Computer based instruction in computer programming—A symbol manipulation-list processing approach

by P. LORTON, JR. and J. SLIMICK

Institute for Mathematical Studies in the Social Sciences
Stanford, California

INTRODUCTION

Since February, 1969, a computer based course in computer programming has been running at an “inner city” high school in San Francisco, California. Each day ninety high school juniors and seniors in classes of fifteen interact with a course designed to teach the fundamentals of computer programming for business applications. For fifty minutes a day each student is on-line with a computer located thirty miles away on the Stanford University campus. The purpose of this paper is to describe the rationale and the major components of the software system used to implement the project.

Lesson material and programming problems for the students are presented on teletypewriters linked via telephone lines to the Computer Based Laboratory of the Institute of Mathematical Studies in the Social Sciences on the Stanford University campus. In this laboratory are several computers which form a unique system for presenting instructional material.

The main computer in the system for this project is a Digital Equipment Corporation model PDP-1D. The PDP-1D is a single address, 18 bit binary machine. The machine has 32,768 words of core memory of which 20,480 words are used by the time-sharing operating system. User programs are permitted up to 12,288 words of core. The time-sharing system allows up to 26 users to run concurrently on the computer. This is made possible by the addition to the PDP-1D of a very high speed drum with 26 tracks, each capable of holding 4096 words. The time-sharing system swaps programs in and out of core memory very rapidly using a simple priority scheme based on “time-slicing.” Because of the necessity for user micro time-sharing the programs in this project occupy 10 of the 26 available tracks.

The PDP-1 communicates with the students at the high school through a smaller computer (DEC PDP-8) used to buffer text output. A PDP-8I has been installed at the school to perform a similar function at the other end of the line. Collins data sets were used in place of the PDP-8I during the first year.

Aim and purpose of the course

The main goal of this course is to present in very general terms the concept of a digital computer as a tool for solving business-related problems. As computers proliferate in business and industry there will be an increased demand for people who can see their jobs in terms amenable to computerized operation. Such tasks as filing and stockroom control, now available to minimally trained individuals, will soon require personnel able to see and solve problems in terms understandable to a computer.

With the goal of training for applications on these kinds of problems, the need for something other than a “formula translation” approach is evident. Using filing and stock control as sample problem areas, an approach which stresses symbol-manipulation and list-processing suggests itself. Inventories can easily
be viewed as ordered pairs (a symbol-manipulation concept) of item names and counts. Retrieving information from a file can be thought of as a "tree search" (a list-processing concept).

The advantages of teaching a symbol manipulation-list processing (abbreviated: SMLP) language are best shown in an analysis of the properties of SMLP languages.

A. SMLP languages operate primarily on symbols and sets of symbols and, secondarily, on quantities. This implies that problems as conceptually complex as text scanning become more manageable. Once text scanning becomes manageable, then many applications such as natural language-based information retrieval or dialogue systems for management information collapse into programmable problems. The power of an approach which emphasizes symbol manipulation is that conceptually difficult problems often become readily programmable.

B. The list structure in SMLP languages provides an absolutely general form of data and program storage. A programmer, given a universal data storage facility, can give some attention to optimization of the structure of his data. The optimization of data structure cannot be over emphasized since information retrieval (among other applications) is not economically possible without structuring the data so that the computer answers efficiently the most frequently asked questions.

C. SMLP languages teach the use of pointers and indices. While properly part of (B), the simplest definition of a pointer is that it is a quantity that specifies the location or existence of some other quantity; an index can be defined as a quantity specifying some base location. The concepts of pointer and index are useful in teaching the manipulation of data by using references rather than moving blocks of data from one place to another. An immediate example of an application of pointers is data sorting.

D. SMLP languages allow simple implementation of push-down stacks. While not of great intrinsic value, push-down stacks simplify the calling and structure of subroutines, particularly recursive ones.

E. SMLP languages simplify the treatment of name scope problems in a hierarchical store. A fundamental concept of symbolic programming is that a quantity can have a name; furthermore, it may be desirable to limit the area of the program in which a given name refers to a particular quantity. Thus, it is desirable to have a method of associating a given name to the relevant quantity on the basis of "area"; this association is referred to as "name-scope."

In general, language possessing properties A-E provide exceptionally general approaches to programming digital computers. It can also be pointed out that the Common Business Oriented Language (COBOL) resembles this kind of language more than it resembles a "formula translation" language. The general concepts available through an SMLP language would, it is believed, be of considerable help to the students in their future efforts to build an understanding COBOL and related languages.

**Basic concepts**

Good computer programming, under the philosophy advanced here, depends on the understanding of certain concepts not particularly oriented toward any one machine or language. The basic concepts which seem necessary for understanding the kind of applications programming taught in this project seem to divide into concepts which are related to making a stored program machine work for the user and concepts which are related to what is felt to be the basic task of business applications programming: symbol manipulation-list processing. It is these concepts which form the basic content for this course.

The first nine general concepts in the following list are of the first type. The tasks described are all associated with the how and why of making stored program machines do the work required of them.

**I. "Machine" related concepts:**

A. Stored Program. Refers to the ability to have a set of imperative actions implying some overall task stored in a machine which can execute it in some sequential fashion.

B. Stored Data. Refers to the ability of a machine to store quantities like "stored program" actions but not encompassing an overall meaning.

C. Variable. Refers to the ability to name some part of the stored program and refer to the properties or value of this part through reference to its name.

D. Operations. Refers to the capabilities contained in the Central Processing Unit. Two main classes of operations are felt important: Arithmetic and Non-Arithmetic.
E. Addressing. Refers to the capability of pointing to various parts of the stored program as well as the ability to form data into clusters or arrays in some useful way. Three sub-concepts are felt noteworthy: Indexing, Base addressing, and Indirect addressing.

F. Branching. Refers to the ability of a stored program to reorder the sequence of events it performs in completing a task.

G. Loops. Refers to the ability to re-execute a subsequence of the stored program to complete a repetitive task.

H. Blocks/Sub-Programs/Procedures. Refers with minor differences in emphasis to sub-groupings of the stored task which form semi-self contained programs often capable of being introduced into the main event sequence by being “called.”

I. Input-Output. Refers to the machine’s methods for listening and talking to the user.

The following concepts are more directly related to the symbol manipulation-list processing approach to the problem space than they are to the problem of making a machine work. This does not mean that the concepts listed above are unrelated to issues associated with the nature of the problem space. Neither does it mean that a symbol manipulation-list processing language is unsuited to presenting them.

II. “Language” related concepts:

A. Data Handling. Refers to the method of viewing and manipulating the data a program is to handle.

B. Recursion. Refers to a “self calling” ability of sub-blocks of the program in an SMLP type language.

C. Arrays and Strings. Refers to a more general and efficient way of clustering stored data so that its manipulation becomes a simpler task.

D. Data Structures. Refers to named functions which use indexing and pointers to locate elements in the stored data. Examples might be “trees,” “lists,” “graphs”, etc.

Languages selected for the project

Given the conclusions on the advantages of teaching a “symbol manipulation-list processing” language and the fact that some machine level concepts might usefully be introduced into the course, a language appropriate to each conclusion was selected: a simple assembly language and a fundamental SMLP language. Each of these languages is briefly described below.

Major components of the project

The implementation of the conclusions reached in the preceding discussion involved developing three separate programs which, when loaded into the PDP-1-D, operate as the software system for this project. The three programs include a “driver” (SLAKER) to supervise the interaction of the student with the curriculum material and the language processors, an interactive assembly language processor (SIMPER), and an interpretive SMLP language processor (SLOGO). Each of these parts of the software package is described below. Appendix A contains a sample lesson illustrating many of the components described below.

Major component: SLAKER

Introduction

SLAKER [Slimick-Lorton All Knowing Educator Routine] is designed to provide the interface between the student at a teletypewriter and the curriculum material of the project. The over-riding concern in the development of this driver was to provide as much freedom and flexibility for each user as is consistent with service at reasonable intervals.

If a student’s program would cause a real machine to enter an infinite loop or write over his data, then this would happen to him in the instructional setting. Certain obvious restrictions have been placed on this goal. A student’s work is not free to “clobber” other users (although this might well happen on a “real” machine). A student can wipe out his own effort and experience the pain of having to recover from the error.

Functions

The balance of the description of SLAKER is devoted to the major functions it is designed to perform.

1. Text Emission

One of the major tasks SLAKER has is the presentation of problems to the student at his teletypewriter. Several of the disc files attached to the driving program contain the curriculum material which is organized into four sequences of lessons and problems through which the student is to proceed. In addition to the lesson-text, the problem code contains certain values which indicate various subsections of the problem such
as the “correct answer” or the “hint,” as well as the problem type to the driver.

The four strands into which lessons and problems are grouped for this project are: Lesson, Homework, Extra Credit, and Test. For problems in the Lesson strand, SLAKER is charged with waiting until the student enters the correct answer before going on to the next problem. With the other three strands, SLAKER presents the next problem as soon as any answer is entered. In every case the student is informed of the correctness of his answer.

2. Response Evaluation

After emitting the text for a problem to the user, SLAKER monitors his output, collecting it as an answer. When the user enters an “evaluate my work” request, SLAKER checks his answer according to the type of problem the student was given.

A. Multiple Choice

Under this format the answer is first compressed so that all duplicate characters are eliminated. Then the answer is searched for matches with the characters recorded as the correct answer. Up to twenty characters are collected from the student as possible answers for problems stated in this format. Only alphabetic characters are collected so that spaces, punctuation marks, or numbers can be inserted in the answers without affecting the correctness of the alphabetic string.

B. Constructed Response

When the student’s input is a response to this type of problem, all the characters he types, with the exception of carriage returns and line feeds, are collected. The checking routine then examines the response string looking for two kinds of characters: those that must be present and those designated as optional. The search and match routine is of such generality that it is felt all possible correct answers will be marked correct if they are defined in the curriculum.

C. Anticipated Alternative

Although not a separate type of problem, this checking capacity is a separate skill of SLAKER. If alternative answers are expected they can be specified and checked for. If a correct response is not found, then the answer evaluation routine checks the student’s effort, in the same fashion, against the strings specified as possible alternatives. If a match is found, then an appropriate comment is given and the student is told to try again, just as if he were wrong. At present this capability is available on constructed response problems and single choice-multiple choice problems.

D. Programming Problems

Evaluation of these problems is done by asking the student questions about his program after he wrote and debugged it with given data. This method of evaluation allows the student flexibility in programming a different solution than the solution the curriculum writers had in mind.

3. Communication with Language Interpreters

Since the main aim of the course is to provide rich and varied experience in programming, a main responsibility of SLAKER is readily to provide this contact. Each language differs slightly in how it wants to be told a student is using it but, basically, SLAKER’s role is to make the initial contact with the language processor, pass subsequent information to it and await the user’s indicated wish to return to the main program.

4. Special requests from the Student Station

The following activities can be requested from a student station. As a group they provide the student with considerable flexibility in how he proceeds through the course.

A. Restart Station. Allows a user to request a station be restarted from the sign-on point. Used to correct improper sign-on efforts by students.

B. Sign-off Station. Allows a user to terminate his lesson when he is ready. Part of the execution of this command involves storing where the student left off on his history file so that he may restart from this point on the following day.

C. Go to Choice Point. Places the user at a point where one of the following choices can be made:

   1. Return to Last Problem. Allows the student to continue working from where he last signed off in the strand he specifies.
   2. Go to Specific Lesson. Allows the student to begin working on the lesson number in the strand he indicates.
   3. Attach a Language Processor. Allows the student to call forth one of the language processors available in the course.
D. Skip Problem. In the Lesson strand, only a correct answer will advance a student on to the next problem. This feature allows a student to skip out of this loop. As the next problem is called, the correct answer to the skipped problem is printed.

E. Give Hint. Commands SLAKER to print the “hint” provided for the particular problem.

F. Erase Answer. The user has the option of erasing all of the answers he has typed or merely the last character. Erasing the last character can be repeated until the entire answer is erased if wished.

G. Communicate with Stanford Monitor. This feature allows student stations to type messages to the monitor teletypewriter at Stanford. Usually, its use is reserved for the classroom teachers who may want to correct a lesson, enter a new student, or ask a question. As part of this feature it is also possible to communicate from the monitor teletypewriter to any of the student stations.

Major component: SIMPER

Introduction

SIMPER [Simple Instructional Machine for the Purpose of Educational Research] represents an attempt to make available to the student at a teletypewriter a simple computer which he can program in a manner analogous to “assembly language programming” on digital computers of modest size.

This instructional package can be most easily understood when viewed as consisting of two main parts: a machine (SIMPER) and an assembler (SASS). The latter is designed to generate the machine code for SIMPER. The “machine” is a mythical digital computer which can be described in a formal way and for which programs can be written. Although the machine responds to 18 bit instructions in its “machine language,” there is no direct access to the machine via 18 bit numbers. The purpose of the machine is to teach students to program so the machine is programmable only through a symbolic assembly language.

The assembler generates code for SIMPER from Assembly Language instructions typed by the student. Assemblers generate code instruction by instruction. This one generates code for SIMPER immediately after each instruction is typed in by the student. This feature enables the student to receive immediate correction for most syntax errors and, when the student availa himself of the option, each line of code can be checked immediately to assure the student that the assembler translated the student’s instruction as he wished.

The current version of SIMPER is designed to time share up to 15 students concurrently. The interpreter occupies 4096 words of PDP-1D core memory while the arrays representing the simulated machines for all 15 possible users occupy an additional 4096 words of memory.

Description of the SIMPER machine

SIMPER is a fixed-point, single address machine with a memory of variable size (currently 128 words). Operations are performed in two general purpose registers. Instructions are six digits in length: two digit operation code, one digit register specification field, and a three digit address field. At present, 16 operations can be performed.

The size of the machine’s memory is variable depending on the available space. For this project the memory size is 128 decimal (200 octal) locations. This size was chosen because it allows the fifteen students to run parallel in the space available on the PDP-1D and it also means the students’ daily programming effort can be “saved” on a disk scratch file of convenient length, enabling the student to continue programming efforts from session to session.

Operation of the SIMPER machine

SIMPER runs by executing the six digit number it finds in the memory location pointed to by the program counter. The program counter is updated as part of the instruction-fetching activity. An instruction by instruction-execution of a program is printed on the Teletype. While thus being able to monitor the execution of his program, hopefully, a student is given special insight into how each instruction operates and how a sequence of instructions can be converted into meaningful work. This “printing out” of the execution sequence also slows down the speed of execution so that the work of the machine is easily followed. The student can also watch the effects of “bugs” arise and develop into problems which require attention. This feature is intended to make the debugging of machine language programs an easier task. A special flag can be set at execution time to suspend this feature. Execution speed is then improved by a factor of four.
The assembler

Description

The assembler receives its instructions from a student through a teletypewriter keyboard. Each student interacting with the program is listened to for characters which are collected as an instruction to be assembled. Students are served by the assembler in a manner which both time shares and "oils the squeaky wheel first."

When the student is given a problem involving assembly language programming, he is told to sign on to SIMPER. He calls the choice point option and, in response to "Where to?", types "SIMPER." The student is then in contact with the assembler. He is informed that he may now write his program and columns labeled "LOC" and "INSTRUCTION" are created. In the LOC column the assembler prints the number of the memory location into which the instruction being written will be assembled. The assembler then awaits an instruction from the student. The student types his instruction and an indicator that he is finished. The assembler immediately examines the text string and attempts to generate SIMPER executable code. If all is in order, programming advances to the next memory location. If all is not in order, the assembler generates an appropriate error message. By assembling in real-time after each instruction is entered, the assembler can give immediate feedback on syntax errors to the student.

Major component: SLOGO

SLOGO (Stanford LOGO) is the L.M.S.S.S. implementation of LOGO, a computer language developed by Wallade Feurzig and Seymour Papert of Bolt, Beranek, and Newman expressly for teaching the principles of computer programming. SLOGO is similar to LISP 1.5 in that both are left prefix languages, both have a simple type of function definition, and both have similar sets of primitive operations. SLOGO functions, unlike LISP, have predefined numbers of arguments which, along with the left prefix notation, allow SLOGO to require minimal user punctuation.

While SLOGO is an ideal symbol manipulation and string processing language, it has substantial weakness in not providing structures that are effectively lists of lists à la LISP 1.5. While generality is very desirable to the programmer, the choice of LISP 1.5 as the symbol manipulation-list processing language for this project posed such severe curriculum problems that the attempt to use it was abandoned; thus, SLOGO, which has less generality, was implemented instead.

SLOGO currently time shares five concurrent users; each user has a 4096 word drum track that contains his own functions, execution stack, etc. SLOGO is a re-entrant program when executing commands from a user, but it is not re-entrant with respect to console input and the queuing apparatus. The currently available functions with short definitions attached are listed in Appendix B. In the following sections, first, the basic data types used in SLOGO are described, and immediately thereafter is a discussion of the two processing modes of SLOGO.

Data types in SLOGO

There are three basic data types in SLOGO: word, sentence, and number. A brief explanation of each follows.

1. A "word" consists of a string of letters, digits, or certain punctuation marks; punctuation marks that cannot be used are blank, single quote, ">", "<", "", and possibly others that depend on which version of SLOGO is being run.

2. A "sentence" consists of a group of words. Although one can argue that sentences could consist of one or more words, to avoid ambiguity we assume that sentences consist of two or more words.

3. A "number" consists of a string of decimal digits plus a leading minus sign, if the number is negative. The largest number acceptable is ± 131,071.

There are three methods of referring to data: function values, pointer variables, and literals. A brief explanation of each follows.

1. A literal is a direct reference to the indicated data. Word and sentence literals are written with the single quote (') surrounding the desired data. Literal numbers appear as the number itself, without quotes. A quoted number is assumed to be a word.

Example:

The following are word literals:

'AARDVARK'
'45'
'3A'
'MIXTEC'
'THISISAWORD'

The following are sentence literals:

'AARNOld IS A APATHETIC AARDVARK'
'ONTogeny recapitulates phyloge-
The following are number literals:
1
1776
-10
131071

(2) Function values. Most of SLOGO's built-in functions and all of the defined functions return a value. This value may be subsequently referenced by other functions, and the type of this function may be any of the three basic types.

(3) Pointer variables are in reality name pairs, where one part of the pair is the name and the other part is the value. Names must have type values of either word or sentence but never number. The value type can be word, sentence, or number. Names are written inside closed symbols, which can be either "<" and " > " or " - " for left and right sides.

Example:
<ANTEATER>
<NURNDY IS A GAME>
_POINTER_
<A>

The peculiar literal " is accepted by the read-in routines, can be generated internally, and is always printed by SLOGO as "NIL".

To illustrate the difference between literals and pointer variables, assume there is a name pair whose name is "HEROINE" and whose value is the sentence "OUR GAL SUNDAE."
The value, then, of <HEROINE>
is OUR GAL SUNDAE.
The value of 'HEROINE' is HEROINE.

SLOGO processing modes

SLOGO operates in two modes, command and definition. There is a special character printed at the extreme left-hand end of the type line to indicate which mode SLOGO is in.

"Command" mode is indicated by a "->" ("right arrow") sign, and is the normal mode of operation. In command mode, as soon as a line of functions and arguments is typed in, terminating with a "." (period), the line is converted to a Polish string of interpretive code and then interpreted by the SLOGO interpreter. Upon detection of an error or the successful execution of the Polish string of code, whatever output produced is printed (if PRINT is used) and SLOGO returns to a listen state while the next line is being typed in.

“Definition” mode is indicated by a "->" ("right arrow") sign, and is the exceptional mode of operation. It is entered from command mode when an input line has been terminated with a period and begun with a "TO". At that point definition mode is entered and cannot be left until the command "END" is entered. There is no attempt at function execution while in definition mode. The only use of definition mode is to define a SLOGO function by entering successive lines of functions and arguments. During definition mode, checking is done on the function names, validity of arguments, etc., but no functions are executed.

SUMMARY

The purpose of this paper has been to describe the software and corresponding rationale for a project designed to teach high school students how to use computers. The main thought behind the project is that, especially for business applications, an approach which stressed symbol manipulation and list processing skills would very likely prove of long-term use to the students.

To implement this course, a three-part software package has been developed which provides guided interaction for each student with important programming concepts. The software package includes a "driver" to shepherd the student through the course material, an assembly language interpreter to provide him with an understanding of basic machine operation and a symbol manipulation-list processing language interpreter to provide him with experience in solving problems in a suitable higher level language.

It is worth noting that all of these programs are written in a subset of ALGOL-60. A course dedicated to the teaching of higher level computer languages could show the utility of such languages in no better way than to have its software packages written in such a language. One of the very useful demonstrations this project has made has been to show that complete, useful and efficient computer-based instruction systems can be written in a higher level language.

Preliminary and informal results from the students in the course are quite encouraging and tend to support the basic philosophy of this approach. There is every reason to believe that the future statistical analysis of the effects of this course will confirm these initial observations.
APPENDIX A

Sample lessons

(The following are short examples from the actual curriculum; they have been retyped. Comments within brackets are parenthetical comments added to indicate various features.)

3 JULY 1969
SLAKER (VERSION OF 28 MAY 69)
PLEASE TYPE YOUR NUMBER...→11
(CTRL G TO BEGIN-CTRL T TO RESTART) →

WHERE TO? → L68

Lesson 68: Using Tests

We can use ‘first,’ ‘bf,’ and so on with ‘call’
If you type this:
   CALL FIRST OF BF OF ‘BEARS Hibernate in winter’ ‘X’
   IF WORD? < X > THEN P < X >.
Then SLOGO finds that < X > is ‘H,’ which is a word,
so SLOGO replies:

H

For problems 1-6, type what SLOGO replies
Type ‘N’ if nothing is printed
1. CALL FIRST OF ‘BLUE SKIES’ ‘W’
   IF WORD? < W > THEN P < W >.
   → BLUE...CORRECT
2. CALL DIFF OF 9 AND 6 ‘X’.
   IF NUMBER? < X > THEN P TIMES OF 4 AND < X >.
Does SLOGO think 3 is a number? [a hint]
   → 12...CORRECT

Now sign on to SLOGO and do problems 7-10.
After all 4 problems are done, type control A.
7. TEST TO SEE IF ‘PLACE KICK’ IS A SENTENCE.
   If it is, print ‘IS SEN.’
8. TEST TO SEE IF 7 IS A WORD. IF IT IS, PRINT ‘IS WORD.’
9. TEST TO SEE IF ‘1 4 8’ IS A NUMBER. IF IT IS, PRINT THE NUMBER.
10. TEST TO SEE IF ‘P’ IS A WORD. IF IT IS, PRINT BF OF THE WORD.

WHERE TO? → SLOGO...OK
SLOGO...the original conjuring cat
> IF WORD? ‘P’ THEN P BF ‘P.’
= NIL
> TO REVERSE < A >.
   → IS < A > ‘.’
> IF YES RETURN ‘.’
> RETURN WORD LAST < A > AND REVERSE OF BUTLAST < A >.
> END.
> P REVERSE ‘1234567890.’
= 0987654321
> ...OK

From the collection of the Computer History Museum (www.computerhistory.org)
OUTPUT SHOULD BE:
7. IS SEN
8. NO OUTPUT
9. NO OUTPUT
10. NIL

LEsson 11: Problem Solving

Write a SIMPER program to solve each of these problems for you.

1. Mary bought 3 pounds of candy at 29 cents per pound. What was her bill?

WHERE TO? → SIMPER...OK
SIMPER (Version of 6 Jun 69)
BEGIN PROGRAMMING
LOC INSTRUCTION
000 → BEGIN
001 → GET X
002 → GET Y
003 → LOAD X
004 → MUL Y
005 → STOR X
006 → PUT X
007 → END
008 →
EXECUTE...STARTING LOC → 0 AND ENDING LOC → 7
PROGRAM EXECUTED ON 3 JULY 1969
  P C INSTR REG A REG B
000 BEGN 0 32768
INPUT → 3
INPUT → 29
  000 LOAD 3 32768
  004 MUL 87 32768
  005 STOR 87 32768
OUTPUT = 87
  007 END 87 32768
...END OF EXECUTION, CONTINUE...
008 → ...OK
HER BILL WAS 87 CENTS. IF YOUR PROGRAM SAID
OUTPUT = 87, SKIP ON.

2. A rectangle is 8 inches long and 4 inches wide. Find its area.
   TO FIND THE AREA OF A RECTANGLE, MULTIPLY THE LENGTH TIMES THE WIDTH.
WHERE TO? → SIMPER...OK
APPENDIX B

Concise guide to SLOGO

(Optional words are italic).

WORDS OF X AND Y
SENTENCE OF X AND Y
FIRST OF X
BUTFIRST OF X
LAST OF X
BUTLAST OF X
SUM OF X AND Y
DIFFERENCE OF X AND Y
TIMES OF X AND Y
QUOTIENT OF X AND Y
IS X Y

IF YES THEN S₁, when S₁ is some executable statement
IF NO THEN S₁
IF WORD? OF X THEN S₁
IF SENTENCE? OF X THEN S₁
IF NUMBER? OF X THEN S₁
TO NAME OF < X > AND < Y >
RETURN X
END

GO TO LINE N

CALL THING X NAME Y

LOGO
ERASE name
TRACE
UNTRACE
PRINT X

produces a word which is X concatenated with Y.
produces a sentence of Y appended to X.
if X is a word, result is the first letter; if X is a sentence, result is the first word.
if X is a word, result is all but the first letter; if X is a sentence, result is all but the first word.
if X is a word, result is the last character; if X is a sentence, result is the last word.
if X is a word, result is all but the last character; if X is a sentence, result is all but the last word.
X + Y
X - Y
X ∙ Y
X ÷ Y
sets internal flag to true if X = Y (equality of arguments for numbers; character by character equality of words; word by word equality of sentences); false otherwise.
execute S₁ if internal flag is true; ignore S₁ if false.
execute S₁ if internal flag is false; ignore S₁ if true.
executes S₁ if X is a word.
executes S₁ if X is a sentence.
executes S₁ if X is a number.
begins definition of a function named “name” and whose formal parameters are X and Y.
exit from current function with value X.
complete definition of function and insert RETURN ‘ ’ in the code for safety’s sake.
branching statement to be used inside of user-defined functions.
associates the name produced by evaluating Y with the value produced by evaluating X.
reset.
erase the function named “name.”
turn on trace for all user-defined functions.
turn off the trace.
print the value of X on the user’s teletype.