*Int. J. Man-Machine Studies* (1976) **8**, 567–596

# The computer as a tutorial laboratory: the Stanford BIP project

AVRON BARR, MARIAN BEARD AND RICHARD C. ATKINSON
*Institute for Mathematical Studies in the Social Sciences, Stanford University, Stanford, California 94305, U.S.A.*

*(Received 18 February 1976 and in revised form 5 June 1976)*

The BASIC Instructional Program (BIP) was developed to investigate tutorial modes of interaction in computer-assisted instruction. BIP is an interactive problem-solving laboratory that offers tutorial assistance to students solving introductory programming problems in the BASIC language. This paper describes how the problem presentation sequence is individualized based on a representation of the structure of the curriculum and a model of the student's state of knowledge. The nature of the student-BIP interaction is captured in an annotated student dialogue illustrating a typical session.

## 1. Background

Computers are now used in a wide variety of applications in education and training, including curriculum presentation and drill, information retrieval, and simulation of complex systems. The research reported here deals with an additional application: the use of the computer as a problem-solving laboratory. In the computer-based laboratory environment, the student attempts to solve problems on-line with the guidance of the instructional system. The system plays the role of interactive tutor, giving hints, correcting errors and evaluating progress. The full power of the computer as calculator and simulator is available to the student, providing the motivational effects of learning by working on real problems with adequate supervision at the student's convenience and at his own pace. The main focus of our work in the Complex Instructional Strategies research group at the Institute for Mathematical Studies in the Social Sciences at Stanford University is the individualization of the sequence of instruction presented in computer-assisted instruction (CAI). This paper describes a computer-based programming laboratory, which we have used in our research and which we believe is an excellent example of an effective CAI program.

The computer-based tutor design has been arrived at by several research groups using different approaches to Artificial Intelligence applications in CAI. Carbonell, Collins, and others (Carbonell & Collins, 1973; Collins, Passafiume, Gould & Carbonell, 1973) developed the GEO-SCHOLAR system to illustrate natural language inquiry of a large data base. However, the GEO-SCHOLAR system is really an elaborate tutor with sophisticated decision algorithms directing "mixed-initiative" dialogues: the instructional system can ask questions as well as answer them. Their recent work explores tutorial schemes for instruction in more procedural subject domains (e.g. an on-line editing system) where simply asking and answering questions is insufficient (Grignetti, Gould, Hausmann, Bell, Harris & Passafiume, 1974).

Danielson & Nievergelt's (1975) work at the University of Illinois PLATO system concentrates on automated problem-solving assistance. They use a top-down problem

solution graph to direct a tutorial dialogue about how to solve a programming problem. Although their system does not build a model of the student from which future instructional decisions could be made, the problem solution graph scheme leads directly to a useful representation of the curriculum. The student model could be updated as the student traverses the graph in his attempt to find a solution; this is a procedure we will incorporate in our work on BIP's REP subsystem (described in section 4) in the coming year.

Perhaps the most impressive and "knowledgeable" computer-based tutor yet devised is Brown's SOPHIE system (Brown, Burton & Bell, 1974) which grew out of research on modes of querying a simulation-based knowledge representation. Although curriculum guidance decisions are minimized (the system teaches only one skill, troubleshooting a complicated electronic circuit), SOPHIE's knowledge of troubleshooting strategy and logical deductions from known measurements fosters "learning by imitation" in a natural and exciting environment, the essence of tutorial style.

Research at IMSSS has approached the computer tutor model by successive refinement of more traditional approaches to CAI in logic and computer programming. The logic and more advanced set theory courses now running in fully tutorial mode were first conceived of as applications of automated theorem proving techniques for checking students' proofs (Goldberg, 1973). Current work in the advanced set theory course that we offer at Stanford involves informal, natural language student–machine dialogues using computer-generated audio to discuss, develop and refine complex mathematical proofs in an informal manner (Smith, Graves, Blaine & Marinov, 1975; Sanders, Benbassat & Smith, 1976).

In 1970 the Institute's Complex Instructional Strategies group developed a large CAI curriculum for a new course to teach the AID programming language (1968) 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, Lorton, Searle & Atkinson, 1973). However, it is a linear "frame-oriented" CAI program and cannot provide individualized instruction during the problem-solving activity itself. After working through lesson segments on such topics as variables, output, 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 is the minimally informative error messages provided by the interpreter.

Furthermore, the AID course was found to be an inadequate vehicle for our investigations of individualization of instruction because of the linear organization of its curriculum. The course consists of a large set of ordered lessons, reviews, and tests, and a student's progress from one segment to the next was determined by his score on the previous segment. A high score would lead to an "extra credit" lesson on the same concepts, while a low score would be followed by a review lesson. It became clear that this decision scheme, based on total lesson scores, was reasonably effective in providing instruction and programming practice, but since it dealt with rather large segments of the curriculum, the individualization of the course of study was minimal. All students covered more or less the same concepts in the same order, with slight differences in the amount of review. We were interested in developing a system whose decisions would be based on a more specifically defined goal: the mastery of particular

programming skills rather than achievement of a criterion lesson score. For this reason, we undertook development of a course with a new and different instructional design, based in part on earlier work by Lorton & Slimick (1969).

The BASIC Instructional Program (BIP) is a stand-alone, fully self-contained course in BASIC programming at the high school/college level. BIP was conceived and designed as a short course in programming for people who desired an informal but useful hands-on introduction to the use of the computer in problem solving. The programming language (BASIC) and the curriculum chosen for the course are appropriate for students who have no previous experience with computers, and who wish to become familiar with them, but do not intend to become serious computer programmers. For example, BIP has been used effectively both as a supplement to a "computer literacy" course and as a requirement in a business administration curriculum. This is not to say, however, that the ideas that have gone into the BIP system would be lost in a formal course for future programmers. A similar interative programming laboratory using a more appropriate language such as ALGOL or PASCAL and a modified curriculum could be a major addition to a first undergraduate course on structured programming.

Over 300 undergraduates at DeAnza College, the University of San Francisco, and Stanford have taken the course and contributed to its design. BIP's major features are as follows.

(a) A monitored BASIC interpreter, written in SAIL (Van Lehn, 1973) by the IMSSS staff, which allows the instructional system maximal knowledge about student errors.
(b) A curriculum consisting of approximately 100 programming problems at widely varying levels of difficulty.
(c) A HINT system, which gives both graphic and textual aid in problem solving.
(d) Individualized task selection based on a Curriculum Information Network, which describes the problems in terms of fundamental skills. Problems are selected using a model of the student's acquisition of the skills required by his earlier programming problems.
(e) A complete Student Manual that includes a general introduction to programming and detailed reference information on BASIC statements and structures.

Figure 1 is a schematic representation of the tutorial programming laboratory environment supported by BIP, described fully by Barr, Beard & Atkinson (1975). Section 4 presents a brief description of the system including some new features not described in the earlier report. Section 5 is an annotated dialogue illustrating the system's features and the student-BIP interaction.

The new work reported here is primarily concerned with BIP's optimized problem selection by means of an internal representation of the curriculum structure, the Curriculum Information Network (CIN). The use of network models to describe curriculum structure is an important development in tutorial CAI. The CIN enables the instructional program to "know" the subject matter it purports to teach, and meaningfully to model the student's progress along the lines of his developing skills, instead of the curriculum elements (problems) themselves. The next section discusses the current state of curriculum design for CAI courseware, the sources and context from which the CIN concept emerged, and our implementation of BIP's CIN. Section 3 discusses our use of the network to optimize task selection, and describes the algorithms currently in use. Section

6 describes an experiment comparing these algorithms with a fixed curriculum path for their effects on student performance. Section 7 outlines some of our future plans.

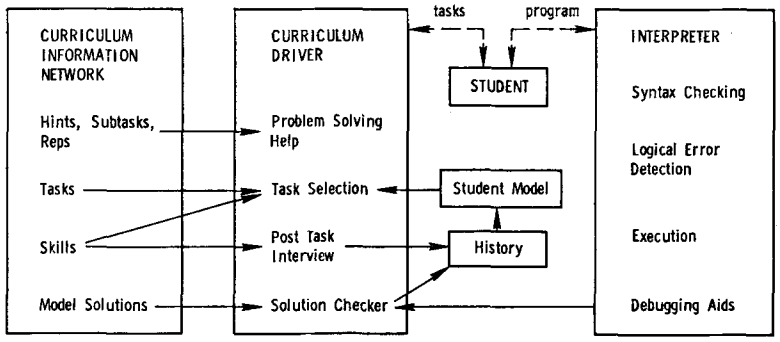


FIG. 1. A schematic representation of the tutorial programming laboratory.

## 2. Curriculum representation in CAI

In much of the current research in tutorial CAI, generative CAI, and mixed-initiative natural language dialogues, the central problem is the "representation" of the subject domain, which is also a fundamental concern of research in cognitive psychology and artificial intelligence. The goal is to provide a representation of the subject matter that is sufficient for individualized tutoring and also has a realistic and manageable computer implementation. A consideration of the different "representational poles" in vogue in CAI will give a perspective on the capabilities of the Curriculum Information Network representation.

The most common style of CAI courseware now being written consists of an automated presentation of a curriculum written by a human author. The author, knowledgeable in the subject matter, has in mind a clear organization of the interrelations among the specific "facts" of that subject, an implicit understanding of the dependency of one concept on another, and a plan for the development of new skills. His personal organization of the discrete elements results in a structured curriculum, consisting of lessons or problems presented in a sequence he considers to be optimal in some sense for his model of his students. This structure is like that of a textbook, established in advance of interaction with the student, but superior to a textbook in that the author builds branching decisions into the program, providing some degree of individualization. His subdivisions of the curriculum and the branching criteria he specifies constitute the author's representation of the subject matter in this traditional CAI style.

At the opposite pole of explicit curriculum structure are "generative" CAI programs, which do not use an author-written curriculum at all. This type of course generates problem statements and solutions by retrieving information from a complete, internal representation of the facts in the subject domain, usually stored in a semantic network. Question-and-answer construction algorithms are used to present the material in the data base to the student. These algorithms also embody heuristics for what to teach when, depending on some model of the student's state of knowledge. All the "facts", dependencies, and logical interrelations that form the author's knowledge of the subject must be embodied within the generative program. Thus, Carbonell's well-known SCHOLAR program (Carbonell, 1970) "knows" the names and populations of the

major cities of Brazil, and is clever enough to answer "What is the largest city in Brazil?" without "knowing" that fact explicitly. (See also Koffman & Blount (1975), for a review of generative CAI and an example of a generative computer based course in programming.) The advantages for individualization of generative CAI over fixed-branching courseware are considerable: the generative program is specifically designed to provide instruction and/or information in precisely the areas needed by the student. All decisions about what material to present can be made dynamically, based on the student's progress rather than on a predetermined sequence of material. Ideally, the program has access to the same information that makes the human author a subject matter expert, and this information can be made available to the student much more flexibly than is possible in author-generated CAI. In particular, the model of the student's state of knowledge is based on the structure of the subject itself (e.g. the student has covered the material on rivers in Brazil) rather than on the structure of the author's curriculum design as reflected in his branching specifications, which are typically triggered by correct/wrong response counters.

In a very simply structured question-and-answer curriculum, a counter-based decision policy can adequately reflect student progress. For instance, if the program asks questions about rivers in Brazil until the student answers two correctly in a row, then there is indeed some confidence about the student's knowledge of that subject. This is exactly the type of course material that can be program-generated by current methods; unfortutunately, both the simple question-and-answer and the program-generated approaches yield interactions that tend to be quite dry and unmotivating. The principal advantage of author-generated courses is that they can be well written; the author's organization of the material and style of writing can be powerful motivating factors.

### 2.1. THE CURRICULUM INFORMATION NETWORK

In technical subjects, development of skills requires the integration of facts, not just their memorization. The organization of instructional material is crucial for effective instruction in these areas. As the curriculum becomes more complex, involving the interrelations of many facts, the author's ability to present it in a format that facilitates assimilation and integration becomes more important. At the same time, using counters to keep track of the student's progress through the curriculum provides a less adequate model of his acquisition of knowledge.

The Curriculum Information Network is intended to provide the instructional program with an explicit knowledge of the structure of an author-written curriculum. It contains the interrelations between the problems which the author would have used implicitly in determining his "branching" schemes. It allows meaningful modelling of the student's progress along the lines of his developing skills, not just his history of right and wrong responses, without sacrificing the motivational advantages of human organization of the curriculum material. For example, in the BIP course, the CIN consists of a complete description of each of 100 programming problems in terms of the skills developed in solving the problems. Thus the instructional program can monitor the student's progress on these skills, and choose the next task with an appropriate group of new skills. The CIN introduces an intermediate step between recording the student's history and selecting his next problem: the network becomes a model of the student's state of knowledge, since it has an estimate of his ability in the relevant skills, not just his performance on the problems he has completed. Branching decisions are based on this model instead of

being determined simply by the student's success–failure history on the problems he has completed.

In this way, a problem can be presented for different purposes to students with different histories. The flexibility of the curriculum is, of course, increased as a result. More importantly, the individual problems in the curriculum can be more natural and meaningful; they do not necessarily involve only one skill or technique. In frame-type presentations the one-dimensionality of the problems has a constricting effect. In essence, the network as implemented in BIP is a method of describing a "real" curriculum in terms of the specific skills that can be identified as a student's problem areas.

The next section describes BIP's implementation of the Curriculum Information Network and the algorithms that use it to select problems for students in an individualized manner.

## 3. Individualized task selection using the network

Computer-assisted instruction has long promised to present an individualized sequence of curriculum material, but in many cases this has meant only that "fast" students are allowed to detour around blocks of curriculum, or that "slow" students are given sets of remedial exercises. By describing the curriculum in terms of the skills on which the student should demonstrate competence, and by selecting tasks on the basis of individual achievement and/or difficulties, we intend to provide each student with programming tasks that are both challenging and instructive. Furthermore, the structure used in BIP can be applied to many other subject areas (such as statistics, algebra or reading) that involve identifiable skills and that require the student to apply those skills in different contexts and combinations.

We describe the curriculum as a set of goals, ordered by a tree hierarchy. In a subject that deals primarily with the formulation and solution of problems, as opposed to the absorption of factual information, a curriculum goal is to be interpreted as the mastery of a particular problem-solving technique specific to the subject matter. The desired end result, then, is the achievement of one or more top-level goals, each of which depends on one or more prerequisite goals. Each goal will be described in the program in terms of the acquisition of a set of skills, and the problems, or curriculum elements, are described in terms of the skills that must be applied to solve them. A skill may be developed in more than one goal, and will most certainly be used in several problems.

In BIP, curriculum goals involve the mastery of certain programming techniques. The techniques we have chosen include: simple output, using hand-made loops, using subroutines, etc. We have chosen for the purposes of our current research a very simple case of the full tree structure for goals. The techniques are linked in a linear order, each having but one "prerequisite", based on dependence and increasing program complexity. Other structures are attractive, but our current research deals primarily with individualizing the sequence of presentation of problems, once the curriculum structure has been specified in the CIN.

The techniques are interpreted as sets of skills, which are very specific curriculum elements like "printing a literal string" or "using a counter variable in a loop". The skills are not themselves hierarchically ordered. Appendix 1 is a list of the techniques and the skills they contain. The programming problems, or "tasks" are described in terms of the skills they use, and are selected on the basis of this description, relative to

the student's history of competence on each skill. Figure 2 shows a simplified portion of the curriculum network, and demonstrates the relationship among the tasks, skills, and techniques.

Essential among the curriculum elements that describe each task are its text, its skills, and its model solution. These elements, we feel, are also fundamental to the description of problems in many technical curriculums, and are broadly applicable in areas unrelated to instruction in programming. The optional elements in the task description are also useful categories in other subject areas, with modifications specifically suited to the given curriculum.

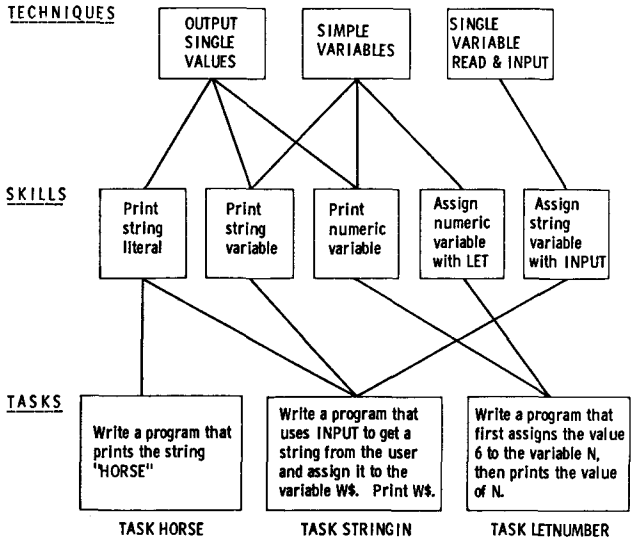


FIG. 2. A simplified portion of the curriculum network.

Computer programming, like many other procedural subjects, is better learned through experience than through direct instruction, especially if that experience can be paced at a speed suited to the individual student. Throughout the BIP course, the primary emphasis is placed on the solution of programming "tasks". BIP does not present a sequence of instructional statements followed by questions. Instead, a problem is described and the student is expected to write his own BASIC program to solve it. As he develops his BASIC program for each task, the student is directed to appropriate sections of the student manual for full explanations of BASIC statements, programming structures, etc. He is also encouraged to use the numerous student-oriented features, such as an interactive debugging facility and various "help" options described in Section 4.

When a student enters the course he finds himself in task "GREENFLAG", which requires a two-line program solution. Because this is expected to be his first programming experience, and perhaps his first interaction of any kind with a computer, he is led through the solution to the task in very small steps. GREENFLAG is the only task in the curriculum that presents text, asks questions, and expects the student to type "answers", which alleviates the trauma of being told to write a program in his first session. However, since the student's "answers" are frequently commands that are

passed to BIP's interpreter, he can see the genuine effects of his input, and he emerges from GREENFLAG having written a genuine program.

Figure 3 shows all the curriculum elements, including the skills, that describe each task. The text states the requirements of the task to the student, and suggests any prerequisite reading in the BIP student manual. The hints present additional information at the student's request, and subtasks isolate a part of the "main" problem as a smaller programming problem which he is to solve, helping him reduce the main task to separately soluble parts. The skills are the specific programming elements required in the solution. The model solution is a BASIC program that solves the problem presented in

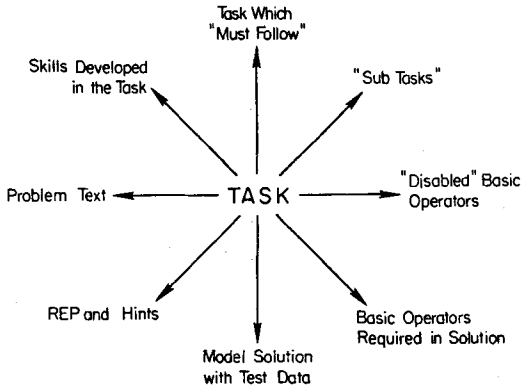


FIG. 3. Elements that describe a task.

the task, and is accessible to the student if he cannot reach his own solution. The model also contains coded test input data that is used to compare the results produced by the student's program against those of the model. The "must follow" tasks (if any) will follow the main task automatically, and require extensions of the student's original solution. The "required operators" are BASIC primitives that must be included in the student's program before he is allowed to progress out of the current task; the "disabled operators" are those primitives that, for pedagogical reasons, are not to be used in his solution program.

The sequence of events that occur as the student works on a task is shown in Fig. 4. When he has finished the task by successfully running his program, the student proceeds by requesting "MORE". His progress is evaluated after each task. In the "Post Task Interview" he is asked to indicate whether or not he needs more work on the skills required by the task, which are listed separately for him.

As soon as the student completes GREENFLAG, therefore, the instructional program knows something about his own estimation of his abilities. In addition, for all future tasks his solution is evaluated (by means of comparing its output with that of the model solution run on the same test data) and the results are stored with each skill required by the task. The program then has two measures of the student's progress in each skill: his self-evaluation and its own comparison-test results.

After completing a task (he may of course leave a task without completing it) the student is free either to request another, or to work on some programming project of

his own. The algorithm by which BIP selects a next task, if the student requests it, is shown in Fig. 5. The selection process begins with the lowest (least complex) technique. All the skills in that technique are put into a set called MAY, which will become the set of skills that the next task "may" use.

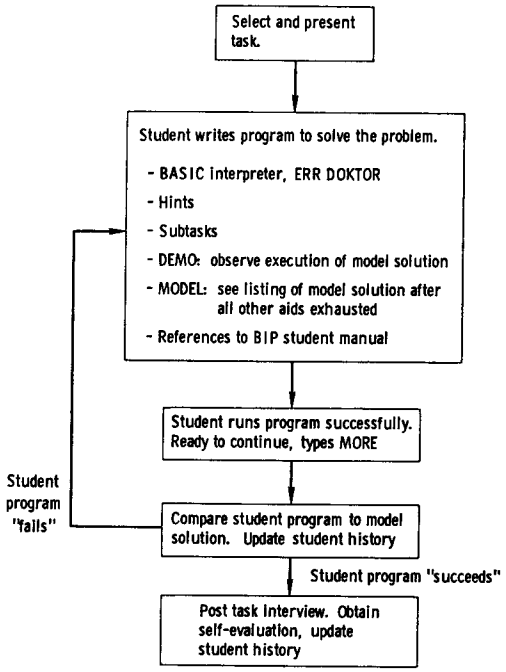


FIG. 4. Working through a task.

The program then examines the student's history on each of the skills associated with the technique, to see if it needs further work. This criterion judgment is the heart of the task selection algorithm, and we have modified it often. Two key counters in the history are associated with each skill. One is based on the results of the solution checker (described in section 4), and monitors the student's continuing success in using the skill. The other is based on his self-evaluation, and monitors his own continuing confidence in the skill. The current definition of a "needs work" skill is one on which either counter is zero, indicating that the student was unable to pass the solution checker the last time that skill was required in a task, or that he requested more work on the skill the last time he used it. Any such not yet mastered skills are put into the MUST set. Eventually BIP will seek to find a task that uses some of these "must" skills.

If no such skills are found (indicating that the student has mastered all the skills at that technique level), the search process moves up by one technique, adding all its skills to the MAY set, then seeking MUST skills again. Once a MUST set is generated, the search terminates, and all of the tasks are examined. Those considered as a possible next task for the student must (a) require at least one of the MUST skills, and (b) require no skills outside of the MAY set. Finally, the task in this group that requires the largest number of MUST skills is presented as the next task. Thus, in the simplified scheme,

shown in Fig. 2, assuming that the student had not yet met the criterion on the skills shown, the first task to be presented would be HORSE, because its skill lies in the earliest technique, and would constitute the first MUST set. Task LETNUMBER would be presented next, since its skills come from the next higher technique; STRINGIN would be presented last of these three.

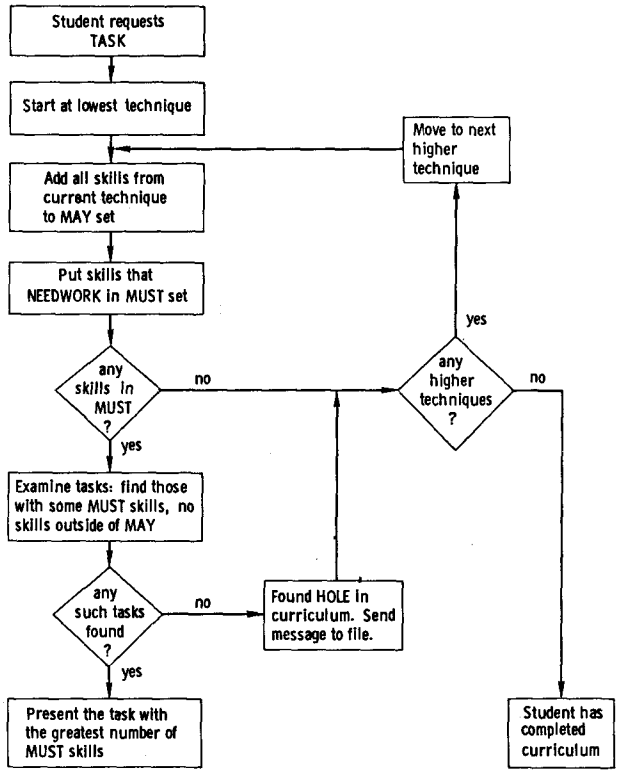


FIG. 5. Selecting the next task.

An interesting curriculum development technique has evolved naturally in this scheme. If BIP has selected the MUST and MAY sets, but cannot find a task that meets the above requirements, then it has found a "hole" in the curriculum. After sending a message to the course authors describing the nature of the missing task (i.e. the MUST and MAY skills), the task selection procedure examines the next higher technique. It generates new, expanded MUST and MAY sets, and searches for an appropriate task. Again, if none is found, a new search begins, based on larger MUST and MAY sets. The only situation in which this process finally fails to select a task occurs when the student has covered all of the curriculum.

Our work for the coming year will concentrate on student models that involve more than counter-type criterion judgments on the skills to be developed. We will attempt to characterize students' knowledge states and "difficulty" states explicitly by analyzing protocols. If we are successful, a production system type of student model (Newell, 1973) can be used to guide the student through the curriculum material.

## 4. BIP'S interpreter: a tutorial laboratory

Most of BIP's specially designed features are described by Barr *et al.* (1975). Since the publication of that report, a number of significant improvements have been made to existing structures, and a major graphic instructional feature has been added. The purpose of this section is to present the motivation for these modifications and to describe their operation. Table 1 lists the BIP commands available to the student grouped by their functions, as an overview of the system.

TABLE 1

*BIP's student commands*

| *Information* | *Instruction* |
|---|---|
| Who (is at this terminal) | Task |
| What (task am I doing) | More |
| When (is it now) | Reset (exit all tasks) |
| Gripe (to Stanford) | Enough (exit current task) |
| Calculator | |
| Hardcopy | |
| *Problem solving aids* | *Debugging aids* |
| Rep | Flow |
| Hint | Trace |
| Subtask | |
| Demo | |
| Model | |
| *Interpreter commands* | *File system* |
| Run | Files (to see directory) |
| Scratch | Save |
| Sequence (renumber lines) | Get |
| List | Merge |
| Edit (a line) | Kill |

### 4.1. IMPROVED ERROR CORRECTION IN THE INTERPRETER

Because the BIP course is aimed at students with no previous programming experience, the error messages are designed to contain more information than those supplied by "standard" BASIC systems, and they are carefully worded in non-computer-oriented terms to minimize confusion.

In many cases, these expanded error messages appear to provide enough information to help students correct their errors. However, especially in the case of the more naïve students, the generality of this error correction system proves to be a drawback. Consequently, the interpreter was substantially modified to provide more specific information about the particular error the student has made, primarily by identifying the place in his own line or program at which BIP detected the error.

For the student learning his first programming language the mechanical intolerance of the computer is often bewildering and frustrating. The slightest spelling mistake will cause the computer to behave in an unexpected way. A statement that seems clear to the

student may be much less obvious to the computer, and often for an obscure reason. One beginner, after successfully entering the line

```
10 PRINT X
```

into her program to print out the answer, could not understand why the improvement

```
10 PRINT THE ANSWER IS X
```

did not work as she had expected. Even more frustrating is the incomprehensibility of the computer's attempts at communication and clarification. Error messages are frequently confusing or even misleading to the novice. The error message

```
INVALID VARIABLE NAME
```

may be triggered by an extra comma or extra quotation mark in the student's line, rather than by a genuine error with a variable name.

Even a naïve user is quick to realize that a computer is not intelligent. Consider the following exchange in which a student was trying to erase her working program:

```
*SCRATCH
  "SCRATCH" IS NOT A VALID BIP COMMAND.
*WHAT IS A VALID BIP COMMAND?
  "WHAT IS A VALID BIP COMMAND?" IS NOT A VALID BIP
  COMMAND.
```

Originally, BIP produced this kind of conversational but absurd response, which provides little useful information. The student begins to distrust error messages like this because the interpreter makes such obviously stupid mistakes, while pretending to produce intelligent natural language. Currently, BIP handles the student's input in a more direct and "honest" way, as illustrated in the sample dialogue in section 5, with appropriately modest messages like "YOUR PROGRAM DOESN'T SEEM TO SOLVE THE PROBLEM", and "ERROR FOUND NEAR . . ." The student is not misled as to BIP's capabilities, and is thus encouraged to look around the error indication to find the error itself.

Another difficulty common to new students is an inadequate or inaccurate conception of what the execution of their program entails. In most systems, variable assignments, logical decisions, and program branching are all invisible to the user, and there is no way that the student can conveniently see the flow of execution of his program. Since learning to debug is a very important part of learning to program, interactive graphic debugging systems are useful tools that can greatly assist the student's conceptualization of program execution. BIP makes available two such facilities, which have proved to be very useful both to students and to more experienced programmers. (These debugging facilities are described later in this section.)

In BIP's BASIC interpreter we have attempted to deal with the problems inherent in student–computer communications. Since BIP runs in an interactive environment, the student receives immediate feedback about his syntax errors, and information about other errors as soon as they are detected, keeping him from going too far down the wrong track without some warning. BIP's interpreter is built right into the instructional program so that the instructional system can continue to offer assistance after the first error message is given by the interpreter.

BIP's error detection capabilities cover four different kinds of program errors: syntax and execution time errors, program structure errors detectable before execution but involving more than the syntax of one line, and errors related to the curriculum task assigned making an otherwise correct program an unacceptable solution to the problem. Although it is not always possible to give an appropriate syntax error message (syntactically wrong statements are by their very nature ambiguous), we have tried to make BIP's error messages as accurate as possible. In addition, we have added clarifying messages for each error, including examples of correct and incorrect statements, which the student receives upon request. He may also ask for a reference to the part of the BIP manual that explains in detail the statement he is trying to use. We are currently designing a problem-dependent error message scheme that will elaborate on errors involving certain skills introduced in a given problem.

BIP uses a top-down parser to produce an internal code that can be efficiently executed. The parser is used to detect syntax errors as well. In addition to normal error checking, the parser specifically looks for certain very common errors. For example, if the student types:

```
10 IF J = 0 THEN GO TO 50
```

an error message informs him that GOTO's are not allowed in IF statements. But rather than merely inform the student that his statement was incorrect, BIP pinpoints the student's error:

```
10 IF J = 0 THEN GO TO 50
                    ↑
SYNTAX ERROR: "GOTO" IN IF STATEMENT
LINE NOT ACCEPTED (TYPE ? FOR HELP)
```

A flashing arrow indicates the point at which the parser detected the error. Although the error message tells the student what is wrong, it does not tell him how to construct a correct IF statement. To get more information he types a question mark, and sees:

```
DON'T USE A "GOTO" IN AN "IF" STATEMENT. IT'S NOT LEGAL THERE
TYPE ? FOR MORE HELP
```

Alternatively, the student may type ?REF for a reference to the section in the BIP manual containing a detailed explanation of IF statements.

Similar information is available following an execution error. For example, if the student ran the following program:

```
10 DIM L (10)
20 FOR N = 1 TO 11
30 L(N) = 0
40 NEXT N
99 END
```

he would receive a runtime error:

```
EXECUTION ERROR: LINE 30
INDEX FOR SUBSCRIPTED (LIST) VARIABLE OUT OF DECLARED
  BOUNDS
SUBSCRIPT OF L IS 11
```

A question mark would elicit additional information:
FOR EXAMPLE,

```
10 DIM X(20)
20 J = 25
30 X (J) = 0
```

IS INCORRECT, BECAUSE 25 IS GREATER THAN THE DIMENSION OF X

In addition to syntax and execution time errors, there are some program structure illegalities that can be detected before execution. (Strictly speaking, these are syntax errors, but they involve more than one line of code and are not handled well by BASIC interpreters.) We have found that identifying these structural bugs, rather than letting them eventually appear as execution errors, can often clarify the error for the student. The ERROR DOCTOR routine is called by the RUN procedure, and examines the program before attempting execution. It notifies the student of errors like branches to non-existent lines, branches from a line to itself, illegally nested FOR. .NEXT loops, and improper transfers into and out of subroutines.

Since the BIP course runs without human graders, a simple "solution checker" is built in to determine whether the student's program is a solution to the problem he was to solve. The checker runs the stored model solution and compares its output to the output of the student's program. While this approach does not catch all possible bugs, it is the method used by most human instructors in grading programming assignments; it is quick and sufficient.

The solution checker executes the model solution invisibly, storing its output, then executes the student's program in the same way. Each time the student's program produces output, it is compared to the list stored from the execution of the model, and any matching element in that list is flagged. If any unflagged output remains in the list when execution is completed, the student is told that his program "doesn't seem to solve the problem". If all outputs of the model have been matched, he is told that his program "looks O.K.".

Because many of BIP's tasks require interactive programs that deal with a hypothetical user, the solution checker must be able to perform its comparison-by-execution on suitable test values, those that might be given by the user in response to an INPUT statement in the program. These values are stored with the model solution as part of the curriculum, and are chosen to provide a test of the student's program appropriate to the level of difficulty of the task and to the explicit requirements stated in its text.

When BIP executes the student's program, it asks him to provide the names of the variables he used for the INPUT functions required; an example might be "What variable do you use for the user's first number?" (The description of the variable's function is also stored in the model, as a REM statement that gives information but does not affect execution in any way.) Then the solution checker assigns the test values to the student's own variables, allowing it to execute his program on exactly the same input as was used in executing the model. If the student's program is found to be inadequate, he is advised to run the DEMO to see the model in action. For interactive programs, the test data are suggested as appropriate input to a DEMO, where the student can see the execution of the model solution with those values. Frequently, the student's program will fail to deal with the test values, and the failure is evident from the correct example provided

by the DEMO. In these cases, the solution checker gives instructive help in just those areas that cannot be identified by the other error detection facilities.

The solution checker ignores output of string constants, since they are frequently optional messages oriented toward the hypothetical user of the student's program, and rarely affect the real correctness of the program. Extraneous output (i.e. output beyond that produced by the model) is also ignored, for much the same reasons. However, in those tasks where string constants of "extra" output are considered relevant to the correctness of the program, the focus of the solution checker can be explicitly narrowed by additional information coded and stored along with the test values.

Though simple and obviously limited in some ways, BIP's solution checker is an effective tool, not only in acting as a grader that governs progress through the curriculum, but also as an additional source of instructive information. Particularly in the more demanding interactive tasks, the values used by the checker and suggested to the student add to the beneficial learning effects of hands-on experience and manipulation of his own BASIC programs.

### 4.2. INTERACTIVE GRAPHICS FEATURES

In addition to the instructive capabilities built into the interpreter, BIP offers two facilities that use the CRT display screen as a dynamic source of information. The REP command presents a flowchart-like representation of the model solution for each problem, which can be probed by the student to reveal more information about the model's programming structure. The FLOW command is a debugging aid that allows the student to execute his own program a line at a time, and makes each step of the execution fully visible.

Several types of "help" information are stored with each task as part of the curriculum network. Here we describe REP, the graphic problem solving aid, in some detail, using a specific example from the curriculum. The requirements of the sample task are:

TASK SLANT: WRITE A PROGRAM THAT USES ONLY ONE PRINT STATEMENT AND PRINTS THIS PATTERN. IT SHOULD PRINT AS MANY @ SIGNS AS THE USER REQUESTS.

```
@
  @
    @
      @
        @
          @
            @
              @
```

The solution is short but the task is difficult:

```
5 PRINT "HOW MANY @ SIGNS WOULD YOU LIKE?"
10 INPUT N
20 S$ = ""
30 FOR I = 1 TO N
40 PRINT S$ & "@"
50 S$ = S$ & " "
60 NEXT I
99 END
```

The critical points of the task are that the variable S$ must be initialized to the null string, the PRINT statement must concatenate the accumulated spaces to the "@" character, and S$ must be reassigned correctly (a space appended to the string) within the loop.

Plate 1 illustrates the information available to the student via REP. The figure shows the initial REP display and two responses to the student's request for more information on the loop structure labeled "E".

In using REP the student is allowed to probe the representation in both breadth and depth and in any sequence. If he probes in the breadth dimension he may first look at control structure information and find that the program requires a loop. Next he may look at INPUT/OUTPUT or other key information. Thus, once he has established that the program requires a loop, more information might be requested on control structure until, finally, he is shown the actual BASIC code. The implementation of REP allows us to experiment with various aspects of its operation; flags can be set to control which labels will blink, how much information will be displayed whether or not REP itself is available to a given student, etc. We are currently using data from experiments that tested the use of REP to determine a way to use the student's probes to update BIP's model of what he understands more accurately.

We have implemented two tracing facilities to assist the student in conceptualizing the execution of his program. Tracing a program is difficult to do correctly by hand, since one tends to make the same mistakes over and over. It is especially difficult for beginning programmers, who may not understand the function of some statements. BIP's TRACE option automates this process. It allows the student to see exactly how his program is executing, and to identify the point at which the program begins to stray from what he intended. As each line of his program executes, the line number is displayed on his teletype or display terminal. Any variable assignments performed in that line are also indicated, as well as any input or output.

FLOW is a more sophisticated program tracing aid designed for CRT displays. The main program is displayed on the terminal, with the text of all subroutines removed. Each time the student presses the CR key, one line of his program is executed, and its line number blinks on the screen display. When an IF or GOTO statement is executed, an arrow is drawn on the screen to indicate the transfer of control When a subroutine is called, the main program display is replaced by the lines that make up the subroutine. Additionally, a message in the corner of the screen indicates the level of nested subroutines.

The student may also request that up to six variables be traced. If an array is traced, the value of the most recently assigned array element is shown. In addition to the variables to be traced, a line number may be specified. The program will execute continuously without waiting for the student to press the key, until the specified line is reached. At that point, the program will resume step-by-step execution. This feature allows the student to reach the troublesome part of his program quickly.

Plate 2 illustrates a hypothetical FLOW through a program written for the SLANT task. The cursor and flashing line numbers show the line that is being executed, the values of the variables I and N are shown at the top of the screen, and input and output appear below the program listing. The different states illustrated in the figure show (1) assignment by INPUT, (2) output, (3) reassignment of the index and reiteration of the

loop after the NEXT, and (4) reassignment and continuation after the index has passed the terminating condition.

## 5. An annotated dialogue

BIP is a programming laboratory. It presents an individually sequenced set of problems that the student is to solve by writing BASIC programs. Typically, the tasks present very little instruction in the syntax of BASIC; a manual describing the language and the BIP system is provided for reference and detailed information. The task shown here is clearly not one of the first that the student would see, since it requires some skill both in planning the program's structure and in using appropriate BASIC constructs to carry out the job.

The student/BIP interaction is shown on the left, with all student input preceded by an asterisk. The dialogue begins as the student signs on to the system. Our explanatory annotation is indented to the right.

WELCOME TO BIP !!
TYPE "?" FOR HELP

> Some sort of assistance is always available. If the student types a question mark, BIP's response is related to the most recent error, as will be seen below. When the student requests a problem, BIP examines his history and tries to select an appropriate one.

*TASK
BEGIN 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 FIRST DAY). 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 NUMBER OF GIFTS SENT ON THAT TWELFTH DAY. (THIS IS NOT THE SAME AS THE TOTAL NUMBER OF GIFTS SENT THROUGHOUT ALL 12 DAYS—JUST THE NUMBER SENT ON THAT SINGLE 12TH DAY.)

> This is a fairly difficult problem for most students, and several problem-solving aids are available. In the previous section we discussed the REP feature which helps students break down problems with flowcharts.
>
> Additional text material is also available, either as a clarification of the task itself or, as in this case, to help the student construct his own algorithm.

*HINT
HINT #1
THIS PROGRAM REQUIRES A LOOP. EACH EXECUTION OF THE LOOP INVOLVES ACCUMULATING THE VALUE OF THE INDEX INTO A TOTAL.

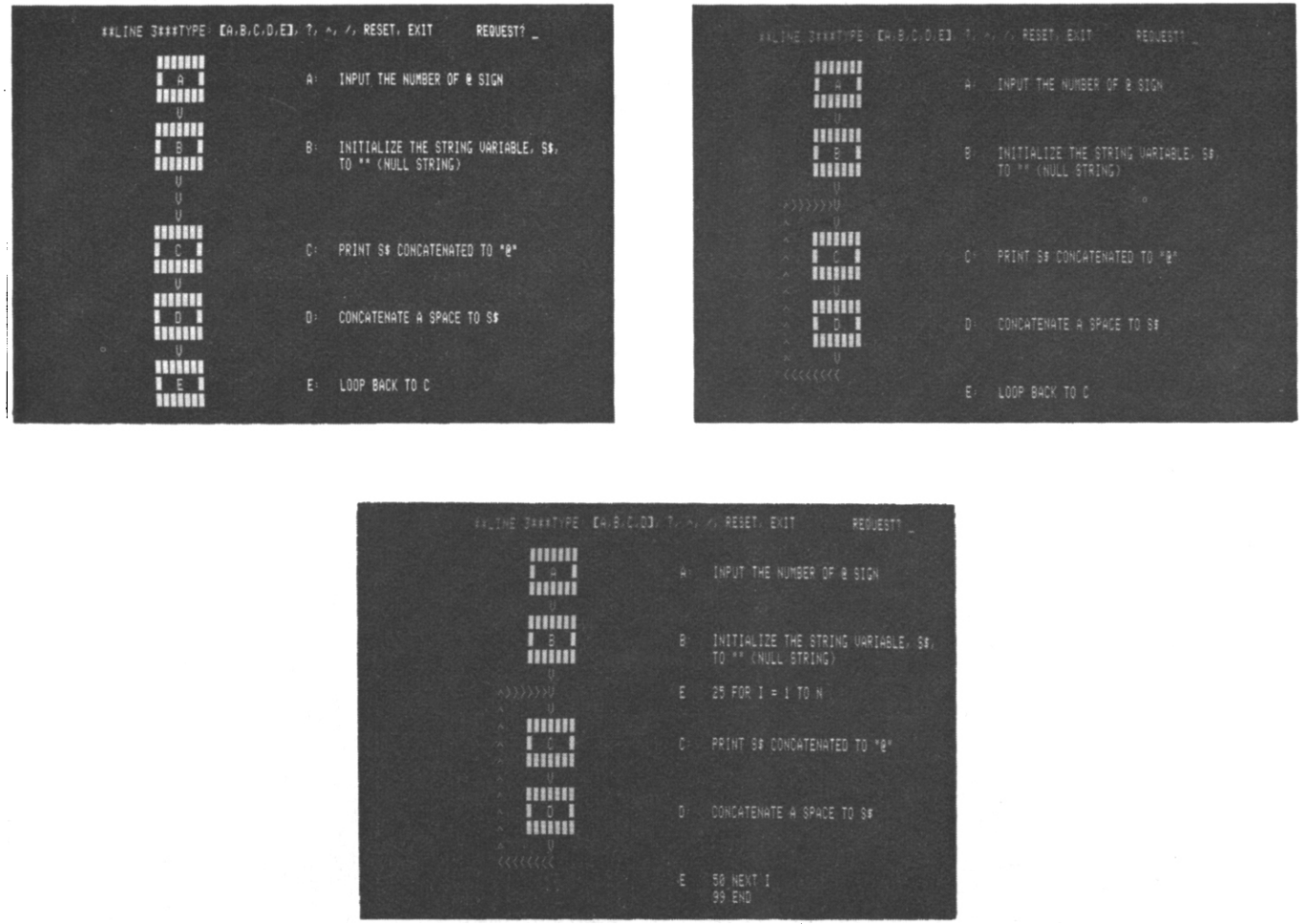


PLATE 1. The REP display.

(facing p. 582)

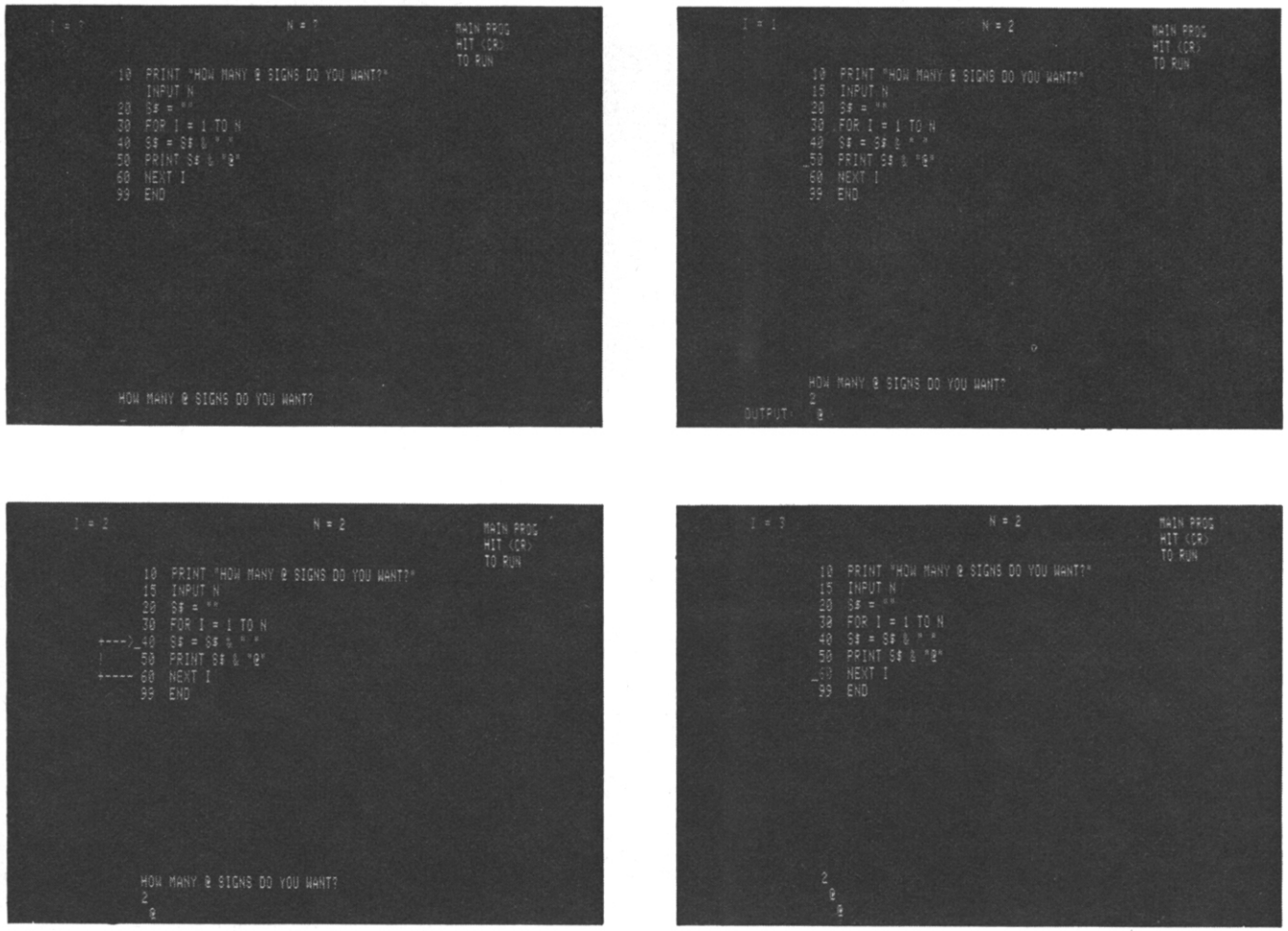


PLATE 2. The FLOW display.

```
*HINT
HINT #2
FINDING A TOTAL OR SUM ALMOST ALWAYS MEANS TWO THINGS:
1. SETTING A VARIABLE EQUAL TO ZERO OUTSIDE A LOOP.
2. ACCUMULATING VALUES INTO THAT VARIABLE WITHIN THE LOOP.
IN WORDS, TOTAL EQUALS OLD TOTAL PLUS ANOTHER VALUE.
```

In some cases, the task involves a number of complications, some of which can be isolated and presented as smaller "subtasks". This subtask directs the student's attention to the problem of writing a simple loop using a FOR. .NEXT statement. The student always has the option of completing the subtask, or of returning to the "main" task as soon as he has gained the information he thinks he needs to solve the original problem.

```
*SUB
BEGIN TASK: S:FORLOOP
WRITE A PROGRAM WITH A LITTLE -FOR. .NEXT- LOOP.

THE WORK OF THE LOOP IS JUST TO PRINT THE VALUE OF THE LOOP'S
INDEX (THE VARIABLE THAT KEEPS TRACK OF THE NUMBER OF TIMES
THE LOOP HAS BEEN EXECUTED).

WHEN YOU RUN THE PROGRAM, IT SHOULD LOOK LIKE IT IS COUNTING
FROM 1 TO 10.
```

The subtask is treated as a regular task by BIP, and can have hints and subtasks of its own. Here the student makes several attempts at a solution, illustrating the error detection facilities.

First, although the problem specifically required a FOR. .NEXT loop, the student will try to construct a "hand-made" loop of his own.

```
*10 LET I = 1

*20 PRINT "THE INDEX IS; I
             ↑
SYNTAX ERROR: UNMATCHED QUOTE MARKS—FOUND NEAR '"THE
INDEX IS' LINE NOT ACCEPTED (TYPE ? FOR HELP)
```

BIP's interpreter was specially written to provide as much instruction and error correction as possible, in contrast to "standard" BASIC systems. Here the student has made a common error by omitting the closing quote marks in his PRINT statement. BIP is able to indicate the point at which it detected the error, and to give further information specifically related to this particular incorrect BASIC statement:

```
*?
' "THE INDEX IS' HAS AN ODD NUMBER OF QUOTE MARKS
REMEMBER THAT ALL STRINGS MUST HAVE A QUOTE AT THE
BEGINNING AND END

TYPE ? FOR MORE HELP
```

```
*?REF
II.7, II.8
```

BIP can always present an appropriate manual reference for all errors.

This student gives up the fancy PRINT statement and includes a simpler but adequate one. Unfortunately he is soon to find that his loop construction is unacceptable for this task:

```
*20 PRINT I

*30 IF I = 9 THEN 99
FOR THE PURPOSES OF THIS TASK, DON'T USE THE 'IF' STATEMENT.
```

His original plan for a loop including a conditional branch will have to be abandoned. Undaunted, the student now tries to solve the problem without a loop. He first deletes the lines he has entered so far.

```
*SCR

*10 PRINT 1

*20 PRINT 2

*30 PRINT 3

. . .
```

After entering all of the PRINT statements, he lists his program, and then runs it.

```
*LIST
10 PRINT 1
20 PRINT 2
30 PRINT 3
40 PRINT 4
50 PRINT 5
60 PRINT 6
70 PRINT 7
80 PRINT 8
90 PRINT 9
99 END

*RUN

EXECUTION OF YOUR PROGRAM

1
2
3
4
```

5
6
7
8
9
EXECUTION COMPLETED AT LINE 99

The program runs without execution errors. When the student considers his program to be an adequate solution to the problem posed in the task, he requests BIP's evaluation of it by typing MORE, indicating that he wants to proceed to another task. Here, the student seems to be satisfied, and decides to continue.

*MORE
WAIT. SOMETHING IS MISSING. FOR THIS TASK, YOUR PROGRAM SHOULD ALSO INCLUDE THE FOLLOWING BASIC STATEMENT(S):
FOR
ADD TO YOUR PROGRAM AND -RUN- IT AGAIN BEFORE YOU TYPE 'MORE'

BIP has recognized that the essential FOR. .NEXT loop is missing from the student's program, and refuses to evaluate it further. Finally he sees the light, deleting his last effort and trying again, this time with a FOR. .NEXT loop.

*SCR
*10 FOR I = 1 TO 9
*20 PRINT I
*99 END

This three-line program is of course syntactically incorrect, as there is an open loop. BIP has a routine called ERROR DOCTOR which checks the structure of complete programs before they are run, so that such errors do not appear as less informative execution time errors.

*RUN
ERROR DOCTOR DIAGNOSIS: 'FOR' STATEMENT WITHOUT MATCHING 'NEXT'

LOOK AT LINE NUMBER: 10
TYPE ? FOR HELP

As always, more help is available after the original error message.

*?

YOU ARE MISSING THE -NEXT- STATEMENT CORRESPONDING TO THE -FOR- ON THIS LINE

TYPE ? FOR MORE HELP

After listing his program, the student sees and corrects the loop error, and the program executes successfully.

```
*LIST
10 FOR I = 1 TO 9
20 PRINT I
99 END

*30 NEXT I

*RUN
EXECUTION OF YOUR PROGRAM
1
2
3
4
5
6
7
8
9
EXECUTION COMPLETED AT LINE 99
```

Again, the student is ready for MORE. BIP evaluates his solution by invisibly executing both the model solution and his program on the same "test" data, then comparing their output. In this case, the model produced output different from that of the student's program, so he is told that his program doesn't seem correct. He elects to remain in the task to try again.

```
*MORE

EXECUTION OF THE MODEL

EXECUTION OF YOUR PROGRAM

YOUR PROGRAM DOENS'T SEEM TO SOLVE THE PROBLEM.

IF YOU ARE COMPLETELY CONFUSED OR LOST, YOU CAN LEAVE THIS
TASK NOW

DO YOU WANT TO STAY IN THE TASK? TYPE Y OR N:  Y

O.K. TRY AGAIN. YOU MIGHT LOOK AT A -DEMO- OF THE MODEL SOLU-
TION AND COMPARE ITS OUTPUT TO THE OUTPUT OF YOUR PROGRAM.
```

Many of BIP's tasks are interactive: the student's program must be able to ask questions of a hypothetical user, and to deal with the user's input in appropriate ways. The DEMO feature allows the student to execute the stored model solution to see the behavior required of his program. In this task, the demo serves to remind the student that his program was supposed to count to 10, not 9.

```
*DEMO
EXECUTION OF THE MODEL
THE INDEX IS 1
THE INDEX IS 2
THE INDEX IS 3
THE INDEX IS 4
THE INDEX IS 5
THE INDEX IS 6
THE INDEX IS 7
THE INDEX IS 8
THE INDEX IS 9
THE INDEX IS 10
WE ARE NOW OUT OF THE LOOP.
EXECUTION COMPLETED AT LINE 99

DEMO ENDED
```

And he corrects the first line. BIP reminds the student that the line has been changed.

```
*10 FOR I = 1 TO 10
WAS: 10 FOR I = 1 TO 9

*RUN
EXECUTION OF YOUR PROGRAM
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
EXECUTION COMPLETED AT LINE 99
```

Again satisfied, the student asks for MORE.

```
*MORE

EXECUTION OF THE MODEL

EXECUTION OF YOUR PROGRAM

LOOKS O.K.
```

Following the successful comparison, BIP presents the "post-task interview," in which the student's own evaluation of his understanding and skills is acquired and stored.

First the model solution is displayed, and the student is asked whether or not he understands how it works.

```
POST TASK INTERVIEW

HERE IS A TYPICAL SOLUTION TO THIS TASK:
10 FOR I = 1 TO 10
20 PRINT "THE INDEX IS "; I
30 NEXT I
40 PRINT "WE ARE NOW OUT OF THE LOOP."
99 END

DO YOU UNDERSTAND THE SOLUTION? (Y/N): Y
```

Then, for each of the specific skills required in the construction of the program for this task, the student is asked to indicate whether or not he feels confident in his use of that skill. These questions are a major source of information for BIP's task selection process, by which the next task will be chosen at a level appropriate for this student. Those skills to which the student responds "Y" will not be sought in his next task, since he feels that he has had enough work on them. Those to which he responds "N", on the other hand, will be looked for explicitly. Unless he has exhausted a portion of the curriculum, some of those "N" skills will be required in his next task, providing him with the opportunity to use those skills again in a new context.

```
THINK ABOUT THE SKILLS USED IN THIS TASK. FOR EACH SKILL,
  TYPE Y IF YOU HAVE HAD ENOUGH WORK WITH THAT SKILL.
  TYPE N IF YOU THINK YOU NEED MORE WORK ON IT.
FOR. .NEXT LOOPS WITH LITERAL AS FINAL VALUE OF INDEX: Y
MULTIPLE PRINT [STRING LITERAL, NUMERIC VARIABLE]: N
```

Since he did not use the "multiple print" statement shown in line 20 of the model, our student indicates that that skill would be appropriate in his next problem.

BIP informs him that he has returned to the larger task at hand, and allows him to have its text re-displayed.

```
RETURNING FROM A SUB TASK.
YOU ARE IN TASK XMAS.
DO YOU WANT THE TEXT PRINTED OUT? TYPE Y OR N.
        *N
*
```

## 6. An experiment evaluating BIP's individualization scheme

An experiment comparing the effectiveness of BIP's teaching strategy with a more traditional "branching" strategy was run in February and March, 1975. BIP's task selection strategy is described in detail in section 3. The control group followed a

predetermined branching strategy through the curriculum arrived at as follows: a committee of staff members with experience both in curriculum design and in teaching programming ordered the tasks by complexity and the inherent hierarchy of programming concepts required for their solution. For each task, two "next" tasks were specified, one to be presented if the student successfully solved the current task without seeing the model solution, and the other to be presented if he failed. All of the existing BIP tasks were incorporated into this fixed path, as either "main line" or remedial problems.

Forty-two Stanford students, 22 men and 20 women, were recruited as subjects for the experiment. All were given the Computer Programming Aptitude Battery (1964) as a pretest. Two matched groups, each with 11 men and 10 women, were created by ranking the subjects' pretest scores and alternately assigning subjects to groups.

Subjects worked at CRT terminals for 10 one-hour sessions, signing up for each hour one or two days in advance. Of the original 42 subjects who took the pretest, one woman failed to begin work on the course; the other 41 all completed the 10 hours' work within three weeks and then took an off-line post-test.

The strategy for selecting the "next task" was the only difference in treatment between the groups. Since this process was invisible to the students, their interactions with BIP appeared identical. They had access to all of the interpreter and assistance features, and both groups were given the post-task interview after each task, although the information collected there was not used in the task selection decisions for the fixed path group.

Extensive data were collected on-line as the subjects worked on the course, including complete protocols whose analysis we are now attempting to automate for our research on student models. For purposes of comparing the performance of the two treatment groups, the information recorded for each task is most interesting. This information includes:

(a) whether the subject "passed" the solution checker on this task;
(b) whether he passed the checker on his first attempt;
(c) whether he said he understood the model solution in his PTI;
(d) whether he requested and saw the model solution before completing the task.

In addition, a comprehensive post-test was administered off-line. The test was designed to measure the students' ability to interpret correct BASIC programs, to complete specified sections of incomplete programs, and to construct entire programs.

An analysis of variance was performed on the task data, measuring the effect of treatment. The results are summarized in Table 2. The experimental group is labeled "tsa" since their tasks were selected by BIP's task selection algorithms; the "path" group followed the predetermined branching strategy through the curriculum. Some conclusions about these results can be drawn from the analysis completed at this time. First, there was no significant difference between the groups' post-test scores (means were 109·0 and 108·2 for the experimental and control groups, respectively) indicating that the two task selection strategies apparently produced the same amount of learning of the material tested. However, the data in Table 2 show a significant difference in the character of that learning experience. During their ten contact hours, students in the experimental group worked 25% more problems than those who followed the predetermined problem sequence, and had significantly less trouble working the problems they were presented, as evidenced by the higher percentage they completed correctly and said they understood in the post-task interview. It should be stressed that the two groups

saw the same problems, but in a different sequence. Neither the evaluation forms filled out after each task nor their post-test scores indicated that they were getting problems that were too easy. We believe that these results show that BIP's task selection algorithm did indeed choose appropriate problems for each student on the basis of a running history, and that the net effect, although it fell short of increasing learning speed in this situation, was a significant improvement over the branching strategy devised by our experts.

TABLE 2
*Mean performance of experimental (tsa) and control (path) subjects*

| | Mean | | Diff | *F* |
|---|---|---|---|---|
| | tsa | path | | |
| Total number of tasks seen | 37·7 | 29·4 | 8·3 | 21·92** |
| % passed solution checker | 90·7 | 82·3 | 8·4 | 6·32* |
| % passed checker the first time | 73·0 | 62·3 | 10·7 | 7·41* |
| % "understood" in pti | 91·1 | 83·2 | 7·9 | 6·68* |
| % where model solution was seen | 10·8 | 14·3 | 4·5 | 0·85 |

*$F$ crit (1,32) = 4·17, $p$ <0·05.
**$F$ crit (1,32) = 5·57, $p$ <0·01.

The data collection routines were designed to be nearly exhaustive, recording all information that we felt might be interesting in some aspect of future work on the design of task selection strategies, student models, and curriculum description; for this reason, we feel that much more is yet to be derived from the data than the results we give here. Still, we are confident that BIP's strategy, by ordering the presentation of tasks on the basis of its continually updated knowledge of each subject's progress, did change the character of the interaction between the teaching system and the students.

## 7. Current goals and plans

It is our goal to improve student performance significantly by tailoring the presentation of tasks more closely to each student's strengths and weaknesses, and we feel that the experimental results indicate a positive first step in that direction. Work during the current year focuses on the three major components of the individualization process: the curriculum description, the student model, and the task selection algorithm or strategy. In order for BIP to present a reasonable task, the curriculum must be represented internally such that BIP can recognize the aspects of each task that qualify it as appropriate for the student's current level of ability. The description of the student, similarly, must represent his abilities in a format that makes it possible for BIP to identify the kind of task that is needed. Finally, the task selection strategy must serve as an interface between the two descriptions, defining an appropriate task in terms of the current state of the two data bases. Further analysis of these data is planned, not only to discern other differences between the two groups. More importantly, we will use the results in designing new forms of the three components of the individualization scheme.

We are also concerned with the difficulty and cost of running these large scale comparative experiments. At least one staff member was absorbed by the logistics of the February experiment during the recruiting and pretesting period, three weeks of classroom interaction with BIP, and the time required for administering post-tests. Considerable effort was expended to insure that the routines that controlled the two treatment groups worked properly, and to insure that the data were collected exactly as planned. No changes could be made to these routines during the subjects' work with BIP, so it was necessary (as in any live-subject experiment) to devote about a month to the perfection of the program. The information collected, while extensive and very useful for a number of purposes, still relates only to two conditions, only one of which (the "tsa" group) is of interest for future development.

Still, such comparative studies should be run for each of the many design decisions made during the development of the student model and task selection algorithms. An alternative means of generating experimental results is needed, to provide efficient comparison and evaluation of our designs. We are developing a procedure for obtaining detailed information about BIP's ability to individualize instruction by simulating large-scale experiments like this one instead of actually carrying them out. With the simulation we expect to obtain reasonable data about new student models as they evolve so that future real-subject experiments focus on more specific evaluations of the task selection process.

The BIP course has recently been made available to the ARPA net community, and interested users who have access to the net should contact the authors. During the coming year we will rewrite the entire BIP system in BASIC to make it available on smaller BASIC systems to those students it was designed to teach.

The main thrust of our work for the coming year will, we hope, be an investigation of the nature of the students' programming bugs. We think that analysis of the hundreds of collected protocols of BIP students will be a valuable addition to current AI work on understanding programming and classifying bugs. Furthermore, with sufficient understanding of where a student is likely to err in a given problem, and what misunderstanding would cause each error, some very impressive improvement will be possible in the immediate context of BIP.

This research was supported jointly by the Office of Naval Research and the Advanced Research Projects Agency of the Department of Defense on contract number N00014-67-A-0012-0054. The authors thank Oliver Buckley, Richard Kahler, Jay Lindsay, and William Swartout for their contributions to BIP.

## References

BARR, A., BEARD, M. & ATKINSON, R. C. (1975). A rationale and description of a CAI program to teach the BASIC programming language. *Instructional Science*, **4**, 1–31.

BEARD, M., LORTON, P., SEARLE, B. & ATKINSON, R. C. (1973). *Comparison of student performance and attitude under three lesson-selection strategies in computer-assisted instruction* (Tech. Rep. 222). Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences.

BROWN, J. S., BURGON, R. R. & BELL, A. (1974). *An intelligent CAI system that reasons and understands* (BBN Report 2790). Cambridge, Mass.: Bolt Beranek and Newman.

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

CARBONELL, J. R. & COLLINS, A. M. (1973). Natural semantics in artificial intelligence. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford, Calif., August.

COLLINS, A. M., PASSAFIUME, J. J., GOULD, L. & CARBONELL, J. R. (1973). *Improving interactive capabilities in computer-assisted instruction* (BBN Report 2631). Cambridge, Mass.: Bolt Beranek and Newman.

DANIELSON, R. & NIEVERGELT, J. (1975). An automatic tutor for introductory programming students. *SIGSCE Bulletin*, **7**.

DIGITAL EQUIPMENT CORPORATION (1968). *Algebraic Interpretive Dialogue*. Maynard, Mass.: Digital Equipment Corporation.

FRIEND, J. (1973). *Computer-assisted instruction in programming: A curriculum description* (Tech. Rep. 211). Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences.

GOLDBERG, A. (1973). *Computer-assisted instruction: The application of theorem-proving to adaptive response analysis* (Tech. Rep. 203). Stanford, Calif.: Stanford University. Institute for Mathematical Studies in the Social Sciences.

GRIGNETTI, M. C., GOULD, L., HAUSMANN, C. L., BELL, A. G., HARRIS, G. & PASSAFIUME, J. (1974). *Mixed-initiative tutorial system to aid users of the on-line system (NLS)* (ESD–TR–75–58). Bedford, Mass.: Deputy for Command and Management Systems, Electronic Systems Division.

KOFFMAN, E. B. & BLOUNT, S. E. (1975). Artificial intelligence and automatic programming in CAI. *Artificial Intelligence*, **6**, 215–234.

LORTON, P. & SLIMICK, J. (1969). Computer-based instruction in computer programming: A symbol manipulation-list processing approach. *Proceedings of the Fall Joint Computer Conference*, pp. 535–544.

NEWELL, A. (1973). Production systems: Models of control structures. In W. G. CHASE, Ed., *Visual Information Processing*. New York: Academic Press.

SANDERS, W., BENBASSAT, G. & SMITH, R. L. (1976). Speech synthesis for computer-assisted instruction: The MISS system and its applications. *SIGSCE Bulletin*, **8**, 200–211.

SCIENCE RESEARCH ASSOCIATES (1964). *Computer Programming Aptitude Battery*. Chicago, Ill.: Science Research Associates.

SELF, J. A. (1974). Student models in computer-aided instruction. *International Journal of Man–Machine Studies*, **6**, 261–276.

SMITH, R. L., GRAVES, H., BLAINE, L. H. & MARINOV, V. G. (1975). Computer-assisted axiomatic mathematics: Informal rigor. In O. LECARME & R. LEWIS Eds., *Computers in Education* (IFIP 2nd World Conference). Amsterdam: North Holland.

SUPPES, P., SMITH, R. & BEARD, M. (1975). *University-level computer-assisted instruction at Stanford: 1975* (Tech. Rep. 265). Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences.

VAN LEHN, K. (1973). *SAIL users manual* (Artificial Intelligence Memo 204). Stanford, Calif.: Stanford University, Stanford Artificial Intelligence Laboratory.

## Appendix 1

BIP's CURRICULUM STRUCTURE

It is a difficult task to determine the fundamental elements of an arbitrary curriculum and to discern or impose a structure. The detail of description required by BIP's task selection algorithms seemed impossible to attain, and indeed we have never settled down to one satisfactory description of the introductory programming curriculum, although we do feel that we understand it better than we did. We have included in this appendix a list of most of the basic skills we have identified, grouped by the subgoals or "techniques" they first appear in. Some skills, like "the END statement" are not included in the technique structure because they are not valuable in task selection decisions. Although this description is neither complete nor final, it certainly represents considerable effort and should be very valuable to others interested in teaching computer programming or in curriculum analysis.

Simple output—first programs

print numeric literal
print string literal
print numeric expression [operation on literals]
print string expression [concatenation of literals]

Variables—assignment

print value of numeric variable
print value of string variable
print numeric expression [operation on variables]
print numeric expression [operation on literals and variables]
print string expression [concatenation of variables]
print string expression [concatenation of variable and literal]
assign value to a numeric variable [literal value]
assign value to a string variable [literal value]

More complicated assignment

assign to a string variable [value of an expression]
assign to a numeric variable [value of an expression]
re-assignment of variable (using its own value) [string]
re-assignment of variable (using its own value) [numeric]
assign to numeric variable the value of another variable
assign to string variable the value of another variable

More complicated output

multiple print [string literal, numeric variable]
multiple print [string literal, numeric variable expression]
multiple print [string literal, string variable]
multiple print [string literal, string variable expression]

Interactive programs—INPUT from user—using DATA

assign numeric variable by –INPUT–
assign string variable by –INPUT–
assign numeric variable by –READ– and –DATA–
assign string variable by –READ– and –DATA–
the –REM– statement

More complicated input

multiple values in –DATA– [all numeric]
multiple values in –DATA– [all string]
multiple values in –DATA– [mixed numeric and string]
multiple assignment by –INPUT– [numeric variables]
multiple assignment by –INPUT– [string variables]
multiple assignment by –INPUT– [mixed numeric and string]
multiple assignment by –READ– [numeric variables]
multiple assignment by –READ– [string variables]
multiple assignment by –READ– [mixed numeric and string]

Branching—program flow

unconditional branch (–GOTO–)
interrupt with ctrl-g

Boolean expressions

print boolean expression [relation of string literals]
print boolean expression [relation of numeric literals]
print boolean expression [relation of numeric literal and variable]
print boolean expression [relation of string literal and variable]
boolean operators [–AND–]
boolean operators [–OR–]
boolean operators [–NOT–]

IF statements—conditional branches

conditional branch [compare numeric variable with numeric literal]
conditional branch [compare numeric variable with expression]
conditional branch [compare two numeric variables]
conditional branch [compare string variable with string literal]
conditional branch [compare two string variables]
the –STOP– statement

Hand-made loops—iteration

conditional branch [compare counter with numeric literal]
conditional branch [compare counter with numeric variable]
initialize counter variable with a literal value
initialize counter variable with the value of a variable
increment (add to) the value of a counter variable
decrement (subtract from) the value of a counter variable

Using loops to accumulate

accumulate successive values into numeric variable
accumulate successive values into string variable
calculating complex expressions [numeric literal and variable]
initialize numeric variable (not counter) to literal value
initialize numeric variable (not counter) to value of a variable
initialize string variable to literal value
initialize string variable to value of a variable

BASIC functionals

the –INT– function
the –RND– function
the –SQR– function

FOR. .NEXT loops

FOR. .NEXT loops with literal as final value of index
FOR. .NEXT loops with variable as final value of index
FOR. .NEXT loops with positive step size other than 1
FOR. .NEXT loops with negative step size

Arrays

assign element of string array variable by –INPUT–
assign element of numeric array variable by –INPUT–
assign element of numeric array variable [value is also a variable]
the –DIM– statement
string array using numeric variable as index
print value of an element of a string array variable
numeric array using numeric variable as index
print value of an element of a numeric array variable

Future extensions to the curriculum

nesting loops (one loop inside another)
subroutines (–GOSUB– and friends)