DOCUMENT RESUME

ED 168 560 IR 007 095

AUTHOR Dageforde, Mary L.; Beard, Marney H.
TITLE The BASIC Instructional Program: Supervisor's Manual.
INSTITUTION Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY Navy Personnel Research and Development Center, San Diego, Calif.
REPORT NO NPRDC-TN-78-10
PUB DATE Apr 78
CONTRACT N-00123-76-C-1543
NOTE 45p.; For related documents, see IR 007 092-096

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Computer Based Laboratories; Computer Managed Instruction; Computer Programs; *Computer Science Education; Higher Education; Input Output; *Instructional Programs; Programing; *Programing Languages; *Supervisors; Tutorial Programs

ABSTRACT

This manual for supervisory instructors documents the goals, methods, and operation of the BASIC Instructional Program (BIP), an interactive problem-solving laboratory that teaches elementary programming in the BASIC language. The first two sections describe features of BIP that may be of interest to the supervisor, especially the individualized task-selection algorithm and the Curriculum Information Network (CIN) which stores the relationships among elements of the author-written course material. The remaining sections describe in detail how to supervise the operation of BIP, including (1) the creation and use of certain files that may be written to during BIP execution (e.g., when there is an error in a model solution or when a student disagrees with the solution checker); (2) how to add and drop students from the course; (3) details of adding new tasks to the BIP curriculum and a description of the solution checker; and (4) how to obtain student progress reports. Appendices include a description of the technique groups and the skills in each, lists of syntax and execution errors, sample pages from the tasks file, and sample runs of BClass and Report Programs. (Author/CMV)

Technical Note 78-10

April 1978

THE BASIC INSTRUCTIONAL PROGRAM: SUPERVISOR'S MANUAL

Mary L. Dageforde
Marney H. Beard

Institute for Mathematical Studies in the Social Sciences
Stanford University
Palo Alto, California 94305

Reviewed by
John D. Ford, Jr.

Navy Personnel Research and Development Center
San Diego, California 92152

FOREWORD

This research and development was conducted in response to Navy Decision Coordinating Paper, Education and Training Development (NDCP Z0108-PN) under subproject Z0108-PN.32, Advanced Computer-Based Systems for Instructional Dialogue, and the sponsorship of the Director, Naval Education and Training (OP-99). The overall objective of the subproject is to develop and evaluate advanced techniques of individualized instruction.

This report is one in a series of six reports dealing with the BASIC (Beginner's All-Purpose Symbolic Instruction Code) Instructional Program (BIP), which is a "tutorial" programming laboratory designed for the student who has had no previous training in programming. The first report, NPRDC Special Report 77-2 (Note 1) was produced as a manual for students, and this report, as a manual for supervisors in charge of the BIP system. The others concern the conversion of the BASIC program into the MAINSAIL language (Note 2), system documentation (Note 3), conversion of the student manual into the MAINSAIL language (Note 4), and curriculum information networks for computer-assisted instruction (Beard, Barr, Gould, & Westcourt, 1978).

The work was performed under Contract N00123-76-C-1543 to Stanford University. The contract monitors were Dr. John D. Fletcher and Dr. James D. Hollan.

J. J. CLARKIN
Commanding Officer

SUMMARY

The BASIC Instructional Program (BIP) is an interactive problem-solving laboratory that teaches elementary programming in the BASIC language. This manual documents the goals, methods, and operation of BIP for supervisory instructors.

The first two sections describe features of BIP that may be of interest to the supervisor, especially the individualized task-selection algorithm and the Curriculum Information Network (CIN), which stores the relationships among elements of the author-written course material.

The remaining sections describe in detail all the necessary information on how to supervise the operation of BIP. Section 3 describes the creation and use of certain files that may be written to during BIP execution (e.g., when there is an error in a model solution or when a student disagrees with the solution checker). Section 4 tells how to add and drop students from the course. Section 5 provides details of adding new tasks to the BIP curriculum and a description of the solution checker. Finally, Section 6 tells how to obtain student progress reports.

CONTENTS

LIST OF FIGURES

SECTION 1. INTRODUCTION

The BASIC Instructional Program (BIP) is a stand-alone, fully self-contained course in BASIC programming at the high school/college level (Barr, Beard, & Atkinson, 1976). It is an interactive problem-solving laboratory that offers tutorial assistance to students in solving introductory programming problems. These problems are presented in an individualized sequence based on (1) a representation of the structure of the curriculum and (2) a model of the student's state of knowledge.

The goal of the tutorial laboratory is informative interaction with the student, which is provided by an instructional BASIC interpreter, information on BASIC syntax cross-referenced with the student manual, and debugging aids. The system also has access, through the Curriculum Information Network (see Section 2.2), to features that the student may use to help him complete his current problem. These features include hints (additional information about the task) and a stored solution program that can itself be executed.

This manual documents the goals, methods, and operation of the BASIC Instructional Program for supervisory instructors. It tells how to set up new students to add special curriculum, and to obtain student progress reports, and describes the goals and details of individualized task selection.

SECTION 2. THE BIP CURRICULUM

2.1 Curriculum Goals

Prior experience with computer-assisted instruction (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, they have little confidence in their own abilities to confront problems involving numeric manipulation. 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. (You may, of course, add such tasks if your student group is more mathematically oriented.)

The curriculum is designed to give the students practice and instruction in developing interactive programs in order to expose them to uses of the computer with which they may be unfamiliar. The emphasis is on programs that are engaging and entertaining, and that can be used by other people. While writing each program, the student keeps in mind a hypothetical user--a person who will use the program for his or her own purposes and to whom the performance of the program must be intelligible. Additional demands for clarity and organization are forced by interactive programming, the increased noticeability of "bugs," and the added motivational effects.

Numerous texts were examined as possible sources for programming principles that must be developed in an introductory course and for the problems that illustrate those principles. Ideas were incorporated from (1) general computer science textbooks (Forsythe, Keenan, Organick, & Sternberg, 1969), (2) the notes for an introductory programming course that were oriented toward the ALGOL language but easily generalizable (Floyd, 1971), and (3) books and notes dealing specifically with BASIC (Albrecht, Finkel, & Brown, 1973; Coan, 1970; Kemeny & Kurtz, 1971; Nolan, 1969; Wiener & Ross, 1972). 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 these problems, the difficulties stem from the demands of logical program organization rather than from the complexities of the prerequisite mathematics.

2.2 The Curriculum Information Network

The Curriculum Information Network (CIN) 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 that the author would have used implicitly in determining his "branching" schemes. Thus, it allows meaningful modelling of the student's progress along the lines of his or her developing skills (not just a 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 attaining these skills, and choose the next task with an appropriate group of new skills. The CIN introduces an intermediate step between the time when the student's history is recorded and his next problem is selected: thus, it becomes a model of the student's state of knowledge, since it has an estimate of his ability in the relevant skills, not just a record of 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, multiplied 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.

2.3 Individualized Task Selection

In BIP, our curriculum goals are the mastery of certain programming techniques, such as simple output; using loops, conditional branches, and arrays; assignment to variables, etc. The techniques are linked in a linear order, each having but one "prerequisite" (i.e., the previous technique), based on dependence and increasing program complexity. They are interpreted or described by the list of skills that are required in the solution program. The skills themselves, which are very specific descriptions of particular programming behaviors like "print a string literal" or "initialize a counter variable" are not themselves hierarchically ordered. Appendix A provides a list of the techniques and the skills grouped within those techniques. 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 1 shows a simplified portion of the curriculum network, and demonstrates the relationship among the tasks, skills, and techniques.

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 students are expected to write their own BASIC program to solve it. While developing a BASIC program for each task, the students are directed to appropriate sections of the student manual for full explanations of BASIC statements, programming structures, etc. They are also encouraged to use the TRACE debugging facility and various "help" options such as HINTs (additional information about the task, or the steps required to reach a solution) and the DEMO, which executes the model solution.

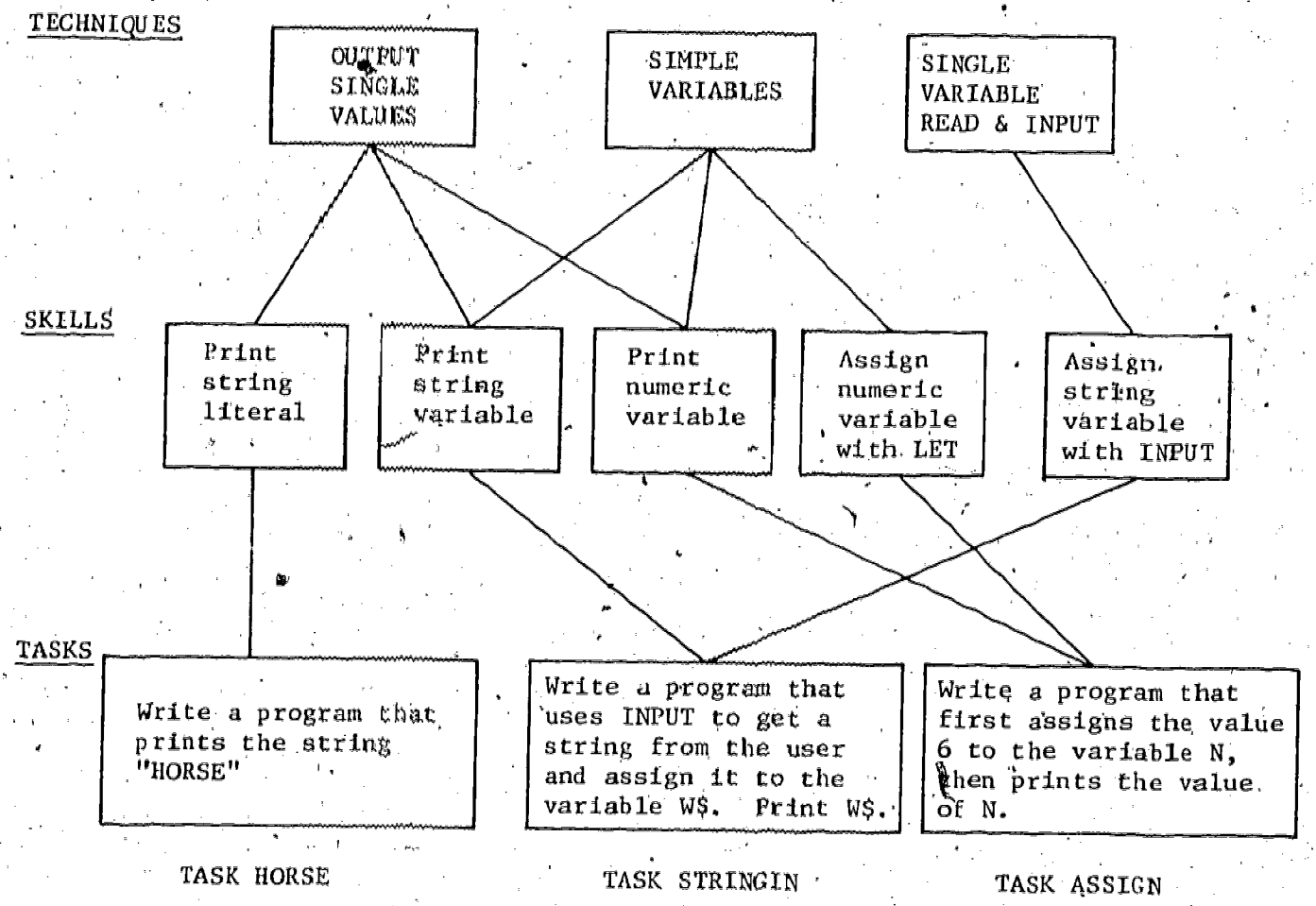

Figure 1. A simplified portion of the curriculum network.

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

The sequence of events that occur as the student works on a task is shown in Figure 2. 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 feels that 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 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 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 Figure 3. 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.

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, 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. For each successful use of a skill, both counters are incremented. If the student quits a task requiring a particular skill, the first counter is decremented; if he requests more work on a skill, the second counter is zeroed. Any such "not yet mastered" skills are put into the MUST set. Eventually the program will seek to find a task that uses some of these "must" skills.

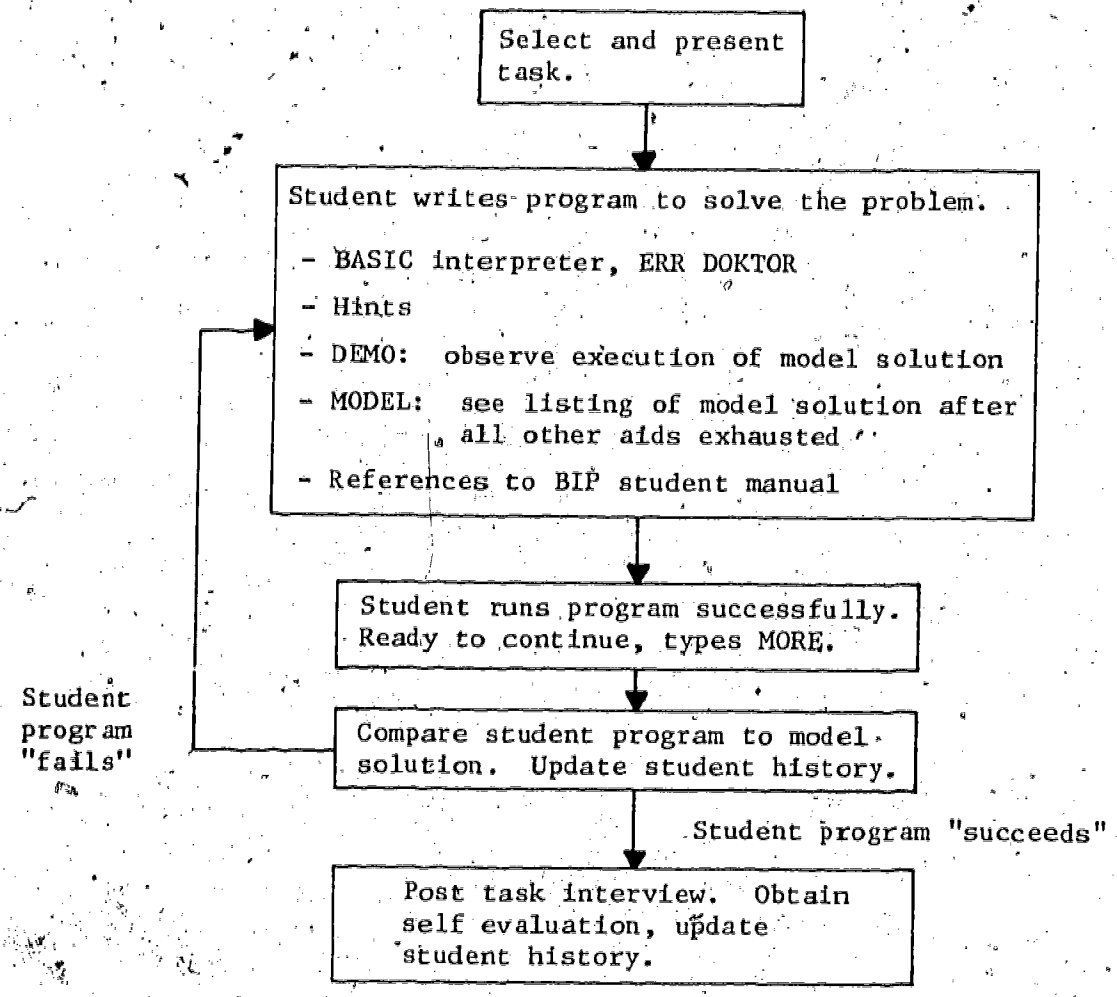

Figure 2. Working through a task.

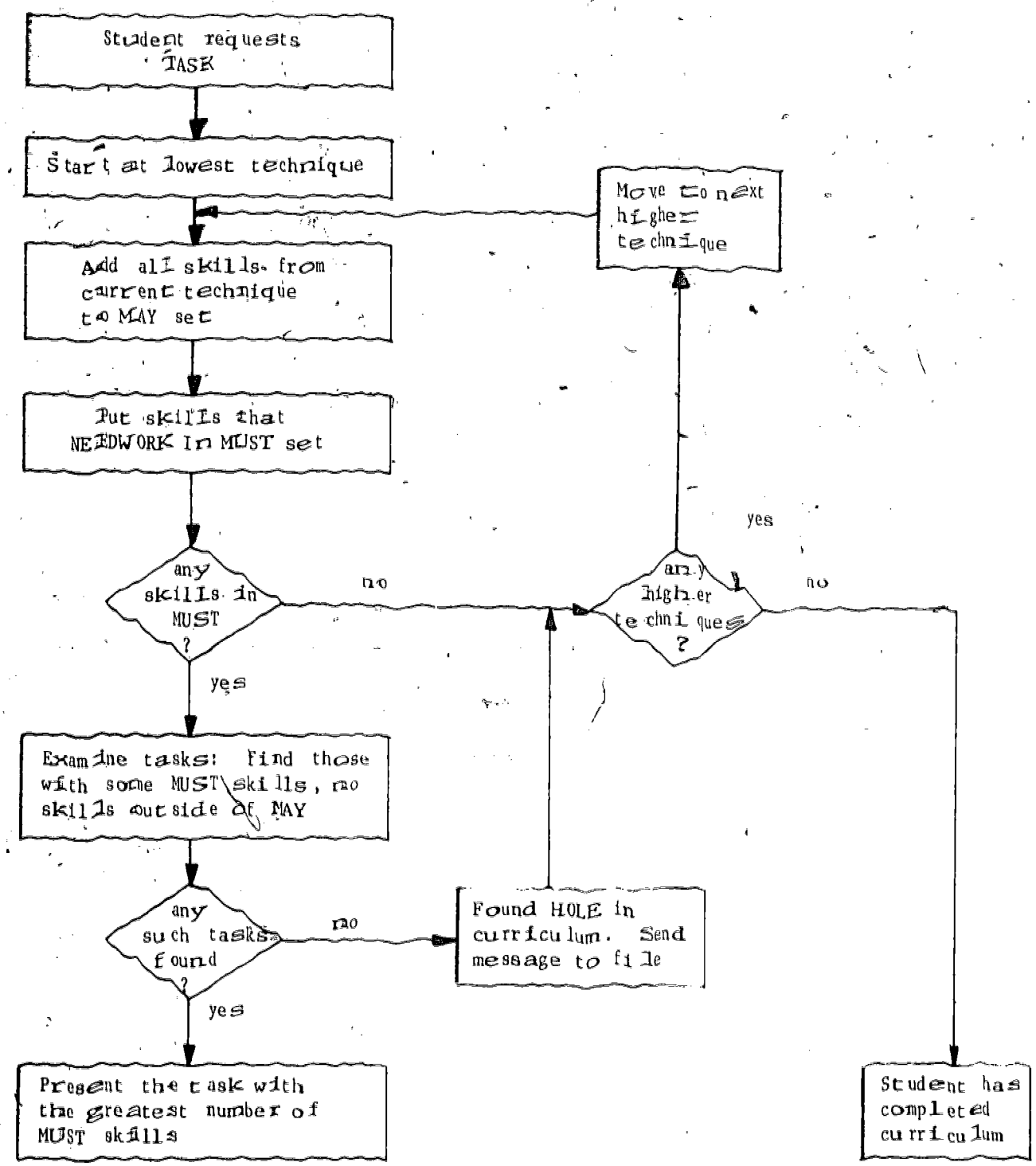

Figure 3. Selecting the next task.

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 require (1) at least one of the MUST skills, and (2) 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 Figure 1, assuming that the student had not yet met the criterion on the skills shown, the first task to be presented would be FORSE, because its skill lies in the earliest technique, and would constitute the first MUST set. Task ASSIGN would be presented next, since its skills come from the next higher technique; STRINGIN would be presented last of these three.

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 writing a message to the HOLES FILE (see Section 3.4) describing the nature of the missing task (e.g., 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.

SECTION 3. CREATION AND USE OF FILES

There are three groups of files with which you should be familiar. The first group includes those that may be written to during BIP execution; namely, the MODER, ARGUE, FIX, and HOLES files. The second group consists of the WHO file, which lists the students enrolled in the course and student history files; and the third is the text file TASKS, which includes the description, model solution, hints, skills, etc. for each task. The first group is described in this section; and the second and third groups, in Sections 4 and 5 respectively.

You should create and initialize the MODER, ARGUE, FIX, and HOLES files before any students begin by running the program FMAKE. This program asks you four questions: "Do you want to create the MODER . . . ARGUE . . . FIX . . . HOLES file?" to which you will respond with a "Y" or an "N." The file(s) is created and initiated when you answer "Y."

All four files are created the first time you run FMAKE. You should check these files occasionally (when students are not running BIP). However, you must not modify them in any way! They are random-access files with a pointer to the end of the file, where BIP appends further information. If you add or delete any characters, this end-of-file pointer will be incorrect, and you may lose information. What you can do is rename one (or all) of these files, create a new file (using FMAKE), and then do whatever you want (e.g., delete messages you've taken care of) with the "old" (renamed) version.

3.1 The MODER File

BIP writes a message to the MODER file whenever an error is found in a model solution.

Example:

```
48 MARILYN SMITH
Model error in:  CALCULATOR
Execution error number:  5
```

Each message will give the nature of the error (i.e., syntax or execution) and its error number. See Appendix B for lists of syntax errors and execution errors.

You should fix the appropriate model solution in the TASKS file (when no students are running BIP) and then (as is always necessary after you make a change to the TASKS file) run the program TODATA to create a new INIT file.

3.2 The ARGUE File

BIP writes a message to the ARGUE file whenever a student disagrees with the solution checker.

Example:

```
99      SUSAN JONES
Argue with task:  INIF
Attempt # 1
Program:
  10 PRINT "TYPE A NUMBER BETWEEN 1 AND 4"
  20 INPUT X
  30 IF X = 1 THEN 70
  40 IF X = 2 THEN 80
  50 IF X = 3 THEN 90
  60 IF X = 4 THEN 100
  70 PRINT "YOU TYPED A 1!"
  80 PRINT "YOU TYPED A 2!"
  90 PRINT "YOU TYPED A 3!"
  100 PRINT "YOU TYPED A 4!"
  999 END
```

Details:

```
Too much output.

YOU TYPED A 2!        t
```

Look at the unaccepted student programs shown on the ARGUE file. The "Details" part of the message lists each line of output the solution checker was looking for, followed by a "t" (if the student's program produced that output) or an "f" (if it didn't). Other unacceptable or missing output (as described in Section 5.2) is also noted under "Details."

An examination of the listing of the student's program, the details from the solution checker, a knowledge of what the program was supposed to do (you should consult a listing of the TASKS file), and of how the solution checker works (see Section 5.2) should make it clear to you why the student's program was not accepted. You may wish to contact the student with an explanation.

3.3 The FIX File

The FIX file is composed of messages sent to you by the students.

Example:

```
55      MARK JOHNSON
Note:
  Terminal 121 is not working.
```

3.4 The HOLES File

The HOLES file notifies you of "holes" in BIP's curriculum.

Example:

```
28 JOHN ADAMS
Technique is 5
Set that failed was:  must
Skills in must set are
    55 74
```

As explained in Section 2.3, tasks considered as a possible next task for the student must require at least one of the MUST skills and no skills outside of the MAY set. Thus, 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. In this case, BIP's task-selection algorithm will examine the next higher technique, generate new, expanded MUST and MAY sets, and continue the search for an appropriate task.

It is not necessary for you to take any action. "Holes" will be fairly common, and don't necessarily adversely affect BIP's task selection. However, if you wish to expand BIP's curriculum, the information in the HOLES file may guide you in designing additional tasks.

SECTION 4. ADDING AND DROPPING STUDENTS

To start new students in the BIP course, assign and tell them their student numbers (any numbers less than or equal to 999), create their history files, add their numbers and names to the WHO file (explained below), give them student manuals, and tell them how to start BIP running. The sign on procedure is explained in the student manual.

Two files are involved in the adding and dropping of each BIP student: (1) the text file "WHO," which lists each student's number, name, and (optionally) sex, and (2) the individual student's personal history file, which is used to store information about his current status (what task he is currently working on, how many tasks completed so far, etc.), and past performance on tasks and skills.

4.1 The WHO File

The WHO file has one line of information for each student currently enrolled in the course, and the end of the file is signalled by the word "END." You, the supervisor, have control over the WHO file: you create it before any students try to sign on, and you add and delete students from the course by adding and deleting lines from the file. It is not necessary for the lines to be arranged numerically, by student number, although you may wish to keep it that way. The format for each student line is:

<student #><tab><first name><space><last name><tab><F or M>, where the F or M sex indication is optional and each tab and space could actually be any number of tabs and/or spaces. Here is an example of what the WHO file would look like if there were three students in the course, assigned (by you) numbers 99-101:

```
99    SUSAN JONES    F
100   MARK SMITH     M
101   JANE ADAMS     F
END
```

A student signs on by telling BIP his number and first name. BIP then checks to make sure that he is enrolled in the course by searching the WHO file for a line with that number and name. If no such number is found, or if the name provided by the student does not match the name in the appropriate line, BIP tells the student that the number and/or name are incorrect, and logs him off. The student must then ask you for the correct information.

4.2 History Files

When a student is added to the WHO file, his history file must be created and initialized. Run the program NEWHST, which asks which student(s) you wish to create a history for. Type a list of student numbers, separated by commas and/or dashes, and NEWHST will create and initialize their histories.

The name of the student's history file consists of the letters "HST" followed by his number. Thus the above three students' history files would be "HST99," "HST100," and "HST101." History files are data files that you should never try to read or edit. Their information is stored in a compact form identical to the internal representation within the computer, not readable characters.

4.3 Dropping Students

To drop students from the course, simply delete their history files and their lines in the WHO file. Students that are added subsequently may, of course, be given numbers formerly assigned to "dropped" students, and set up as usual.

SECTION 5. ADDING NEW TASKS

The curriculum for BIP is contained in a text file called TASKS. The file is read by two different programs--TODATA and INIT.

a. The TODATA program compresses certain essential information from TASKS and writes it onto a data file called INIT, which is read when the student signs on. The INIT data is used to initialize the curriculum data structure.

b. Throughout a student's session, BIP reads from the TASKS file to access the text of the current task, its hints and models, etc. The pointers that were initialized from the INIT data give BIP efficient access to the text in the TASKS file.

The format of TASKS is therefore somewhat rigid. Figure 4 and the following information describe the format and necessary contents of the file. If no restriction is specified, none exists; for example, there is no limit imposed on the number of skills allowed in any task.

5.1 Details of Task Information Format

a. New tasks may be added either between current tasks or at the end of the TASKS file. The FIRST page of the TASKS file (which lists the skills for each technique and which has all the information for the first task, Greenflag) should remain the first page (see Appendix C, page C-1).

b. The tasks need not appear in numeric order.

c. A task's name and its number must be separated by a single space or a single tab. The task name may contain only letters, digits, and periods (no other punctuation or spaces).

d. The order in which the task information is given is important. For each task, the following information groups are REQUIRED:

*main (or *moreTask), *text, *model, and *skills.

All the other information groups are optional. For example, there may be no hints for a certain task, in which case there simply is no *hint information.

e. Tasks are either MAIN tasks or MORETASKs. A MORETASK is usually an extension of its MAIN. It requires a minor modification of the program written for the MAIN task, and is presented automatically after the student completes the MAIN. When the student requests a task, BIP selects from the MAIN tasks only, and most of the curriculum consists of MAINs. A MAIN may have any number of MORETASKS, which will be presented in the order in which they appear in the TASKS file: MAIN task first, followed by its MORETASKS. Appendix C shows an example of information for a MAIN task (BACK) and its moreTASK (Back.1).

```
*main (or *moreTask)
<task name>  <tasknumber>

*text
<task description>

*model
<code line for verifier -- see Section 5.2>
<model solution>

*hint
Hint #1
<hint #1>
*
Hint #2
<hint #2>
 ...
*
#

*reqOps
<required operators, separated by commas>

*disOps
<disabled operators, separated by commas>

*reqFns
<required functions, separated by commas>

*disFns
<disabled functions, separated by commas>

*skills
<list of skill numbers, IN NUMERIC ORDER, separated by commas>

<page mark -- optional>
<blank line>


*main (or *moreTask)
 ...

<information for subsequent tasks>

 ...
*end
```

Figure 4. Format for task information in file TASKS.

f. A task may have any number of hints (though four seem to be as many as students are likely to benefit from). Each hint must begin with the work "Hint" and its number, and each is terminated by an asterisk on the line following the hint text. The entire group of hints is followed by a hash mark (#) on the line following the last asterisk.

g. The operators (BASIC statements) that can be required or disabled are:

LET	INPUT	GOTO	IF
REM	DIM	STOP	FOR
NEXT	GOSUB	RETURN	READ
DATA	REOPEN	BEGINSUB	ENDSUB

Do not require or disable PRINT or END because they are always automatically required for each task. If more than one operator is required or disabled, they must be separated by commas.

h. The functions that can be required or disabled are:

INT	RND	SQR

If more than one appears, they must be separated by commas.

i. The list of numbers following the "*skills" line must be given in numeric order, separated by commas. The list of skills is the description of the task that BIP uses when it presents the student's next problem, so the skills should be carefully selected to reflect the requirements of the task. The skills' numbers and meanings are included in Appendix A.

j. If a page mark is used to separate one task from the next, it must be followed by at least one blank line before the "*main" or "*moreTask" line.

k. "*end" must be the last thing in the TASKS file. If you add new tasks to the end of the file, don't forget to delete the "*end," add the new tasks, and then replace "*end" at the end of the file.

l. Many of BIP's tasks require programs that use INPUT to interact with a hypothetical user. When the solution checker evaluates a student's program that includes INPUT, it executes the program with a specified set of values for the input variables. The model solution given in the TASKS file must include a REM statement for each of the input variables used in the model program, describing the use of that variable. The format of the REM statement is

<line #> REM <variable name> IS: <description>.

(For example: 10 REM X IS: THE USER'S FIRST ADDEND
20 REM Y IS: THE USER'S SECOND ADDEND)

Section 5.2 explains the coding of the model solution in detail.

m. The TASKS file is a text file, which makes it easy to read and change. Since BIP depends on the INIT data to provide accurate pointers into the text file, the TODATA program must be run whenever a change is made to TASKS. Any change, no matter how small (e.g., correcting a misspelled word), means that TODATA must be run.

5.2 How the Solution Checker Works

When the student types "MORE," his program is checked in a few different ways. The actual solution checker procedure is not even called if the student has not RUN the program since the last time he changed it, or if any of the required operators is missing. Once these two tests are met, the solution checker evaluates the student's program by comparing its output to that of the model solution. The model is executed first (invisibly), and every line that it prints is stored in an array. As the student's program is executed (also invisibly), each line of its output is compared to the stored output from the model. If that line matches an element in the model-output array, a flag is set. If, after the student's program has completed execution, any of the elements in the model-output array have not been matched, he is told that his program "doesn't seem to solve the problem," and the unmatched elements are listed for him. If all the model outputs have been matched, he is told that his program "looks ok," and the post-task interview is presented.

In order to allow as much flexibility as possible in BIP's curriculum, the solution checker involves a number of complications. These fall into three groups: (1) determining whether the student's program for a given task is to be checked at all, (2) specifying how much of the output the checker should store, and (3) specifying the values that the model and the student's program will be given as input.

5.2.1 Whether to Check

In the TASKS file, the line "*model" is followed by a "coding line" that gives the necessary information. If the first character on that line (perhaps the only character) is a semicolon, then the solution checker will evaluate the program. Any other character tells the checker to assume that the student's program is acceptable. The character currently used in our TASKS file is the number 9.

5.2.2 How Much to Store and Compare

Unless otherwise specified in the coding, the checker will ignore all string and numeric constants. That is, as the model and the student's program are executed, any expression containing a quoted string or a numeric constant will not be stored for comparison. By ignoring string constants, BIP allows the student to have his program print messages of his choice, rather than forcing him to make his program say exactly the same things as the model solution. (For example, where the model might print "TYPE YOUR NAME," the student might prefer to say "WHAT IS YOUR NAME?"; as long as his program performs equivalent computations, he should not be penalized for the

nonessential aspects of the program.) Numeric constants in the output are ignored because they are rarely useful, and students should not be allowed to think (as the data indicate many have, in the past) that they should do the computation and simply have BASIC print the already-calculated result.

For those tasks in which string or numeric constants are essential parts of the output (e.g., those early tasks that require a program that prints a specified value, for illustrative purposes), the character "s" or "n" must be given in the coding line. For example, the model for a very simple task looks like this:

```
*model
;s
        10 PRINT "SCHOOL"
        99 END
```

For this task, the only output from the model that can be matched by the student's program is the string constant "SCHOOL"; since the "s" appears in the coding line, the solution checker will store "SCHOOL" when it is printed by the model, and when it is printed by the student's program.

Similarly, the model solution for another easy task is:

```
*model
;n
10 PRINT 3
20 PRINT 3.14
99 END
```

Again, since the purpose of the task is to illustrate printing numeric constants, it is necessary to store the numeric constant output from both the model and the student's program. Therefore, the coding line includes the "n" flag.

Finally, two rarely used options may be given. An "e" on the coding line specifies that the student's program must produce exactly the same number of lines of output. If his program prints more than the model solution, it will not be accepted. A "v" on the coding line specifies that all spaces that would appear at the beginning of a line of output are ignored. Spaces within the line are preserved.

5.2.3 Specifying INPUT Variables and Values

The solution checker evaluates the student's program by executing it (and the model solution) invisibly; therefore, no interaction with the student or any other "user" of the program takes place. To evaluate solutions to tasks that require an interactive program capable of dealing with input from a user, the solution checker must have access to two kinds of information: the input values to be invisibly assigned, and the names of the variables used in the student's program. As described earlier, each variable in the model solution to be assigned via INPUT must be described in a specially formatted REM statement. In addition, the coding line must include a list of the value(s) to be assigned to each input variable. A simple example is:

```
*model
; 2
1 REM X IS:  THE USER'S NUMBER
10 PRINT "TYPE A NUMBER"
20 INPUT X
30 PRINT "THAT NUMBER WAS "; X
99 END
```

The semicolon indicates that the student's program for this task is to be checked. The "2" in this example is the value that will be assigned to the input variable during the invisible execution of the model and the student's program. Before the solution checker begins, it asks the student for the input variable his program uses; here, it would ask "What variable do you use for THE USER's NUMBER?" . The description ("the user's number") comes from the REM statement in the model solution. The advantage of these complications is that they allow the student to use whatever variables he chooses, rather than forcing on him the same variables used in the model solution.

If more than one input variable is required in the task, their values are given on the coding line separated by spaces and colons, as in the beginning of the model solution for task CALCULATOR:

```
*model
; 3 : 15 : 4
1 REM X IS:  THE CODE INDICATING WHICH OPERATION
2 REM M IS:  THE USER'S FIRST NUMBER
3 REM N IS:  THE USER'S SECOND NUMBER
```

During invisible execution of the INPUT statements in the model, X will get the value 3; M, 15; and N, 4. If the student chose to use the variables P, Q, and R instead X, M, and N respectively, then his INPUT P statement would give the value 3 to his variable P, his INPUT Q would assign 15 to Q, etc. (The REM statements are not required in the student's program; they appear in the model so that the solution checker can ask "What variable do you use for . . . " and as additional clarification for the student when the model is shown to him.)

Finally, if the requirements of the task are such that a variable is to be given a value via INPUT more than once (e.g., within a loop), the coding line must include the list of values to be assigned to each such variable. The coding line for CALCULATOR.1, which prints the results of different arithmetic operations until the user types 0, looks like this:

```
*model
; 1 2 3 0 : 10 20 30 : 4 5 6
```

which means that X (or whatever variable the student used for "the code indicating which operation") will be assigned the value 1 the first time INPUT X is executed, 2 the second time, etc. M ("the user's first number") will be assigned 10 when INPUT M is first executed, 20 when INPUT M is executed the second time, etc.

Some of BIP's tasks require the student to generate random numbers. For the purposes of the solution checker, both the model solution and the student's program must use the same "random" number if they are to produce comparable output. Therefore, the coding line must include the "r" flag and specified values that will be used whenever the RND function is executed during solution checking. The model solution for task GUESS.1 is:

```
*model
;r .600 .010 : 17 2 16 1 : "YES" "NO"
1 REM Y IS:  THE USER'S GUESS
2 REM A$ IS:  WHETHER OR NOT TO REPEAT THE GAME
10 PRINT "TYPE A NUMBER BETWEEN 1 AND 25."
20 X = INT (RND*25 + 1)
25 G = 0
30 PRINT "TYPE YOUR GUESS."
40 INPUT Y
45 G = G + 1
50 IF Y = X THEN 200
60 IF Y < X THEN 100
70 REM !! NOW Y MUST BE GREATER THAN X
80 PRINT "HIGH"
90 GOTO 30
100 PRINT "LOW"
110 GOTO 30
200 PRINT "RIGHT IN "; G; " TRIES"
210 PRINT "TYPE YES TO PLAY AGAIN"
220 INPUT A$
230 IF A$ = "YES" THEN 10
999 END
```

The "r" tells the solution checker that random value(s) follow. The first time that the RND function in either program is executed during evaluation, the value .600 will be returned. Since the statement that uses RND is

```
20 X = INT (RND*25 + 1)
```

when the value of X (the random integer "picked" by the program) will be INT (.60 * 25 + 1) = 16. The values 17, 2, and 16 will be assigned to Y in turn as the INPUT Y statement is executed repeatedly, invisibly simulating the user's guesses. After the assignment of 16, the values of X and Y will be equal, so the program will execute the INPUT A$ statement, where the value "YES" will be invisibly assigned, causing the program to "pick" another number. This time .010 will be returned as the value of RND, resulting in X being assigned the value 1. The next available value for Y is 1; the "user's guess" equals the "random integer," and the value "NO" is invisibly assigned to A$. Execution terminates.

It is clearly important for the task author to know exactly how he wants the model solution and the student's program to execute when he specifies the input and random values on the coding line. The main purpose of the values given is, of course, to test the student's program adequately. The author must at the same time ensure that the values given cause the model solution to execute without error, and produce output that can be accurately compared to an acceptable student solution.

To summarize the format requirements of the coding line:

Options are one or more of the following:

e, n, r, s, v

If any are used, they must appear in alphabetic order. They are not preceded by a space.

Value lists are sequences of numeric or string constants used as values for INPUT variables or as values to be returned by the RND function. A value list always begins with a space, and a space is used within the list to separate one value from another. If anything follows a value list, the end of the list is marked by a colon. If values are specified for more than one variable, a separate value list (beginning with a space, terminated by a colon) must be given for each variable.

The "r" option is always followed by its value list. (The other options do not need values.) The value list used for the INPUT variables always follows the options, if any. To illustrate a combination, consider the coding for task GUESS:

```
;r .545 :s 19 10 14
```

The value .545 will be returned when the RND function is executed; string constant output will be stored for comparison; and the values that will be assigned to the input variable (described in a REM statement) are 19, 10, and 14, in that order, each successive time the INPUT statement is executed. If the author wished to allow extra leading spaces in the output (i.e., allow the student to print leading spaces whether or not the model does so), the "v" option would be added, and the coding line would be

```
;r .545 :sv 19 10 14.
```

And if the author wanted to require the student to print only as much as the model prints, the "e" option would be added:

```
;er .545 :sv 19 10 14.
```

SECTION 6. STUDENT PROGRESS REPORTS

The BCLASS and REPORT programs provide information on student progress.

The BCLASS program will give you a tabular summary of the progress of any group of students you specify. It will, for all students specified, print their name and number, the number of tasks they have completed, the number of sessions and hours they spent running BIP, the name of the last task they were in, and their last sign on date.

The REPORT program will give you a more detailed summary of individual student progress on the curriculum. The options available include: which student(s) you want a report for, where you want the output (on a file or written to your terminal), whether you want information about each student's last task only or about all the tasks he has completed, and whether or not you want an explanation of the abbreviations used in the report.

Sample runs of the BCLASS and the REPORT programs are provided by Appendix D.

REFERENCES

Albrecht, R. L., Finkel, L., & Brown, J. R. BASIC. New York: Wiley, 1973.

Barr, A., Beard, M., & Atkinson, R. C. The computer as a tutorial laboratory: The Stanford BIP project. International Journal of Man-Machine Studies, 1976, 8, 567-596.

Beard, M., Barr, A. V., Gould, L., & Westcourt, K. Curriculum information networks for computer-assisted instruction (NPRDC TR 78-18). San Diego: Navy Personnel Research and Development Center, April 1978.

Coan, J. S. BASIC. New York: Hayden Book, 1970.

Floyd, R. W. Notes on programming and the ALGOL W language. Stanford, CA: Computer Science Department, Stanford University, 1971.

Forsythe, A. I., Keenan, T. A., Organick, E. I., & Sternberg, W. Computer science: A first course. New York: Wiley, 1969.

Kemeny, J. G., & Kurtz, T. E. BASIC programming, (2nd ed.). New York: Wiley, 1971.

Nolan, R. L. Introduction to computing through the BASIC language. New York: Holt, Rinehart, and Winston, 1969.

Wiener, H., & Ross, B. BASIC workbook. Berkeley, CA: Lawrence Hall of Science, University of California, 1972.

REFERENCE NOTES

1. Beard, M. H., & Barr, A. V. The BASIC instructional program: student manual (NPRDC Special Rep. 77-2). San Diego: Navy Personnel Research and Development Center, October 1976.

2. Dageforde, M. L. The BASIC instructional program: Conversion into MAINSAIL language (NPRDC Tech. Note 78-11). San Diego: Navy Personnel Research and Development Center, April 1978.

3. Dageforde, M. L. The BASIC instructional program: System documentation (NPRDC Tech. Note 78-12). San Diego: Navy Personnel Research and Development Center, April 1978.

4. Dageforde, M. L., Beard, M. H., & Barr, A. V. The BASIC instructional program student manual: MAINSAIL conversion (NPRDC Tech. Note 78-9). San Diego: Navy Personnel Research and Development Center, April 1978.

APPENDIX A

THE TECHNIQUE GROUPS AND THE SKILLS

THE TECHNIQUE GROUPS AND THE SKILLS

Technique 1. Simple output--first programs.

1 Print numeric literal
2 Print string literal
5 Print numeric expression [operation on literals]
8 Print string expression [concatanation of literals]

Technique 2. Variables--assignment.

3 Print value of numeric variable
4 Print value of string variable
6 Print numeric expression [operation on variables]
7 Print numeric expression [operation on literals and variables]
9 Print string expression [concatanation of variables]
10 Print string expression [concatanation of variable and literal]
11 Assign value to a numeric variable [literal value]
12 Assign value to a string variable [literal value]

Technique 3. More complicated assignment.

34 Assign to a string variable [value of an expression]
35 Assign to a numeric variable [value of an expression]
69 Re-assignment of string variable (using its own value)
70 Re-assignment of numeric variable (using its own value)
82 Assign to numeric variable the value of another variable
83 Assign to string variable the value of another variable

Technique 4. More complicated output.

28 Multiple print [string literal, numeric variable]
29 Multiple print [string literal, numeric variable expression]
30 Multiple print [string literal, string variable]
74 Multiple print [string literal, string variable expression]

Technique 5. Interactive programs--INPUT from user--using DATA.

13 Assign numeric variable by -INPUT-
14 Assign string variable by -INPUT-
15 Assign numeric variable by -READ- and -DATA-
16 Assign string variable by -READ- and -DATA-
55 The REM statement

Technique 6. More complicated input.

17 Multiple values in -DATA- [all numeric]
18 Multiple values in -DATA- [all string]
19 Multiple values in -DATA- [mixed numeric and string]
22 Multiple assignment by -INPUT- [numeric variables]
23 Multiple assignment by -INPUT- [string variables]
24 Multiple assignment by -INPUT- [mixed numeric and string]
25 Multiple assignment by -READ- [numeric]
26 Multiple assignment by -READ- [string]
27 Multiple assignment by -READ- [mixed numeric and string]

Technique 7. Branching--program flow.

36 Unconditional branch (-GOTO-)
37 Interrupt execution

Technique 8. Boolean expressions.

38 Print Boolean expression [relation of string literals]
39 Print Boolean expression [relation of numeric literals]
40 Print Boolean expression [relation of numeric literal and variable]
41 Print Boolean expression [relation of string literal and variable]
75 Boolean operator -AND-
76 Boolean operator -OR-
77 Boolean operator -NOT-

Technique 9. IF statements--conditional standards.

42 Conditional branch [compare numeric variable with numeric literal]
43 Conditional branch [compare numeric variable with expression]
46 Conditional branch [compare two numeric variables]
47 Conditional branch [compare string variable with string literal]
48 Conditional branch [compare two string variables]
59 The -STOP- statement

Technique 10. Hand-made loops--iteration.

44 Conditional branch [compare counter with numeric literal]
45 Conditional branch [compare counter with numeric variable]
49 Initialize counter variable with a literal value
50 Initialize counter variable with the value of a variable
53 Increment the value of a counter variable
54 Decrement the value of a counter variable

Technique 11. Using loops to accumulate.

51 Accumulate successive values into numeric variable
52 Accumulate successive values into string variable
71 Calculating complex expressions [numeric literal and variable]
78 Initialize numeric variable (not counter) to literal value
79 Initialize numeric variable (not counter) to value of a variable
80 Initialize string variable to literal value
81 Initialize string variable to the value of another variable

Technique 12. Using "dummy" value to signify end of data.

20 Dummy value in -DATA- statement [numeric]
21 Dummy value in -DATA- statement [string]

Technique 13. BASIC functionals.

56 The -INT- function
57 The -RND- function
58 The -SQR- function

Technique 14. FOR...NEXT loops.

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

Technique 16. Arrays.

31 Assign element of string array variable by -INPUT-
32 Assign element of numeric array variable by -INPUT-
33 Assign element of numeric array variable [value is also a variable]
60 The -DIM- statement
65 String array using numeric variable as index
66 Print value of an element of a string array variable
67 Numeric array using numeric variable as index
68 Print value of an element of a numeric array variable

Technique 16. Nesting loops (one loop inside another).

72 Nesting loops
73 Subroutines (-GOSUB- and friends)

APPENDIX B

LISTS OF SYNTAX AND EXECUTION ERRORS

LIST OF SYNTAX ERRORS

1. PARENTHESIS MISMATCH
2. ILLEGAL VARIABLE
3. MISPLACED + OR -
4. MISSING QUOTE MARKS (OR ILLEGAL FUNCTION CALL)
5. -RND- TAKES NO ARGUMENTS
6. MISSING ARGUMENT FOR -SQR-
7. MISSING ARGUMENT FOR -INT-
8. ILLEGAL STRING EXPRESSION
9. MISSING ARGUMENT FOR -LEN-
11. ILLEGAL LINE NUMBER
12. ILLEGAL EXPRESSION
13. NO TEXT ALLOWED AFTER -END-
14. NO TEXT ALLOWED AFTER -STOP-
15. JUNK AT THE END OF THE LINE
16. MISSING " " OR "=" IN A -LET-
17. LOOPING BRANCH TO THE SAME LINE
18. UNMATCHED QUOTE MARKS
19. MISSING OR ILLEGAL LINE NUMBER
20. SEMI-COLON IN A -READ- STATEMENT
21. SEMI-COLON IN AN -INPUT- STATEMENT
22. ILLEGAL VARIABLE FOR A -READ- OR -INPUT-
23. MISSING "THEN" IN AN -IF- STATEMENT
24. COMMA IN A -PRINT- STATEMENT
25. ILLEGAL EXPRESSION IN A -PRINT- STATEMENT
26. ILLEGAL COUNTER VARIABLE IN A -NEXT-
27. SEPARATION OF DATA WITH A SEMI-COLON
28. MISSING COMMA BETWEEN -DATA- ENTRIES
29. INCORRECT DATA
30. ILLEGAL COUNTER VARIABLE IN A -FOR-
31. MISSING "=" OR " " IN A -FOR- STATEMENT
32. MISSING A "TO" IN A -FOR- STATEMENT
33. ILLEGAL NAME FOR AN ARRAY VARIABLE
34. INCORRECT -DIM- STATEMENT
35. COMMA IN A -DIM- STATEMENT
36. MISSING BASIC STATEMENT
37. INCORRECT FUNCTION NAME
38. INCORRECT FUNCTION DEFINITION
39. INCORRECT PARAMETER NAME
40. ILLEGAL LINE NUMBER
41. TOO MANY LINES IN PROGRAM
42. ASSIGNMENT TO AN EXPRESSION
43. ILLEGAL BOOLEAN EXPRESSION
44. BIP COMMANDS ARE NOT LECAL FOLLOWING A LINE NUMBER
45. ILLEGAL BIP COMMAND
46. BASIC STATEMENTS MUST HAVE A LINE NUMBER
47. ILLEGAL CHARACTER

LIST OF EXECUTION ERRORS

1. DIVISION BY ZERO NOT ALLOWED
2. FUNCTION CALL WITHOUT A FUNCTION DEFINITION
3. RECURSIVE FUNCTION CALL
4. SQUARE ROOT OF A NEGATIVE NUMBER
5. VARIABLE WITHOUT A KNOWN VALUE
6. MISSING SUBSCRIPT FOR SUBSCRIPTED (ARRAY) VARIABLE
7. MISSING DIM STATEMENT FOR SUBSCRIPTED (ARRAY) VARIABLE
8. TOO FEW SUBSCRIPTS FOR THIS VARIABLE
9. IMPOSSIBLE SUBSTRING
10. NON-NUMERIC VALUE FOR NUMERIC VARIABLE
11. NO MORE DATA TO READ
12. DATA TYPE MISMATCH DURING READ
13. DIMENSION MUST BE GREATER THAN ZERO
14. RE-DIMENSIONING A SUBSCRIPTED VARIABLE DURING EXECUTION
15. NESTING OF FOR...NEXT LOOPS TOO DEEP
16. TOO MANY GOSUBS EXECUTED BEFORE EXECUTION OF A RETURN
17. RETURN WITHOUT MATCHING GOSUB
18. INDEX FOR SUBSCRIPTED(LIST) VARIABLE OUT OF DECLARED BOUNDS
19. TOO MANY SUBSCRIPTS
20. FUNCTION CALL WITH WRONG TYPE OF ARGUMENT
21. FUNCTION DEFINED TWICE IN THE PROGRAM

APPENDIX C

SAMPLE PAGES FROM THE TASKS FILE

SAMPLE PAGES FROM THE TASKS FILE

(The technique and skills list and GREENFLAG are on p.1 of the TASKS file, BACK is on p.26, BACK.1 on p.27, and ALPH on p. 49).

```
TECHNIQUE       SKILLS
1               1,2,5,8
2               3,4,6,7,9,10,11,12
3               34,35,69,70,82,83
4               28,29,30,74
5               13,14,15,16,55
6               17,18,19,22,23,24,25,26,27
7               36
8               38,39,40,41,75,76
9               42,43,46,47,48,59
10              44,45,49,53,54
11              51,52,71,78,80
12              20,21
13              56,57
14              61,62,63,64
15              31,60,65,66,67,68
16              72

*main
GREENFLAG 1

*text
Write a program that prints the number 6.
-RUN- the program, then type -MORE-.

*model
9
        10 PRINT 6
        99 END

*hint
Hint #1
    Congratulations!  This is a hint.
    Your program should have two statements: one -PRINT- statement, and
one -END- statement.  Each needs to have a line number.
    If you type -HINT- again, you'll get another hint.
*
Hint #2
    Congratulations!  This is the second hint.  In any task, you can
type -HINT- as many times as you like. . . if there are more hints, you
will get them.  And as a last resort, you can always type -MODEL- to
see the model solution.  (But you won't get it unless you've
exhausted the hints and the demo.)  Section III.2 is the place to look.
Try out all the commands you like.)
*
#

*disFns
INT,RND,SQR

*skills
1
```

```
*main
BACK 65

*text
This task and its continuation will help you count backwards.

Write a program that counts from 10 down to 1.  In this task, do the
whole thing "by hand", like this:
        1. set some variable equal to 10.  (Say, X)
        2. Print the value of the variable.
        3. Subtract 1 from its value.
        4. if the variable is still greater than zero, go back to
           step 2.  Otherwise (automatically) continue.
        5. Print "sdrawkcab gnitnuoc"

Use -TRACE- or -FLOW- to see what your program is doing.  Use
-DEMO TRACE- to see what the model solution is doing.

*model
;
        10 X = 10
        20 PRINT X
        30 X = X-1
        40 IF X > 0 THEN 20
        50 PRINT "SDRAWKCAB GNITNUOC"
        99 END

*hint
Hint #1

Step 3 means: whatever the value of X is, subtract 1 from that value.
Assign the result to the variable X.  Look at "assignment" in the
glossary if you are confused.
*
#

*reqOps
IF,LET

*disOps
FOR,GOTO

*skills
2,3,11,44,54
```

*moreTank
BACK,1 66

*text
You just saw how to write that counting loop "by hand," using specific statements to assign the first value to X, to subtract 1 from it, and to see if it was low enough to stop.

Now write a program that looks like it does exactly the same thing (count backwards from 10 to 1), but this time use a -FOR . . NEXT- loop and make BASIC do some of the work for you.

*model
;

```
10 FOR X = 10 TO 1 STEP -1
20   PRINT X
30 NEXT X
40 PRINT "SDRAWKCAB GNITNUOC"
99 END
```

*hint
Hint #1

Starting with the glossary, find out what -FOR . . NEXT- loops do and how they do it. Don't be confused by extra indentations. They just help you see which statements are "inside" the loop, where they will be repeated.
*
#

*reqOps
FOR

*disOps
IF,LET,GOTO

*skills
2,3,61,64

*main
ALPH 43

*text
Compare two strings typed by the user. A string is "less than"
another string if it comes before the string alphabetically:
"APPLE" < "FISH" is true.

Your program should print something like
APPLE COMES BEFORE FISH
depending, of course, on the user's two strings.

*model
;e "ARTICHOKE" : "ASTROLABE"

```
1 REM P$ IS: THE USER'S FIRST STRING
2 REM Q$ IS: THE USER'S SECOND STRING
10 PRINT "TYPE A STRING - A WORD WILL DO."
20 INPUT P$
30 PRINT "TYPE ANOTHER STRING."
40 INPUT Q$
50 IF P$ < Q$ THEN 80
60 PRINT Q$; " COMES BEFORE "; P$
70 STOP
80 PRINT P$; " COMES BEFORE "; Q$
99 END
```

*reqOps
IF,INPUT

*disOps
LET

*skills
2,14,30,48

APPENDIX D

SAMPLE RUNS OF BCLASS AND REPORT PROGRAMS

SAMPLE RUN OF BCLASS PROGRAM

```
@bclass
BIP Student Class-Report Program:
Type a  "?"  for help at any time.

List of student numbers, please:  ?
Type a student number for a single student, or a list of numbers,
separated by commas and/or dashes.
For example,
        1-30,35,37-40
would get you student numbers 1 through 30 inclusive, student 35,
and 37 through 40.

List of student numbers, please:  12,14-25

Where do you want the output?  ?
Type a file name if you want the report written to a file.  If you
want the output written to your terminal right now, just type a <cr>.

Where do you want the output?
```

BIP Class Summary Report

2-JUN-77 11:58:10

Student	Tasks	Sessions	Hours	Last Task	Last Signon
12 MARY [redacted]	31	16	15.2	XMAS.1	23-MAY-77
14 SHIRLEY [redacted]	23	9	10.8	BACKARRAY	1-JUN-77
15 TOM [redacted]	54	16	15.1	ROUNDER	29-MAY-77
16 SUSAN [redacted]	36	10	11.4	CHANGER	29-MAY-77
17 DAVID [redacted]	27	14	15.7	USERLOOP	30-MAY-77
18 KEVIN [redacted]	44	14	13.9	ODDCOUNT	24-MAY-77
19 JOHN [redacted]	34	9	16.8	XMAS.1	30-MAY-77
20 MIKE [redacted]	23	10	11.9	ARRAYINDEX	1-JUN-77
21 MARIE [redacted]	66	12	15.0	PAY.1	31-MAY-77
22 BARBARA [redacted]	32	19	14.1	CHANGER	28-MAY-77
23 DICK [redacted]	52	37	20.3	ROMAN	1-JUN-77
24 STEVE [redacted]	29	12	13.3	SCISSORS	29-MAY-77
25 LAURA [redacted]	25	11	12.7	CALCULATOR	1-JUN-77

That is all!

SAMPLE RUN OF REPORT PROGRAM

```
@report
BIP Student Class-Report Program:
Type a  "?"  for help at any time.

List of student numbers, please:  ?
Type a student number for a single student, or a list of numbers,
separated by commas and/or dashes.

List of student numbers, please:  88,90

Where do you want the output?  ?
Type a file name if you want the report written to a file.  If you
want the output written to your terminal right now, just type a <cr>.

Where do you want the output?

Short or long form?  Type "S" or "L":  ?
The short form lists only the most recent task.  The long form lists
all tasks, in reverse chronological order.

Short or long form?  Type "S" or "L":  L

Do you want an explanation of the abbreviations used?  Y


Key to the abbreviations in this report:
   who?        s if student chose this task, b if bip's selection.
   pqo         p = passed, q = quit, o = "other":  either used
                  "enough" to get out, or still in the task.
   und?        y if student "understood the solution," n if not,
                  - if not asked.
   try         number of "MORES" before leaving the task.
   arg?        y if student disagreed with the solution checker.
   mod?        y if student saw model solution before the interview.
   hints       "*" if student saw all the hints, num of hints otherwise


TOTALS FOR
90 JENNY [redacted]          5 total tasks      2 signons      1.133 hours
last signon:  14-MAY-77 09:31:12

Each Task:   who?  pqo  und?  try  arg?  mod?  hints  mins      date
SELFCAT        b    p     y    2     n     n     *      12     14-MAY-77
HORSE          b    p     y    1     n     n     1      10     14-MAY-77
ASSIGN         b    q     n    2     y     n     1       8     14-MAY-77
PI             b    p     y    1     n     y     *       7     13-MAY-77
GREENFLAG      b    p     n    1     n     n     0      31     13-MAY-77
```

TOTALS FOR
88 SUSAN [redacted] 8 total tasks 3 signons 1.617 hours
last signon: 14-MAY-77 18:20:15

Each Task:	who?	pqo	und?	try	arg?	mod?	hints	mins	date
PLUSFOUR	b	p	y	1	n	n	1	12	14-MAY-77
HORSE	b	p	y	1	n	n	1	6	14-MAY-77
ASSIGN	b	p	y	2	n	n	1	5	14-MAY-77
CAT.1	b	p	y	1	n	n	0	5	14-MAY-77
CAT	b	p	y	1	n	y	*	17	13-MAY-77
OPERATOR	b	q	n	2	y	n	1	18	13-MAY-77
STRINGY	b	p	y	1	n	n	*	6	12-MAY-77
GREENFLAG	b	p	y	1	n	n	0	28	12-MAY-77

THAT IS ALL!