC error message	WRONG. TYPE LETTERS ONLY.
MM5	EQ error message	WRONG. ANSWER BY TYPING A NUMBER.
MM11	No-Tell message	NO ANSWER WAS WRITTEN.
MM14	Disk or file error message	I'M SORRY. THERE IS SOMETHING WRONG.
MM18	Response too long (i.e., more than 80 characters)	TOO MUCH ... START OVER.
MM19	Line erased	...ERASED.
MM23	Course choice (coders only)	WHAT COURSE?
MM24	Temporary name (for sign-on)	PAL
MM25	Name request for automatic enroll	TYPE YOUR FULL NAME, THEN THE RETURN KEY.
MM26	Name check for automatic enroll	DO I HAVE YOUR NAME RIGHT? TYPE YES OR NO, THEN THE RETURN KEY.
MM28	Drill time expired	THAT'S ALL FOR TODAY.
MM29	Time-out message	PLEASE ANSWER SOON.
MM30	Not enrolled message	SORRY. I CANNOT FIND YOUR ENROLLMENT RECORDS.

The last set of standard messages is the set which varies by course type. For every new course type these messages must be specified.

INST message ID	Function	Example
M1	Choice point message	WHAT LESSON DO YOU WANT?
M2	Choice point hint message	TYPE THE NAME AND NUMBER OF THE LESSON YOU WANT.
M3	Choice point error message	I DON'T UNDERSTAND YOU. TRY AGAIN OR TYPE A QUESTION MARK FOR HELP.
M6	Yes-No error message	WRONG. TYPE "YES" OR "NO."
M7	Sign-off message	GOODBYE.
M8	True-False error message	WRONG. TYPE "TRUE" OR "FALSE."
M9	No-Hint message	NO HINTS WERE WRITTEN.
M10	No-More-Hints message	THERE ARE NO MORE HINTS.
M12	Sign-on message	HELLO, (the sign-on routine then fills in the student's name).
M13	Bug message	THERE IS A BUG IN THIS PROGRAM. TRY ANOTHER PROBLEM.
M15	Lesson not available	SORRY. THAT LESSON IS NOT READY YET.
M16	Problem not available	SORRY. THAT EXERCISE IS NOT READY YET.
M17	End-of-strand message	YOU HAVE REACHED THE END OF THIS STRAND. (After printing the message, the end-of-strand routine then branches to the choice point.)
M20	Number too long	PLEASE LIMIT NUMBERS TO 9 SIGNIFICANT DIGITS. TRY AGAIN.

M21	Exponent too large	EXPONENT TOO LARGE. TRY AGAIN.
M22	Number base too large	SIZE OF NUMBER IS OUT OF BOUNDS. TRY AGAIN.
M27	Ready message	* (This is the character printed when the program is ready for the student's response.)

Note: ID numbers preceded by MM are fixed; those preceded by M are course variables.

C. Other Course Type Variables

A few other features of INST are variable and must therefore be established for each course type.

Each course type must specify whether or not a student begins at the choice point each day. If yes, a student is asked what lesson he wants as soon as he signs on. If no, he starts immediately at the problem he was working on when the previous session ended.

Another variable is the maximum number of trials per exercise. This is to be specified either as a number or as "MAX." MAX is an extremely large number so it appears to the student that he has unlimited chances. Otherwise, the number specifies how many trials are allowed. If the student uses the maximum number of trials permitted him, INST prints the TELL message, if any, and then branches to the next problem.

The time allowed per session must also be set. The time is expressed in minutes, and after the indicated number of minutes the student is automatically signed off. He may sign on again if he or his teacher wants him to.

Each course also identifies up to six named strands as course variables. The restart points of these specific strands are saved. For the coder this means that there may be branches to the restart point in one of these strands (i.e., BRCA L,0,0) instead of a specific problem. It also permits a student working on one strand to go to his restart point in a different strand, e.g., LES, by going to the choice point and simply typing LES. Other strands may be used by the coder, but restart points will not be saved for strands not specifically named as course variables.

The chart on page 100 shows the values given for these variables by each course type.

Definitions of Other Course-dependent Variables

	Course Type					
	0	1	2	3	4	5
Does student start at choice point each day?	Yes	Yes	Yes	No	Yes	Yes
Number of trials allowed per exercise	Max	Max	Max	3	3	5
Time allowed per day (in minutes)	Max	120	120	20	20	120
Time allowed for response (in seconds)	Max	600	600	180	180	180
Name of Strand 1	L	L	L	INTRO	INTRO	LES
Name of Strand 2	None	R	H	DIR	SENT	HW
Name of Strand 3	None	X	T	None	QUES	Q
Name of Strand 4	None	S	S	None	None	T
Name of Strand 5	None	None	R	None	None	None
Name of Strand 6	None	None	X	None	None	None

INDEX

(Continued from inside front cover)

96 R. C. Atkinson, J. W. Brelsford, and R. M. Shiffrin. Multi-process models for memory with applications to a continuous presentation task. April 13, 1966. (J. math. Psychol., 1967, 4, 277-300).
97 P. Suppes and E. Crothers. Some remarks on stimulus-response theories of language learning. June 12, 1966.
98 R. Bjork. All-or-none subprocesses in the learning of complex sequences. (J. math. Psychol., 1968, 1, 182-195).
99 E. Gammon. The statistical determination of linguistic units. July 1, 1966.
100 P. Suppes, L. Hyman, and M. Jerman. Linear structural models for response and latency performance in arithmetic. (In J. P. Hill (ed.), Minnesota Symposia on Child Psychology. Minneapolis, Minn.: 1967. Pp. 160-200).
101 J. L. Young. Effects of intervals between reinforcements and test trials in paired-associate learning. August 1, 1966.
102 H. A. Wilson. An investigation of linguistic unit size in memory processes. August 3, 1966.
103 J. T. Townsend. Choice behavior in a cued-recognition task. August 8, 1966.
104 W. H. Batchelder. A mathematical analysis of multi-level verbal learning. August 9, 1966.
105 H. A. Taylor. The observing response in a cued psychophysical task. August 10, 1966.
106 R. A. Bjork. Learning and short-term retention of paired associates in relation to specific sequences of interpresentation intervals. August 11, 1966.
107 R. C. Atkinson and R. M. Shiffrin. Some Two-process models for memory. September 30, 1966.
108 P. Suppes and C. Ihrke. Accelerated program in elementary-school mathematics--the third year. January 30, 1967.
109 P. Suppes and I. Rosenthal-Hill. Concept formation by kindergarten children in a card-sorting task. February 27, 1967.
110 R. C. Atkinson and R. M. Shiffrin. Human memory: a proposed system and its control processes. March 21, 1967.
111 Theodore S. Rodgers. Linguistic considerations in the design of the Stanford computer-based curriculum in initial reading. June 1, 1967.
112 Jack M. Knutson. Spelling drills using a computer-assisted instructional system. June 30, 1967.
113 R. C. Atkinson. Instruction in initial reading under computer control: the Stanford Project. July 14, 1967.
114 J. W. Brelsford, Jr. and R. C. Atkinson. Recall of paired-associates as a function of overt and covert rehearsal procedures. July 21, 1967.
115 J. H. Stelzer. Some results concerning subjective probability structures with semiorders. August 1, 1967
116 D. E. Rumelhart. The effects of interpresentation intervals on performance in a continuous paired-associate task. August 11, 1967.
117 E. J. Fishman, L. Keller, and R. E. Atkinson. Massed vs. distributed practice in computerized spelling drills. August 18, 1967.
118 G. J. Groen. An investigation of some counting algorithms for simple addition problems. August 21, 1967.
119 H. A. Wilson and R. C. Atkinson. Computer-based instruction in initial reading: a progress report on the Stanford Project. August 25, 1967.
120 F. S. Roberts and P. Suppes. Some problems in the geometry of visual perception. August 31, 1967. (Synthese, 1967, 17, 173-201)
121 D. Jamison. Bayesian decisions under total and partial ignorance. D. Jamison and J. Kozielecki. Subjective probabilities under total uncertainty. September 4, 1967.
122 R. C. Atkinson. Computerized instruction and the learning process. September 15, 1967.
123 W. K. Estes. Outline of a theory of punishment. October 1, 1967.
124 T. S. Rodgers. Measuring vocabulary difficulty: An analysis of item variables in learning Russian-English and Japanese-English vocabulary parts. December 18, 1967.
125 W. K. Estes. Reinforcement in human learning. December 20, 1967.
126 G. L. Wolford, D. L. Wessel, W. K. Estes. Further evidence concerning scanning and sampling assumptions of visual detection models. January 31, 1968.
127 R. C. Atkinson and R. M. Shiffrin. Some speculations on storage and retrieval processes in long-term memory. February 2, 1968.
128 John Holmgren. Visual detection with imperfect recognition. March 29, 1968.
129 Lucille B. Miodnosky. The Frostig and the Bender Gestalt as predictors of reading achievement. April 12, 1968.
130 P. Suppes. Some theoretical models for mathematics learning. April 15, 1968. (Journal of Research and Development in Education, 1967, 1, 5-22)
131 G. M. Olson. Learning and retention in a continuous recognition task. May 15, 1968.
132 Ruth Norene Hartley. An investigation of list types and cues to facilitate initial reading vocabulary acquisition. May 29, 1968.
133 P. Suppes. Stimulus-response theory of finite automata. June 19, 1968.
134 N. Moler and P. Suppes. Quantifier-free axioms for constructive plane geometry. June 20, 1968. (In J. C. H. Gerretsen and F. Oort (Eds.), Compositio Mathematica. Vol. 20. Groningen, The Netherlands: Wolters-Noordhoff, 1968. Pp. 143-152.)
135 W. K. Estes and D. P. Horst. Latency as a function of number or response alternatives in paired-associate learning. July 1, 1968.
136 M. Schlag-Rey and P. Suppes. High-order dimensions in concept identification. July 2, 1968. (Psychom. Sci., 1968, 11, 141-142)
137 R. M. Shiffrin. Search and retrieval processes in long-term memory. August 15, 1968.
138 R. D. Freund, G. R. Loftus, and R.C. Atkinson. Applications of multiprocess models for memory to continuous recognition tasks. December 18, 1968.
139 R. C. Atkinson. Information delay in human learning. December 18, 1968.
140 R. C. Atkinson, J. E. Holmgren, and J. F. Juola. Processing time as influenced by the number of elements in the visual display. March 14, 1969.
141 P. Suppes, E. F. Loftus, and M. Jerman. Problem-solving on a computer-based teletype. March 25, 1969.
142 P. Suppes and Mona Morningstar. Evaluation of three computer-assisted instruction programs. May 2, 1969.
143 P. Suppes. On the problems of using mathematics in the development of the social sciences. May 12, 1969.
144 Z. Domotor. Probabilistic relational structures and their applications. May 14, 1969.
145 R. C. Atkinson and T. D. Wickens. Human memory and the concept of reinforcement. May 20, 1969.
146 R. J. Titiev. Some model-theoretic results in measurement theory. May 22, 1969.
147 P. Suppes. Measurement: Problems of theory and application. June 12, 1969.
148 P. Suppes and C. Ihrke. Accelerated program in elementary-school mathematics--the fourth year. August 7, 1969.
149 D. Rundus and R.C. Atkinson. Rehearsal in free recall: A procedure for direct observation. August 12, 1969.
150 P. Suppes and S. Feldman. Young children's comprehension of logical connectives. October 15, 1969.

(Continued on back cover)

(Continued from inside back cover)

151 Joaquim H. Laubsch. An adaptive teaching system for optimal item allocation. November 14, 1969.

152 Roberta L. Klatzky and Richard C. Atkinson. Memory scans based on alternative test stimulus representations. November 25, 1969.

153 John E. Holmgren. Response latency as an indicant of information processing in visual search tasks. March 16, 1970.

154 Patrick Suppes. Probabilistic grammars for natural languages. May 15, 1970.

155 E. Gammon. A syntactical analysis of some first-grade readers. June 22, 1970.

156 Kenneth N. Wexler. An automaton analysis of the learning of a miniature system of Japanese. July 24, 1970.

157 R. C. Atkinson and J. A. Paulson. An approach to the psychology of instruction. August 14, 1970.

158 R.C. Atkinson, J.D. Fletcher, H.C. Chetin, and C. M. Stauffer. Instruction in initial reading under computer control: the Stanford project. August 13, 1970.

159 Dewey J. Rundus. An analysis of rehearsal processes in free recall. August 21, 1970.

160 R.L. Klatzky, J.F. Juola, and R.C. Atkinson. Test stimulus representation and experimental context effects in memory scanning.

161 William A. Rottmayer. A formal theory of perception. November 13, 1970.

162 Elizabeth Jane Fishman Loftus. An analysis of the structural variables that determine problem-solving difficulty on a computer-based teletype. December 18, 1970.

163 Joseph A. Van Campen. Towards the automatic generation of programmed foreign-language instructional materials. January 11, 1971.

164 Jamesine Friend and R.C. Atkinson. Computer-assisted instruction in programming: AID. January 25, 1971.

165 Lawrence James Hubert. A formal model for the perceptual processing of geometric configurations. February 19, 1971.

166 J.F. Juola, I.S. Fischler, C.T. Wood, and R.C. Atkinson. Recognition time for information stored in long-term memory.

167 R.L. Klatzky and R.C. Atkinson. Specialization of the cerebral hemispheres in scanning for information in short-term memory.

168 J.D. Fletcher and R.C. Atkinson. An evaluation of the Stanford CAI program in initial reading (grades K through 3). March 12, 1971.

169 James F. Juola and R.C. Atkinson. Memory scanning for words versus categories.

170 Ira S. Fischler and James F. Juola. Effects of repeated tests on recognition time for information in long-term memory.

171 Patrick Suppes. Semantics of context-free fragments of natural languages. March 30, 1971.

172 Jamesine Friend. Instruct coders' manual. May 1, 1971.