UC Office of the President
Computer Assisted Instruction (CAI)

Title

A Rationale and Description of a CAI Program to Teach the BASIC Programming Language

Permalink

https://escholarship.org/uc/item/3133k0f9

Authors

Atkinson, Richard C.
Barr, Avron
Beard, Marian

Publication Date

1975

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Instructional Science 4 (1975) 1–31
© Elsevier Scientific Publishing Company, Amsterdam – Printed in the Netherlands

A RATIONALE AND DESCRIPTION OF A CAI PROGRAM TO TEACH THE BASIC PROGRAMMING LANGUAGE*

AVRON BARR, MARIAN BEARD and RICHARD C. ATKINSON

Institute for Mathematical Studies in the Social Sciences, Stanford University, California

ABSTRACT

A BASIC Instructional Program is being developed as a vehicle for research in tutorial modes of computer-assisted instruction (CAI). Several design features will be appropriate to training in other technical areas and applicable in other instructional settings where the development of analytic and problem-solving skills is a goal.

Methods are incorporated for monitoring and aiding the student as he works on programming problems in the BASIC language. The instructional program developed can be used to investigate schemes for optimizing problem presentation and giving assistance during problem-solving based on a model of the student's abilities and difficulties. Previous experience in the instructional and technical aspects of teaching a programming language indicates that a course in computer programming can be designed to help the student acquire programming concepts in a personalized and efficient manner as he develops skills at increasingly advanced levels.

This article reports on work currently in progress and briefly summarizes observations and conclusions based on operation during the pilot year.

A major goal of the research project is to increase the sophistication with which the instructional program monitors the student's work and responds to it with appropriate hints and prompts. One aspect of such work is the utilization of algorithms for checking the correctness of a student procedure. Limited but sufficient program verification is possible through simulated execution of the program on test data stored with each problem. Within the controllable context of instruction, where the problems to be solved are predetermined and their solutions known, simulated execution of the student's program can effectively determine its closeness to a stored model solution.

The BASIC Instructional Program (BIP) is written in SAIL (VanLehn, 1973; Swinehart and Sproull, 1971), a versatile, ALGOL-like language, implemented exclusively at present on the DEC PDP–10 computer. SAIL includes a flexible associative sublanguage called LEAP (Feldman et al., 1972), which was used extensively to build BIP's information network. The course is now running on the PDP–10 TENEX timesharing system at the Institute for Mathematical Studies in the Social Sciences. It

*This research is funded by Personnel Training and Research Programs, Office of Naval Research. During these developmental months, we have received considerable cooperation from the staffs of the pilot institutions, notably Professor Carl Grame of DeAnza College and Dr. Paul Lorton, Jr. of the University of San Francisco.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

was offered during the pilot year as an introductory programming course at DeAnza College in Cupertino, California, and at the University of San Francisco in San Francisco, California. The collected data are being used to modify the problems and the "help" sequences in preparation for a more controlled experimental situation planned for the next academic year.

Overview of IMSSS Research in Tutorial CAI

The Institute for Mathematical Studies in the Social Sciences (IMSSS) at Stanford University has been involved in CAI projects in computer programming and in tutorial CAI in other technical areas since 1968. Work in teaching computer programming began with the development of a high-school level CAI course in machine language programming (Lorton and Slimick, 1969). The project, called SIMPER, taught programming via a simulated three-register machine with a variable instruction set. Later, lessons in the syntax of the BASIC language were added to the curriculum. Programming problems using BASIC were presented, but the student solved them by linking to a commercial BASIC interpreter, without receiving assistance or analysis of his efforts from the instructional program.

In 1970 the Institute developed a much larger CAI curriculum for a new course to teach the AID programming language at the introductory undergraduate level. This course has been used in colleges and junior colleges as a successful introduction to computer programming (Friend, 1973; Beard et al., 1973). However, because no information about the student's progress is passed between the instructional program and the AID interpreter, the course cannot provide individualized instruction during the problem-solving activity itself. After working through lesson segments on such topics as syntax and expressions, the student is assigned a problem to solve in AID. He must then leave the instructional program, call up a separate AID interpreter, perform the required programming task, and return to the instructional program with an answer. As he develops his program directly with AID, his only source of assistance are the minimally informative error messages provided by the interpreter.

In recent years, developments in interactive CAI and in artificial intelligence have enabled teaching programs to deal more effectively with the subject matter they purport to teach, in effect, to "know" their subject better. Generative CAI programs provide one example of this increased sophistication. The generative programs developed by Carbonell and others (Carbonell, 1970; Collins et al., 1973) employ a semantic

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

network interrelating a large factual data base. Instruction then takes the form of a dialogue in which the program can both (a) construct, present, and evaluate the answers to a multitude of questions, and (b) answer questions posed by the student. An interesting generative CAI program in digital logic and machine-language programming has been developed by Elliot Koffman at the University of Connecticut (Koffman and Blount, 1973). Another course in programming is being written by Jurg Nievergelt for the PLATO IV system at the University of Illinois (Nievergelt et al., 1973).

Two CAI courses developed at IMSSS are capable of dealing in a sophisticated way both with their subject matter and with the student. These courses provide instructive interaction throughout the problem-solving activity by performing operations specified by the student, evaluating the effect of the operations, and, on request, suggesting a next step in the solution.

The first of these, a CAI program for teaching elementary mathematical logic, is described in a report by Adele Goldberg (1973). An experimental version of the program employed a heuristic theorem-prover as a proof-analyzer to generate appropriate dialogue with students who needed help with a proof. The proof-analyzer can determine relevant hints when a student requires help in completing a solution, and when he can discover diverse solution paths. While the prover was limited, the heuristics it supplied were more natural than those that might be supplied by more powerful, resolution-based theorem-provers. A version of this program without a theorem-prover has been used successfully as a primary source of instruction in an introductory symbolic logic course at Stanford for the past three years.

A CAI course described in Kimball (1973) uses symbolic integration routines and an algebraic expression simplifier to assist students in learning introductory integration techniques. The program stresses development of student heuristics by performing most of the tedious computations (substitutions, integration by parts, and so on) for the student after he has completely specified the parameters. An attempt is made to estimate each student's knowledge of integration methods individually, in order to select problems dynamically. Furthermore, by incorporating student-generated solutions that are superior to the stored solutions (which occur more often than one might expect) the program is capable of "learning" as well as teaching.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

The BIP Course

The goal of a tutorial CAI program is to provide assistance as the student attempts to solve a problem. The program must contain a representation of the subject matter that is complex enough to allow the program to generate appropriate assistance at any stage of the student's solution attempt. Both the logic and the calculus courses approach this goal. However, computer programming is an activity fraught with human variability, and how an individual calls on his programming skills to write a program is not as clear as, for example, how he uses integration methods to transform an integral. Furthermore, the difficulty of describing and verifying program segments precludes the kinds of solution analysis performed by the logic and calculus courses. BIP contains a representation of information appropriate to the teaching of computer programming that allows the program to provide help to the student and to perform a limited but adequate analysis of the correctness of his program as a solution to the given problem. As a vehicle for research in instructional strategies, BIP will serve as both a teaching and a learning tool.

To the student seated at his terminal, BIP looks very much like a typical timesharing BASIC operating system. The BASIC interpreter, written especially for BIP, analyzes each program line after the student types it and notifies the student of syntax errors. When the student executes his program, it is checked for structural illegalities, and then, during runtime, execution errors are indicated. A file storage system, a calculator, and utility commands are available.

Residing above the simulated operating system is the "tutor", or instructional program. It overlooks the entire student/BIP dialogue and motivates the instructional interaction. In addition to selecting and presenting programming tasks to the student, the instructional program identifies the student's problem areas, suggests simpler subtasks, gives hints or model solutions when necessary, offers debugging aids and a facility for communicating with the Stanford staff, and supplies incidental instruction in the form of messages, interactive lessons, or, most often, manual references. Each student receives a BIP manual that introduces him to programming, the BIP system, and the syntax of BIP's version of BASIC. The manual serves as the student's initial source of information throughout the course. To the extent that the student receives off-line information from the manual, the BIP program is not entirely self-contained. The most powerful instructional features are those provided through on-line interaction, however; improving this tutorial interaction is the main focus of our research.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

At BIP's core is an information network that embodies the interrelations of the concepts, skills, problems, subproblems, prerequisites, BASIC commands, remedial lessons, hints, and manual references. We believe that with a sufficient student history, the network can be successfully applied to a student learning model to present an individualized problem sequence, to control the frequency and type of assistance given during programming, and to identify problem areas. Our experimental work will compare different student models and decision algorithms, including a "free" or "student-choice" mode where the student is given enough information for him to select his own problems.

Figure 1 illustrates schematically the interactions of the parts of the BIP program. Each of these is discussed in detail below.

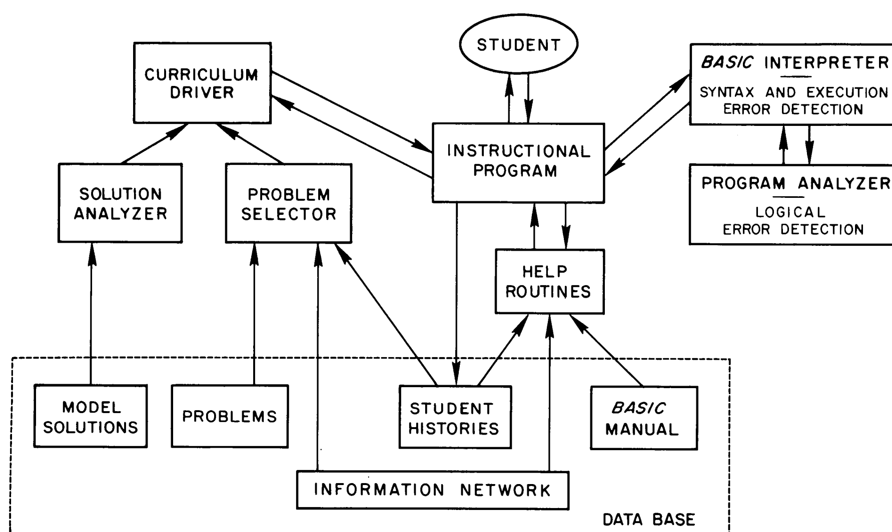

Figure 1. Information flow within BIP

The BASIC Interpreter, Error Detection, Assistance, Debugging Aids

BIP's interpreter was specially designed to allow the instructional program full access to the student's programs and his errors. It handles a complete subset of BASIC. During a student's work on a task, each of the BASIC operators can be temporarily de-activated as required for pedagogical purposes. For example, during a simple task whose instructions require the use of a "FOR . . . NEXT" loop and in which no other branching is necessary, "IF" statements will not be accepted. The text of the task, of course, explicitly instructs the student to use the "FOR . . .NEXT" structure to form his loop.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

Immediately after the student enters a line, syntax analysis is performed. If a syntax error is discovered, an error message ("illegal print list", "missing argument for INT") is sent to the student, the error number is retained by the instructional program for reference if the student requests more help, and the line is rejected.

If he does not understand the syntax mistake immediately, the student can request one of two types of assistance by beginning his next line with a question mark:

? An explanatory message stored for this syntax error is printed. Repeated requests summon different messages until they are exhausted.

?REF A manual reference covering the particular syntax involved in the error is printed for the student.

Once the student has entered a syntactically legal program, he can have it executed in one of three formats, two of which involve debugging aids. After his request, and before the actual execution, the student's program is checked for illegal program structure (e.g., a missing END statement, or illegally nested loops) by a routine we call ERR DOKTOR. If all is well, one of the three modes of program execution is initiated:

RUN The student's program is executed, as in standard BASIC implementations, in the order of its line numbers.

TRACE (A debugging option) As a line is executed, its number is printed. This allows direct observation of the execution sequence of such structures as loops and conditional branches. When an assignment statement, which initializes or changes the value of a variable, is executed, the variable and its new value are printed with the line number. The student can easily see the "internal" activity of the program, which would otherwise be visible to him only via insertion of extra PRINT statements displaying interim results.
By specifying inclusive line numbers, the student can TRACE a selected section of his program. This is useful when he is satisfied with other parts of the program and wishes to avoid the time-consuming process of tracing those parts.

FLOW (The second debugging aid) This option is available on CRT display terminals. The student's program is listed on the screen, and BIP waits for the student to press a key to execute each line. The number of the current line blinks, indicating the execution sequence clearly to the student. Each time a transfer

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

of execution occurs (altering the normal flow of the program) an arrow is drawn to the line to be executed next. In addition, the student may specify up to six variables whose values will be updated and displayed at the top of the screen throughout execution. Up to three lines of input and output are displayed at the bottom of the screen, scrolling up and disappearing as new lines replace them.

There are four ways in which any mode of execution can terminate. Normal termination follows execution of a "BASIC END "or" STOP" statement. The student is told that "execution terminated at line xxx." Alternatively, the student can abort execution by typing a control key; BIP responds with the message "execution aborted at line xxx." The third cause of termination is excessively long duration, which is at present determined on the basis of a count of the number of lines executed. A message indicating BIP's suspicion of an infinite loop is printed. (The student is always allowed the option of continuing execution for a specified number of statements after this point, since his program may simply be extremely repetitious. It was felt that limiting the student in his execution of a program would counteract the benefits of true hands-on experience, in which the novice programmer sees the evidence of his program's inefficiency immediately.)

Finally, runtime errors terminate execution. If an unassigned variable, illegal GOTO, or other error is discovered, an appropriate error message is printed, the error number is stored by the instructional program, and execution terminates. The student may then request the same types of assistance for execution errors discussed under syntax errors above.

Goals of the Curriculum

Prior experience with CAI in programming at the college level has convinced us that many students who wish to learn the fundamental principles and techniques of programming have limited mathematical backgrounds. More important, their confidence in their own abilities to confront problems involving numerical manipulation is low. The scope of the BIP curriculum, therefore, is restricted to teaching the most fundamental of programming skills and does not extend to material requiring mathematical sophistication.

The curriculum is designed to give the student practice and instruction in developing interactive programs in order to expose him to uses of

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

the computer with which he may well be unfamiliar. BIP guides the student in construction of programs that he can "show off." The emphasis is on programs that are engaging and entertaining, and that can be used by other people. As the student writes his programs, he keeps in mind a hypothetical user, a person who will use the student's program for his own purposes and to whom the performance of the program must be intelligible. The additional demands for clarity and organization forced by interactive programming, as well as the increased noticeability of bugs, are valuable, as are the added motivational effects.

Numerous texts were examined as possible sources for the necessary programming principles to be developed in an introductory course and for the problems that illustrate those principles. We incorporated ideas from general computer science textbooks (Forsythe et al., 1969), from the excellent notes for an introductory programming course that were oriented toward the ALGOL language but whose examples were easily generalized (Floyd, 1971), and from books and notes dealing specifically with BASIC (Albrecht et al., 1973; Coan, 1970; Kemeny and Kurtz, 1971; Noland 1969; Wiener, 1972; various publications of the People's Computer Company, Menlo Park, California.)

In addition, problem sets from Stanford University's introductory computer science courses were collected and examined.

In general, the curriculum provides useful, entertaining, and practical computer experience for students who are not necessarily mathematically oriented. It gives them the opportunity to develop programming skills while working on problems that are challenging but not intimidating, in which the difficulties stem from the demands of logical program organization rather than from the complexities of the prerequisite mathematics. The pilot year's curriculum text is listed in Appendix A.

The Curriculum Driver

The curriculum is organized as a set of discrete programming problems called tasks, whose text includes only the description of the problem, not lengthy descriptions of programming structures or explanations of syntax. There is no default ordering of the tasks; they are not numbered. The decisions involving a move from one task to another can be made only on the basis of the information about the tasks (skills involved, prerequisites required, subtasks available) stored in BIP's information network.

A student progresses through the curriculum by writing and running a program that solves the problem presented on his terminal. Virtually no

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

limitations are imposed on the amount of time he spends, the number of lines he writes in his program, the number of errors he is allowed to make, the number of times he chooses to execute the program, or the changes he makes within it. The task he is performing is stored on a stacklike structure, so that he may work on another task and return to the previous task automatically. All BIP commands (listed in Appendix B) are available to the student at all times. The following commands deal specifically with the curriculum driver:

TASK — A student may be logged in to the BIP program without being assigned a particular task. (Thus he may use the special features of the interpreter to help him write programs not related to any task.) The TASK command presents a specific problem for the student to solve; if he does not specify a particular task by name (from a printed list of task names and texts), BIP selects a task for him.

HINT — When a student experiences difficulty with a task, several levels of help are available. HINT retrieves problem-specific hints from a set stored in the network.

SUB — If, after pondering the available hints, a method of attack has still not occurred to the student, he can have the task broken into conceptually simpler subtasks. These are presented one at a time as tasks, while the main task is pushed onto the stack structure. When the student completes a subtask, BIP returns him automatically and explicitly to the larger problem.

DEMO — The student may request that the stored "model" solution be executed, as a demonstration of the interaction required by the task. He may use this option either to clarify the task itself before he writes his program, or to check his program as it takes shape, comparing its output with the DEMO.

ENOUGH — If he understands the demands of the larger program during his work on the subtask, he can type ENOUGH and return to the larger task from which he started. Outside of a subtask, typing ENOUGH terminates work on the current task without giving the student credit for having completed it; the same task may be presented to him at a later time.

MODEL — After exhausting all hints and subtasks available for a given task, and after having seen the DEMO, the student can ask BIP to suggest a model solution. The model stored for each task is intended to be easily understood, and correct, but it is not necessarily the shortest or most elegant solution.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

RESET Typing RESET clears the task stack of all the tasks on which he has been working, so the student can start fresh.

MORE When he feels that he has solved the problem, the student types MORE and BIP takes over, as described in the "Solution Analysis" Section.

The curriculum structure allows for a wide variety of student aptitudes and skills. Most of the curriculum-related options are designed with the less competent, less confident student in mind. A more independent student may simply ignore the options. Thus BIP gives all students the opportunity to determine their own individual challenge levels simply by making assistance available, though not inevitable.

BIP offers the student considerable flexibility in making task-related decisions. As explained above, he may ask for hints and subtasks to get started in solving the given problem, or he may ponder the problem on his own, using only the manual for additional information. He may request a different task by name, in the event that he wishes to work on it immediately, either completing the new task or not, as he chooses. On his return, BIP tells him the name of the again current task and allows him to have its text printed to remind him of the problem he is to solve. Taken together, the curriculum options allow for a range of student preferences and behaviors; this flexibility will be put to use in the experiments referred to earlier, comparing student-selected and BIP-determined curriculum decisions.

Solution Analysis

At present a student is not considered to have completed a problem if he has not executed his current program successfully. BIP "knows" at all times (a) whether an executable, syntactically legal program exists, (b) whether the student has executed that program, (c) whether execution errors have occurred, and (d) whether the student has made changes or additions since the last execution. The student's history will be updated to indicate successful completion of a task only if he has succeeded in an error-free execution of the most recent version of his program.

Error-free execution of a program is no guarantee that the program correctly solves the problem presented. Program analysis is an embryonic art, and BIP is not capable of "understanding" a student's programs in the fullest sense implied by current research in artificial intelligence. We are, however, investigating two promising approaches that are expected to provide sufficient solution analysis for pedagogical purposes, without

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

involving a full-scale application of program verification techniques. The results of the two analysis efforts should allow BIP to give the student an indication of (a) the kinds of test values that his program fails to handle properly, and (b) the kinds of programming structures that his program should have but does not.

The first analysis scheme has been implemented and will be evaluated and expanded during the 1974–75 academic year. It involves the simulated execution of the student's program on a set of test data, comparing its output to that of the stored solution. A preliminary dialogue establishes the variable names that the student has used for input variables, allowing the verification routine to assign test values as though the program were actually executed normally. The student's program is considered to be an acceptable solution if it produces as output all the values produced by simulated execution of the stored solution. Unless otherwise specified (as a parameter to the given task), the student's program is not considered unacceptable on the basis of "extra" output. Should the student's program fail to pass this comparison test, he is advised to run the DEMO in order to see more clearly the interactive requirements of the task. Future efforts will be directed toward a second analysis scheme that compares the internal (structural) representations of the student's program and the stored solution.

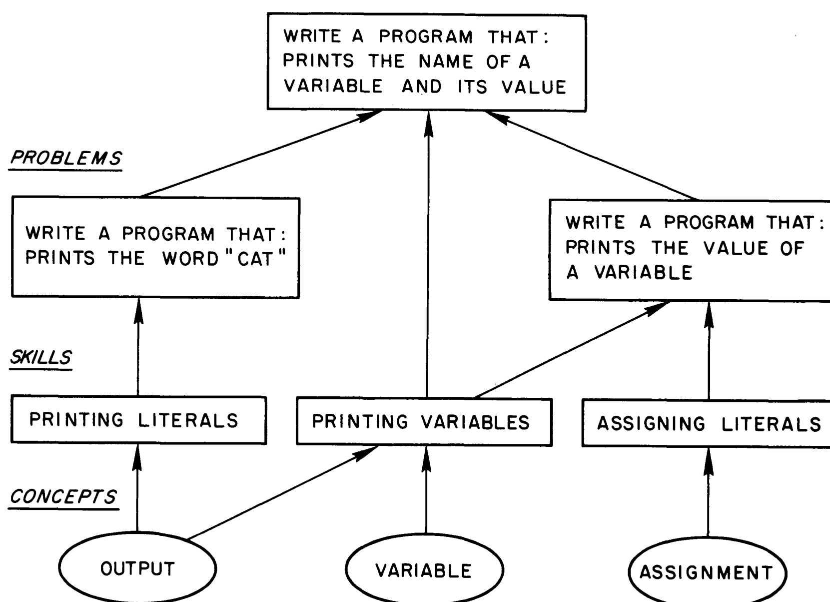

Figure 2. A segment of BIP's information network

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

BIP's Information Network

Task selection, remedial assistance, and problem area determination, BIP's "tutorial" activities, require that the program has a flexible information store interrelating the task, hints, manual references, and so on. This store has been built using the associative language LEAP (Feldman, 1972). The network is constructed using an ordered-triple data structure and is best described in terms of the various types of nodes:

TASKS — All curriculum elements exist as task nodes in the network. They can be linked to each other as subtasks, prerequisite tasks, or "must follow" tasks.

SKILLS — The skill nodes are intermediaries between the concept nodes and the task nodes (see Figure 2). Skills are very specific, e.g. "concatenating string variables" or "incrementing a counter variable." By evaluating success on the individual skills, the program estimates competence levels in the concept areas. In the network, skills are related to the tasks that require them and to the concepts that embody them.

CONCEPTS — The concept areas covered by BIP are, for the time being, the following:

- Interactive programs
- Variables and literals (numeric and string)
- Expressions, (algebraic, string, and Boolean)
- Input and output
- Program control – branching
- Repetition – loops
- Debugging
- Subroutines
- Arrays (one dimensional)

The specific implementation of concept nodes in the network is not completely determined, but the links will be to the skills and only through them to the tasks.

BASIC OPERATORS — Each BASIC operator (PRINT, LET, . . .) is a node in the network. The operators are linked to the tasks in two ways: first as elements that must be used in the solution of the problem, and second as those that must not be used in the solution. (These are temporarily disabled in the interpreter.) The existence of "required" and "disabled" operators is made known to the student only when necessary.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

He is told not to use a disabled operator, for the purposes of the current task, if he enters a line using that operator. He may run his program as much as he likes, even if one or more of the required elements are absent, but he may not progress out of the current task until he has included those operators that are required in the task.

HINTS — The hint nodes are linked to the tasks they may be helpful in. A single hint is occasionally associated with more than one task, as appropriate.

ERRORS — All discoverable syntax, structural, and execution errors exist as nodes in the network, and are linked to the relevant help messages, manual references and remedial lessons.

The network is established when the BIP program is initialized, immediately before its execution. All of the curriculum information (task texts, stored solutions, hints, skills, required operators, and so on) is read from a source file, and the associations that characterize each task are made at this time. This structure provides considerable flexibility in the curriculum and will be useful as we evaluate and experiment with the nature of the associations that make up the network.

Following a successful comparison of his program with the stored solution, the student is given a post-task interview in which BIP presents the model solution stored for that problem. (The student is encouraged to regard the model as only one of many possible solutions.) BIP asks the student whether he understands the solution, then asks, for each of the skills associated with the task, whether he needs more practice involving that skill. The responses are stored and used in future BIP-generated curriculum decisions. BIP then informs the student that he has completed the task, and either allows him to select his next task by name or selects it for him.

An example of the role of the Information Network in BIP's tutorial capabilities is the BIP-generated curriculum decisions mentioned above. By storing the student's evaluation of his own skills, and by comparing his solution attempts to the stored models, BIP can be said to "learn" about each student as an individual who has attained a certain level of competence in the skills associated with each task. BIP can then search the network to locate the skills that are appropriate to each student's different abilities and to present a task that incorporates those skills. The network provides the base from which BIP can generate decisions that take into account both the subject matter and the student, behaving

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

somewhat like a human tutor in presenting material that either corrects specific weaknesses or challenges and extends particular strengths, proceeding into as yet unencountered areas.

The BIP Manual

It is tedious and probably ineffective to present voluminous description, explanation, and examples from the computer directly on the terminal. We have chosen instead to present this material to the student in a printed manual of approximately 50 pages. The manual includes complete instructions on the operation of the course (signing on, dealing with the terminal, dealing with BIP), a general introduction to computers (their capabilities and the concepts involved in programming languages), and the syntax of BIP's BASIC, completed with examples and suggestions for the appropriate uses of each of the BASIC statements.

All programming terms used in the manual and in the tasks are defined briefly in the glossary at the end of the manual. References to the relevant sections of the manual are included in each glossary entry. All words that have precise programming meanings different from their normal English meanings are listed.

We believe that when the student encounters another programming language with which he is not familiar his primary resource will be the manual for that language. He is not likely to have an instructor or a CAI course at hand, and the principal means by which he will learn the new language will be through his own experimentation, guided by the explanations and examples in the manual. Experience with BIP (with its frequent cross-references to the manual) will, we hope, give the student a degree of confidence and ease in finding his way in other situations, when the manual may be his only guide.

Miscellaneous Options Available to the Student

Several additional features are available to BIP students:

CALC All literal BASIC expressions (numeric, string, and Boolean) can be evaluated by this BIP command. This is not only a convenience, freeing the student from having to write and run a complete program to make a simple calculation, but it is also useful as a debugging aid.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

FILE SYSTEM: FILES, SAVE, GET, MERGE, KILL

BIP allows each student to save permanently as many as four programs, with names he designates. This gives him the opportunity to work on an extended programming project and simultaneously to accumulate his work from each session at the terminal. He can obtain a listing of his file names, with their most recent write dates, and his saved programs are always immediately retrievable for modifications or additions.

FIX

This feature allows the student to send a message to the programmers at Stanford. It gives him a chance to communicate difficulties and confusions and helps both to improve BIP's interaction abilities and to identify and locate errors in the program. The convenience of typing a message or complaint while seated at the terminal encourages students to provide us with immediate and valuable feedback.

Experience during the Pilot Year

BIP has been in operation for a full school year, and reaction from the more than 200 students was generally favorable. As was expected, it was necessary for members of the project staff to be available to the students frequently in the early part of the year to clarify ambiguous problems and confusing features. After four months, however, much of this staff support was no longer required, and students at one school worked successfully with no consultation whatsoever.

The most frequent criticisms of the course early in the year were (1) that insufficient information was available on-line, and (2) that a student did not know how far he had progressed in the body of the curriculum. The first of these criticisms largely disappeared as BIP's tutorial abilities expanded, and the second will be dealt with through the addition of a "curriculum status" command.

Some students complained about insufficiencies in the curriculum, citing the lack both of remedial, easy tasks and of more challenging, really difficult problems. Since the spring of 1974, the curriculum has expanded considerably in both directions, and now offers tasks as simple as printing a given string and as difficult as changing numbers from one base to another. On-going efforts, involving analysis of the student-BIP dialogue, will lead to revisions both in the content of the curriculum and in the efficient specification of information relevant to the selection of appropriate tasks.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

We are in continuous communication with students who are using the course and whose suggestions regarding more flexible, intelligible interaction with BIP have generated several improvements. Past experience has shown that superficial problems in dealing with an instructional program can become significant barriers to acquiring the concepts and skills presented by the program, and we continue to make additions to BIP to eliminate frustrating confrontations between the student and the uncomprehending machine.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

APPENDIX A: THE BIP CURRICULUM

The following is the text for all tasks, hints, and subtasks in the pilot year curriculum. Some explanatory remarks are in order.

(1) The tasks appear in the order in which BIP would present them if it had no access to the student history. This order is modified in two ways: either by the student's choice of a particular task, or by BIP's decision based on the student's previous work.

(2) A MORT is a continuation of the original problem, calling for a modification or extension of the program just completed. Within this listing, the text of each task is followed by the hints and subtasks associated with it; the MORTs of the task are printed next, followed by their own hints and subtasks.

(3) Because some tasks require similar skills and strategies, some hints and subtasks are associated with more than one main task, and thus they appear more than once in this listing.

(4) References to Section Numbers refer to the BIP manual supplied to each student.

(5) Terms enclosed in asterisks (e.g., *print*) call attention to the special use of that term. All such terms are listed and explained in the glossary of the manual.

TASK PR1:

Before you start the first problem, be sure to read about the BIP course in the BIP manual.
Then read about the structure of BASIC programs.
Type "MORE" when you're ready.

MORT:

Now write a *program* to *print* the *number* 6 on your teletype. Then *run* the *program*.

TASK OP1:

SCRATCH your old program. Then write and *run* a *program* that *prints* the *sum* of 6 and 4.

MORT:

Now modify the program to do each of the following:
print the *difference*
print the *product*
print the *quotient*

HINT:

"Sum" means addition
"Difference" means subtraction
"Product" means multiplication
"Quotient" means division

TASK VN1:

SCRATCH your old program. then write a program that:
1. *Assigns* the *value* 6 to a *numeric variable* N.
2. *Prints* the value of this variable.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

TASK VX1:

Write a program that:
1. Assigns the value 6 to N.
2. Prints the sum of N and 4.

TASK VX2:

Write a program that:
1. Assigns the value 6 to M.
2. Assigns the value 4 to N.
3. Prints the sum, difference, product and quotient of M and N.

HINT:

"Sum" means addition
"Difference" means subtraction
"Product" means multiplication
"Quotient" means division

TASK IN1:

Write a program that:
1. Allows the user to *input* a value to M and a value to N.
2. Prints their sum, difference, product and quotient.

TASK IN2:

Write a program that:
1. Allows the user to choose the arithmetic operation he wants the program to perform. He should type 1 to add, 2 to subtract, 3 to multiply or 4 to divide. Use the variable X for this code number.
2. Allows him then to input the values for M and N.
3. Prints out the result of the operation he asked for when he gave a value to X. For example, if he typed 4, you should print the quotient of the numbers he gave for M and N.

SAVE this program when you get it to work. It will help you later.

HINT:

Read about **IF . . THEN** statements in Section III.11.

HINT:

Depending on the value of X, the program should do one of four things. Get X first, then get M and N. then use X to decide which **PRINT** statement to *branch* to.

SUB:

You need a program that can make decisions, then you can incorporate the arithmetic operations into it.
Translate the following into BASIC (it is definitely not BASIC now), and run it:

1. let the user type a number between 1 and 4.
2. if the number is 1, jump to 7

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

3. if the number is 2, jump to 9
4. if the number is 3, jump to 11
5. the number must be 4, so print "YOU TYPED A 4!"
6. jump to the end of the program
7. the number is 1, so print "YOU TYPED A 1!"
8. jump to the end
9. print "YOU TYPED A 2!"
10. jump to the end
11. print "YOU TYPED A 3!"
12. the end

Once this program works, type "MORE" and return to the main task.

MORT:

Now fix up the program so that it prints out questions and little messages that tell the user:
a) What to do (e.g. "TYPE 1 FOR ADDITION", . . .).
b) What the result represents (e.g. "THE SUM IS . . .").

MORT:

Modify the program once again so that it keeps *looping* back to the beginning until the user inputs a 0 for the operation code.

HINT:

You need two more statements:
an **IF . . . THEN** after the "INPUT X" that jumps to the end if X is zero,
A **GOTO** back to the line with the instructions.

TASK ST1:

Please read about *strings* before you get confused.
Write (and run) a program that prints the string "SCHOOL".

TASK VS1:

Assign the value "HORSE" to the *string variable* X$ and print the value of X$.

TASK SX1:

Allow the user to **INPUT** the value of the string variable X$ then print that value. (Your program will just "echo" what the user types, whether he types a number or a word.)

MORT:

Read about *concatenation* of strings.
Concatenate the word "OKAY" (or any word you like) to the user's input. Print the result.

TASK SX2:

Assign the string "DOG" to X$ and the string "HOUSE" to Y$. Print the *concatenation* of X$ and Y$..

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

HINT:

Concatenation is in Section III.6. Type the & character with the shift key and the 6 key.

MORT:

(Keep the same string values of X$ and Y$.)
Assign the *concatenation* of Y$ and X$ to the variable Z$. Print the value of Z$.

MORT:

(Still with the same values of X$ and Y$.)
"HOUSEDOG" should have a space between the words.
Concatenate a space between Y$ and X$ and print the results

HINT:

The literal "A" prints the letter A.
What character between quotes will print as a space?

TASK SX3:

Allow the user to input the values of X$ and Y$.
Concatenate the strings with a space between them and print the result.

TASK SX4:

Let the user make up a sentence.
1. Ask him how many words he wants to have in the sentence.
2. Let him input those words, one at a time.
3. After each input, concatenate a space and his latest word into a string variable. Use X$ for the input word, and use S$ to hold all the concatenations.
4. After you have looped around the specified number of times, print his sentence.

HINT:

make S$ equal to the string version of nothing, like this: S$ = "" outside the loop.
Inside the loop, use S$ to accumulate the sentence:
S$ = S$ & " " & X$

SUB:

A very important sub task:
Write a program with a little loop. The "work" of the loop is just to print the value of the loop's index.
When you run the program, it should look like it is counting from 1 to the top value. Use whatever top value you like.

SUB:

Very important:
Write a loop that prints the value of its index. Start the loop at 1, but let the user give the top value. You can add to this program, making the loop do some real work, and the work will then be done as many times as the user likes.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

TASK INT1:

Rewrite your calculator so that the user can type
"+" for addition
"–" for subtraction
"*" for multiplication
"/" for division
to tell the calculator which operation to perform. You may have *SAVED* your calculator program; if so, use GET to retrieve it.

HINT:

Type MODEL IN2 and copy what you need, then make the necessary additions to it.

SUB:

You need a program that can make decisions about strings, then you can incorporate the arithmetic operations into it. Write a program that asks the user to type any character. If he typed a ! mark, the program should say "YOU TYPED A !". If he typed something else, it should say "YOU DID NOT TYPE A !"

TASK XMAS:

On the first day of Christmas, someone's true love sent him/her a partridge in a pear tree (one gift). On the second day, the true love sent two turtle doves in addition to another partridge (three gifts on the second day). This continued through the 12th day, when the true love sent 12 lords, 11 ladies, 10 drummers, . . . all the way to yet another partridge. Write a program that computes and prints the total number of gifts sent on that 12th day.

HINT:

This program requires a loop. Each execution of the loop involves accumulating the value of the index into a total.

HINT:

Finding a total or sum almost always means two things:
1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.

In words, total equals total plus another value.

SUB:

A very important sub task:
Write a program with a little loop. The "work" of the loop is just to print the value of the loop's index.
When you run the program, it should look like it is counting from 1 to the top value. Use whatever top value you like.

MORT:

Modify your program so that it prints the total gifts for each day. (Day 1 = 1 gift, Day 2 = 3 gifts, Day 3 = 6 gifts, etc.)

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

HINT:

You need one statement that prints the value of the index (the number of days) and the accumulated total of gifts.

MORT:

The user of your program has a true love who will send presents in the same way for as many days as the user wants. Let your user say how many days, and calculate the number of gifts sent on that day. (The generous true love may send presents for more than 12 days, if the user likes.)

SUB:

Very important:
Write a loop that prints the value of its index. Start the loop at 1, but let the user give the top value. You can add to this program, making the loop do some real work, and the work will then be done as many times as the user likes.

TASK PAY:

A man is paid 1 cent the first day he works, 2 cents the second day, 4 cents the third, 8 cents the fourth, etc. (doubling his wage each new day). Calculate his wage for the 30th day.

HINT:

Say W is the variable for the wage. On the first day, W equals 1. For every day after that, W equals W * 2.

MORT:

Modify the program to calculate the total wages for the month: sum of the first day plus the second day
. . . plus the 30th day.

HINT:

You have a variable for each day's wage. You need another variable to accumulate the total.

HINT:

Finding a total or sum almost always means two things:
1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.
In words, total equals total plus another value.

MORT:

Your program's user has a contract with this man, for the same schedule of wages. Tell the user how much he will owe the man for any number of days he (the user) specifies.

SUB:

Very important:
Write a loop that prints the value of its index. Start the loop at 1, but let

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

the user give the top value. You can add to this program, making the loop do some real work, and the work will then be done as many times as the user likes.

TASK IT1:

Write a program that counts (and prints) the number of odd numbers between 5 and 187 inclusive. For example, there are 3 odd numbers between 5 and 9 inclusive: they are 5, 7, and 9. And a program that counted those numbers would print something like this:

THERE ARE 3 ODD NUMBERS BETWEEN 5 AND 9

Do not print each odd number as you count it.

HINT:

Any odd number plus 2 equals the next odd number.

HINT:

You know the bottom and top values of the loop, but the point of the program is to see how many times the loop must be executed before it gets to the top. Use a counter inside the loop and add to it with each execution.

MORT:

Now find the sum of all those odd numbers you just counted.

HINT:

Finding a total or sum almost always means two things:

1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.

In words, total equals total plus another value.

MORT:

Let the user specify a range, and tell him 1) how many odd numbers are in that range, and 2) the sum of those numbers. For example, you ask him for the lower limit (suppose he gives 9). Then you ask him for the upper limit (suppose he gives 17). The number of odd numbers in that range is 5 (9, 11, 13, 15, 17), and the sum is 65.

HINT:

The top and bottom values for the loop come from the user. The work of the loop is to count how many times it is executed, and to add up all the successive values.

TASK IT2:

Find the number of integers greater than 99 and less than 278 that are divisible by 11. You don't need any division to do this.

HINT:

You know the bottom and top values of the loop, but the point of the program is to see how many times the loop must be executed before it gets to the top. Use a counter inside the loop and add to it with each execution.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

MORT:

Now find the sum of the numbers greater than 99 and less than 278 that are divisible by 11.

HINT:

Finds a total or sum almost always means two things:
1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.

In words, total equals total plus another value.

TASK AV:

Find the average of 10 numbers. Ask the user to give the numbers, one at a time.

HINT:

Finding a total or sum almost always means two things:
1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.

In words, total equals total plus another value.

HINT:

The average of 10 numbers is their sum divided by 10.

SUB:

A very important sub task:
Write a program with a little loop. The "work" of the loop is just to print the value of the loop's index. When you run the program, it should look like it is counting from 1 to the top value. Use whatever top value you like.

MORT:

Modify the program to let the user specify how many numbers he wants to average. Let him type that many numbers one at a time, then tell him the average.

HINT:

The average of N numbers is their sum divided by N.

SUB:

Very important:
Write a loop that prints the value of its index. Start the loop at 1, but let the user give the top value. You can add to this program, making the loop do some real work, and the work will then be done as many times as the user likes.

TASK GAS:

Write a program to calculate the user's gas mileage. He recorded his car's mileage at the beginning of the trip, and again at the end of the trip, when he bought some amount of gas. Ask him for the starting and ending mileages (and calculate the miles driven), then ask for the number of gallons of gas he bought. Then tell him his gas mileage (miles per gallon).

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

Example: starting mileage = 5325
ending mileage = 5550
(miles driven = 5550–5325 = 225)
gallons of gas = 9
gas mileage = 225 miles / 9 gallons = 25 mpg.

MORT:

Each time the user buys gas, he records the mileage and the gallons bought. Modify your program to ask him how many times he bought gas; then ask for the mileage and gallons he recorded each time. Accumulate the total miles traveled and the total gallons, then print those totals and the gas mileage. Test the program with some very simple numbers to be sure that it calculates correctly.

HINT:

You only need the starting mileage once. Totals miles equals the last mileage recorded minus starting mileage.
Keep a running total of gallons bought.

TASK GUESS:

Write a program that plays a guessing game. Generate a random integer between 1 and 25 (read the manual first), then let the user guess what the number is. Print appropriate messages if his guess is too high or too low, and give him another chance to guess. Congratulate him for guessing correctly.

HINT:

Break this problem into parts. You need a loop whose "work" is to get and compare the user's guess. Generate the random number before the loop, and print the correct-guess message after the loop.

SUB:

Forget about random numbers for now. Write a program that gets a number from the user and compares his number to 100. Print "HIGHER THAN 100!" or "LOWER THAN 100!" or "100 EXACTLY!" appropriately. Then you can put this part together with the other parts you need in the main task.

SUB:

Your program must get a number from the user again and again, until the input number equals some set value (the random number). For now, write a program that asks for a number and checks to see if that number equals 100. If it is 100, the program should stop; if not, it should ask for another input. Then you can fit this part into the main task.

MORT:

Add a feature to your program that tells the user how many guesses he needed. Three lines will do it: one to assign the value 0 to a counter variable, one to add to the counter each time he guesses, and one to print the value of the counter with some appropriate message.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

MORT:

Add another feature that lets the user start the game again with a new random integer. Print an instruction like "TYPE 'YES' IF YOU WANT TO PLAY AGAIN." If he types 'YES' then start the game over; Otherwise, let the program stop.

TASK TWOS:

Write a program using a **FOR. .NEXT** loop to count by twos, up to a number typed by the user. If he types
8, your program should print
2
4
6
8

TASK BACK:

Use a **FOR. .NEXT** loop to count backwards from 20 to 0, by twos. You will need a STEP –2 in your 'FOR' statement.

TASK NGREAT:

Ask the user to type two numbers, then compare them. If the user types 4 and 12.5, for example, your program should print

12.5 IS GREATER THAN 4

TASK ALPH:

Compare two strings typed by the user. A string is "less than" another string if it comes before the other string alphabetically: "APPLE" < "FISH" is true.
Your program should print something like

APPLE COMES BEFORE FISH

TASK LLOOP:

Use a loop to get three numbers from the user, and print the largest of those numbers. Do not use three variables for the numbers. Hint: set a variable L (for largest) equal to 0. Then compare each user number with L. Change the value of L to a larger number if one is typed.

HINT:

Set a variable L (for largest) equal to zero. Then compare each user number with L. Change the value of L to a larger number if one is typed.

TASK SLIST:

Let the user input a *list* of 4 strings (a *subscripted variable* with 4 "slots" in it) – for example, the names of the courses he is taking. Print out the list after it is all typed in. Use a **FOR. . NEXT** loop in this program.

HINT:

There are two parts to this:
Looping to input a string list, and looping to print it out.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

SUB:

Think about a number list for now. The key is to use the index of the loop as the index of the list. Write a loop whose index starts at 1 and goes to 4. The work of the loop is to assign the value of the index to the corresponding element of the list:

L (I) = I

The only way to test your program is to use another loop, indexed from 1 to 4, whose work is to print the list, one element at a time:

PRINT L (I)

The first execution of the loop should print the first element of the list, etc. When you finish this subtask, return to the main task. Change the list variable to a string list variable, and change the work of the first loop so that each execution asks the user to input a string.

TASK BACKLST:

Take a list of strings from the user, then print the list in the opposite order. The list may be of any length up to 25 (ask how long the user wants it to be, then set up a loop whose top value is that number.) You will need a **FOR . . NEXT** loop with a STEP −1 to print the list backwards.

SUB:

Very important:
Write a loop that prints the value of its index. Start the loop at 1, but let the user give the top value. You can add to this program, making the loop do some real work, and the work will then be done as many times as the user likes.

TASK OTHER:

Take a list of numbers from the user, of any length he likes up to 15. After he types the numbers, print out every other number in his list. (If he types these 6 numbers: 2 8 12 5 3 9 your program should print the 2, 12, and 3.)

HINT:

Use a **FOR . . NEXT** loop with STEP 2. Then use the index of the loop as the index of the list to get every other element in the list.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

APPENDIX B: THE BIP COMMANDS

This is an alphabetic listing of the BIP commands and their functions. Many (e.g., RUN, LIST, SAVE) are identical in function to their standard BASIC counterparts. The others serve specifically instructional purposes, in that they deal with BIP's curriculum structure, file system, or student history.

CALC — Evaluates an expression. This feature allows the student to see the result of quick calculations without writing and running a complete program.

CURRIC — Writes the text of the curriculum to a disk file. This is available to Stanford programmers and designated course instructors only. CURRIC provides a readable version of the curriculum-related parts of the network, with the text of the tasks listed along with the associated hints and subtasks. This listing appears as Appendix A.

DEMO — Executes the stored model solution as a demonstration of the requirements of the task.

ENOUGH — Terminates the current task without giving the student credit for having completed it.

FILES — Lists the names of the files in permanent storage with their last write dates.

FIX — Allows the student to leave a message for Stanford.

FLOW — Allows the student to step through the execution of his program, graphically indicating the sequence of execution by blinking the number of the current line and drawing arrows to show a transfer of control.

GET (name) — Retrieves the named program from permanent storage. The retrieved program replaces the current program (if any) in the student's core space.

HINT — Prints a hint, if any remain. Some tasks have more than one hint associated with them in the network; a few have no hints. When a student asks for a hint, BIP internally flags the hint that it supplies. Another request for a hint, during work on the same task, initiates a search for an associated hint not yet flagged.

KILL (name) — Erases the named program from permanent storage. Students cannot affect each other's file storage, so indiscriminate use of this command can inconvenience only the KILLer himself.

LIST — Prints the current program in the order of its line numbers. Students are encouraged to LIST often, in order to avoid confusion between what was intended and what actually exists in the program.

MERGE (name) — Retrieves the named program from permanent storage and adds it to the current program. Unlike GET, MERGE does not erase the current program before retrieval. MERGE allows the student to

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

develop larger programs, a section at a time, testing and saving separate pieces of the program as he goes. BIP informs him of instances in which a line from permanent storage replaces or duplicates the current line (i.e., where the two programs have one or more identically-numbered lines).

MODEL — Prints a typical solution to the current task, only after all available hints and subtasks have been presented, and after the student has seen the DEMO.

MORE — Continues the presentation of a task. If all parts of the task have been completed, the post task interview is presented. Some tasks require that the student complete two or three closely related problems, calling for a modification or expansion of the original program. These "must-follow" tasks are referred to as MORTs, both internally in BIP and in the curriculum listing given in Appendix A. The MORE routine will not allow a student to advance, either to a MORT or to a new task, unless he has successfully run his current program.

REPORT — Provides Stanford programmers and designated course instructors with a summary of student activity, either by school (currently DeAnza or the University of San Francisco) or for all students using BIP. The report shows student number, name, number of sessions and total hours accumulated on the course, and number of tasks completed.

RES — Terminates all currently entered tasks, without giving the student credit for completing them. This option allows him to extricate himself from a nest of tasks, should the need arise.

RUN — Executes the current program.

SAVE (name) — Stores the current program for future use. Saving the program in permanent storage does not affect the current version in any way.

SCR — Erases the current program.

SIMPER — Allows the BIP student to use a simulated three-register machine described in Lorton and Slimick (1969). The SIMPER option allows instructors to demonstrate the differences between BASIC and a machine language by assigning problems to be solved with both.

SUB — Presents a subtask – a smaller part needed to complete the current task at the student's request. Upon completion of a subtask, BIP returns the student automatically and explicitly to the larger task.

TASK (name) — Presents the student's next programming task. He may request a task of his choice by supplying its name; otherwise, BIP selects the next task on the basis of the student's history on previous tasks.

TRACE — Executes a program, but prints out line numbers and variables as execution progresses.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

WHAT	Gives the name of the current task and (optionally) prints the problem text again. The student may request the text of a different task by supplying its name.
WHEN	Prints the current date and time.
WHO	Prints the name of the student signed on to the terminal. This option was included because of past experience with groups of students sharing a small number of terminals, and is intended to prevent the inadvertent termination of unfinished sessions.

References

Albrecht, R. L., Finkel, L., and Brown, J. R. (1973) *BASIC.* New York: Wiley.

Beard, M. H., Lorton, P., Jr., Searle, B. W., and Atkinson, R. C. (1973) *Comparison of student performance and attitude under three lesson selection strategies in computer-assisted instruction,* (Technical Report No. 222) Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University.

Carbonell, J. R. (1970) "AI in CAI: An artificial intelligence approach to computer-assisted instruction," *IEEE Transactions on Man-Machine Systems,* MMS–11, 190–202.

Collins, A. M., Carbonell, J. R., and Warnock, E. H. (1973) *Analysis and synthesis of tutorial dialogues.* (Technical Report No. 2631) Cambridge, Mass.: Bolt, Beranek and Newman.

Coan, J. S. (1970) *BASIC.* New York: Hayden Books.

Feldman, J. A., Low, J. R., Swinehart, D. C., and Taylor, R. H. (1972). "Recent developments in SAIL." *AFIPS Fall Joint Conference Proceedings,* 1193–1202.

Floyd, R. W. (1971). *Notes on Programming and the ALGOL W Language.* Calif.: Computer Science Department, Stanford University.

Forsythe, A. I., Keenan, T. A., Organick, E. I., and Sternberg, W. (1969). *Computer Science: A First Course.* New York: Wiley.

Friend, J. (1973) *Computer-assisted Instruction in Programming: A Curriculum Description.* (Technical Report No. 211). Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University.

Goldberg, A. (1973) *Computer-assisted Instruction: The Application of Theorem-proving to Adaptive Response Analysis.* (Technical Report No. 203) Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University.

Kemeny, J. G. and Kurtz, T. E. (1971) *BASIC Programming.* (2nd ed) New York: Wiley.

Kimball, R. B. (1973). *Self-optimizing Computer-assisted Tutoring: Theory and Practice.* (Technical Report No. 206) Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University.

Koffman, E. B. and Blount, S. (1973). *A Modular System for Generative CAI in Machine Language Programming.* Storrs, Conn.: University of Connecticut, School of Engineering.

Lorton, P., Jr. and Slimick, J. (1969). "Computer based instruction in computer programming – a symbol manipulation-list processing approach." *Proceedings of the Fall Joint Computer Conference,* 535–544.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions

Manna, Z. (1973). "Program schemas" In A. V. Aho (Ed.), *Currents in the Theory of Computing,* Englewood Cliffs, N.J.: Prentice Hall.

Nievergelt, J., Reingold, E. M., and Wilcox, T. R. (1973). "The automation of introductory computer science courses." *Proceedings of the International Computing Symposium.*

Nolan, R. L. (1969). *Introduction to Computing through the BASIC Language.* New York: Holt, Rinehart and Winston.

People's Computer Company Newsletter, Box 310, Menlo Park, Calif.

Swinehart, D. C., and Sproull, R. F. (1971) Stanford Artificial Intelligence Laboratory Operating Note 57.2, Stanford University.

VanLehn, K., (1973). *SAIL User Manual,* Stanford, Calif: Stanford Artificial Intelligence Laboratory, Stanford University.

Wiener, H., and Ross, B. (1972). *BASIC Workbook.* Berkeley, Calif.: Lawrence Hall of Science, University of California.

This content downloaded from 132.239.239.184 on Tue, 4 Mar 2014 18:12:37 PM
All use subject to JSTOR Terms and Conditions