DOCUMENT RESUME

ED 092 157 IR 000 718

AUTHOR	Milner, Stuart
TITLE	Learner-Controlled Computing: A Description and Rationale.
INSTITUTION	Catholic Univ. of America, Washington, D.C. School of Education.
PUB DATE	Apr 74
NOTE	16p.; Paper presented at the American Educational Research Association Annual Meeting (Chicago, Illinois, April 15-19, 1974)
EDRS PRICE	MF-$0.75 HC-$1.50 PLUS POSTAGE
DESCRIPTORS	*Affective Objectives; *Case Studies; *Cognitive Processes; *Computer Assisted Instruction; *Problem Solving; Programing; Student Centered Curriculum
IDENTIFIERS	Learner Controlled Computing

ABSTRACT

Learner controlled instruction in which the student controls the computer (e.g., computer programing) instead of it controlling the student (e.g., drill-and-drill-and-practice) is described. The nature of this mode of computer use is explored, and some examples based on case studies conducted by the author are given. A rationale for learner control is discussed in terms of cognitive and affective outcomes of computing. The cognitive outcomes include relatively specific learning and thinking skills and more general systematic methods of problem solving. Affective outcomes include self-confidence, curiosity and exploratory behaviors, and motivation. (Author/WCM)

LEARNER-CONTROLLED COMPUTING: A DESCRIPTION AND RATIONALE[1]

Stuart Milner

School of Education
The Catholic University of America
Washington, D. C. 20017

This paper discusses a use of technology in which the student controls the computer (e.g., computer programming) instead of it controlling the student (e.g., drill-and-practice). A description of the nature of this mode of computer use is provided, and some examples based on case studies conducted by the author are given.

A rationale for learner control is discussed in terms of cognitive and affective outcomes of computing. The cognitive outcomes include relatively specific learning and thinking skills and more general systematic methods of problem solving. Affective outcomes include self-confidence, curiosity and exploratory behaviors, and motivation.

INTRODUCTION

Instructional uses of computers may be classified on a continuum with the amount of student control as the underlying variable. At one end, lie the classical computer-assisted instruction methods (e.g., drill-and-practice) in which the student has no real control except for his responses to pre-programmed instruction. At the other extreme are the methods used by the student (e.g., computer programming) to control the hardware and/or software of the computer. The purpose of this paper is to describe the nature of, and provide a rationale for, learner-controlled computing.

---

[1] a paper presented at the Annual Meeting of the American Educational Research Association, Chicago, Illinois, April 1974.

One of the reasons it is important to describe the nature of learner-controlled computing is that there exists, at least for some, a misconception or lack of understanding about it. For instance, it is often considered as the so-called problem-solving mode, wherein the computer is used merely as a calculating aid. Actually, learner-controlled computing transcends this, and facilitates the acquisition of thinking and learning skills, among other things. The computer is to be used by the student in his problem-solving endeavors in much the same way as professionals use it to gain insight into complex areas such as sending man to the moon.

Learner-controlled computing in this context refers to the learner's power to control the course of learning by implementing his <u>own</u> problem-solving strategies and executing these at <u>his</u> will. Involved in computing is the development of algorithms by the student for problem solution. In the process there are a number of tasks which include, analyzing the problem, devising and implementing a program for its solution, testing the validity of the program, and, if necessary, finding errors in, or debugging, that program. In effect, the student is the teacher and the computer is the student--supporting the general agreement that in teaching a subject one learns it better. As an example, if a student is to learn the laws governing a process, he writes the program to simulate the process <u>and</u> studies its operation.

Contrasted with "closed-loop" learning objectives (e.g., programmed instruction, many CAI tutorials), computing involves "open-loop" or less restrictive objectives. Accordingly, subject-matter experiences are student-determined and are often serendipitous in occurrence.

Nonetheless, there is a seeming paradox between the students' freedom

to develop complex programs such as advanced plotting and game-playing routines, according to their own interests, and the forced clarity and rigorous thinking involved in computer programming. Perhaps this is what makes computing so facinating to so many.

The rationale for learner-controlled computing includes the potential for: 1) individualization, 2) the use of "real world" applications, and 3) the acquisition of generalized learning skills (i.e., "learning how to learn").

If one accepts the broad definition of individualized learning as that which proceeds according to the unique needs and interests of the student, one can see that by allowing students to implement their own problem-solving strategies through computer programs based on unique needs and interests, individualization is served. Moreover, in the highly responsive environment of computing, it is conceivable that students will learn to find out things by themselves based, in part, on their own spontaneous activities, and, in part, on the support system (e.g., teacher, software, courseware) we provide for them.

In addition to individualization, computing can include meaningful, "real world" projects such as, the design of lunar simulation or instructional programs written by a student for his peers (e.g., drill-and-practice programs). A special advantage of such learning situations is that creativity is fostered when real problems of the sort professionals must solve are used. Another is that students are motivated by seeing the meaning and relevance of what they are doing.

A third justification stems from the need to make more pervasive

broad educational goals that are now somewhat excluded. As Toffler (1970) states:

> Given further acceleration, we can conclude that knowledge will grow increasingly perishable. Today's 'fact' becomes tomorrow's 'misinformation.' This is no argument against learning facts or data--far from it. But a society . . . places an enormous premium on learning efficiency. Tomorrow's schools must therefore teach not merely data, but ways to manipulate it. Students must learn how to discard old ideas, how and when to replace them. They must, in short, learn how to learn.

The major focus of this paper is the explication of generalized learning skills which can be acquired through computing. Discussion will be in terms of cognitive and affective learning outcomes based on both empirical evidence and conjecture.

Before describing them, though, it should be mentioned that the formulation was partially the result of case studies in computing by this investigator. A study involving fifth grade students is described elsewhere (Milner, 1973). In a more recent study, sixth grade students of a wide range of ability in a suburban elementary school participated. The environment was open-ended in that students were free to define their own projects or to pursue suggested ones according to their own interests and motivation. It might be useful to note some examples of projects the students were involved in, all of which were their own choice.

Several chose to work on gaming routines, based on a prototype game, whose object was to guess a predetermined number within a fixed range. The introductory program consisting of several sub-procedures proved useful as a starter; the students added their own extensions, generalizations, and refinements.

One student developed a tic-tac-toe game which consisted of several sub-procedures including printing and interaction routines, and represented a first approximation to a complex project.

Another student, who was learning a programming language (LOGO) from an available manual, became interested in an example of a drill-and-practice arithmetic program. She extended and modified the example program into a fairly sophisticated drill-and-practice multiplication program.

In general, the students' projects were interesting and non-trivial. Of significance was the fact that they were involved in these projects over extended periods of time--almost two months in some cases.

This project-oriented approach has been advocated by Papert (1971a) and Dwyer (1974), who are using computer-controlled mechanical devices and other complex systems at both the elementary and secondary school levels. Regarding projects that extend in time, Papert (1971a) states:

> . . . the process is long enough for the child to become involved, to try several ideas, to have the experience of putting something of oneself in the final result, to compare one's work with that of other children, to discuss, to criticize on some other basis than 'right or wrong'.

In another case study conducted by this investigator, eigth grade students in an urban secondary school learned, among other things, computer programming. What was interesting here was the fact that the students were previously identified as "low achievers," yet they were able to engage in the kind of algorithmic thinking required in computing. Affective outcomes such as increased motivation to do "academic" work, which was previously eschewed, were also observed.

ACQUISITION OF GENERALIZED LEARNING SKILLS

The cognitive and affective outcomes of learner-controlled computing involve the acquisition of generalized learning skills (i.e., "learning how to learn"). This discussion is primarily in the context of problem solving, although it need not be limited to that domain. Moreover, the skills dealt with are not exhaustive and constitute an attempt, based on observation and conjecture, to explain the outcomes.

The skills described may be acquired with or without direct instruction. In other words, the skills may be taught by another person or acquired naturally by the student without human intervention through computing. The case for using computers is based partially on the dynamic, challenging, and responsive environment they produce.

Cognitive Outcomes

For purposes here, cognitive outcomes have been divided into two broad categories. One includes learning and thinking skills and is relatively specific. The other, a more general category, concerns systematic methods of problem solving. As the relatively more specific skills are developed and incorporated with the more general ones, powerful problem solving processes emerge. Following is a discussion of them.

The algorithmic nature of computing can serve as an excellent context for learning how to organize information. In developing an algorithm for problem solution, the student needs to structure material in an unambiguous sequence and, in the process, think ahead, anticipate outcomes in advance, and occasionally alter previously formed steps. More specifically,

the student needs to do such activities as determining when and how to input and output information, assign values, allocate storage, perform iterations, make decisions, implement procedures, etc. In doing so, students gain insight into the nature of algorithms--how they are formed, debugged, and executed.

The fostering of independent thinking, a broad outcome of any individualized learning environment, can be partially achieved by giving students the opportunity to choose their own projects and provide unique solutions to them. It becomes important in this case to get students to take the initiative to develop, modify, solve, and extend problems in ways that make sense to them, just as scientists do in their endeavors. Of course, for students who "don't know what to do," projects could be suggested and a certain amount of structure or direction provided. Observations have led me to believe that once students are given freedom to explore, autonomous thinking follows.

As students learn to learn, they need experience in viewing problems in alternate ways. In one sense, this involves improvisation. Anyone who used various types of programming languages has experienced this. In another sense, it is the using of different paths toward the same end. Different algorithms can produce the same result. In any event, viewing problems in different ways can be made transparent in computing.

Logical thinking is facilitated by the very nature of computing. Consider, for example, the sequencing of operations; although other dimensions of logic are involved, too. Writing computer programs requires a

precise sequence of operations. Students learn very quickly that even one operation out of sequence or missing from a sequence will not yield the desired results. I have observed cases where students know that a function is necessary and how to use it, but are unsure as to where to place or reassign the function in an existing algorithm.

If one has internalized a process, one should be able to explicate that process. This involves something more complex than just recognizing an alternative on a multiple choice test item, or producing a single-valued solution to a problem. Part of the complexity is due to virtually an unlimited number of possible paths leading to the problem solution, a solution which may involve the algorithm itself, or even a package including the algorithm, and other components.

In traditional modes of instruction, students rarely have the opportunity to give a detailed explanation of their own understanding of a subject. Moreover, they often cannot describe how they arrived at a solution nor how a process works. There may be at least two reasons for this: First, students and teachers may lack a sufficiently precise language in which to communicate (Feurzeig, et al., 1969). I have observed cases where students obviously understood a process but were unable to use the right language to express it. Through programming, which involves a precise language, students not only can present their thoughts via the computer, but can discuss their programs in terms of structure, content, etc. Secondly, students do not usually get enough practice in actually stating their thoughts. I always encourage students to explain what their program is about and whether it is a completed project or an approximation of one--"expressive power to the students."

Critical to problem solving is the ability to debug or search for and eliminate errors. In computing, this involves determining at various points if the program executes correctly. Not only can students be taught specific techniques, but they also quite naturally develop their own. In the former case, if programs do not execute properly, students could be instructed to build checks in the logic or print out values at various stages in the program--tasks which are relatively easy to do on a computer. In the latter case, students in the process of debugging develop useful heuristics through guessing and trial and error methods. Heuristics, defined here, are plausible strategies that serve to guide, discover, or reveal. [Polya (1957) provides an excellent discussion of heuristic reasoning]. What is involved, then, is a concrete problem-solving situation, in which the student has the power to change and experiment, and hopefully get immediate knowledge of results.

Often, when students work on problems, they do not work on them long enough to develop generalizations. However, if students are given the opportunity to develop problems that extend beyond traditional time allocations, generalizations can be facilitated. For example, a sixth grade student was writing a drill-and-practice program in mathematics, and her initial program simply dealt with adding two one-digit random numbers. She generalized it to present two-digit addition problems and two-digit multiplication problems with record keeping and other feedback functions. All of this took place over a period of approximately six weeks.

The above specific learning and thinking skills should be used in conjuction with the relatively more general systematic methods of problem solving. These systematic methods include problem comprehension, hypothesis generation, experimentation, and reflection, and can be learned through computing. Polya (1957) describes some of these methods for teaching mathematics. Computer programming, by its very nature, requires the careful analysis and precise explication of thought that these methods imply. Also, it is relatively easy to hypothesize and experiment in programming by changing steps in an algorithm or by varying input/output parameters. Once a program is executed, the validity of an algorithm can be tested and reflection can occur.

Initially, the student needs to gain a general understanding of the problem in terms of expected type of outcome or mode of solution, conditions involved, etc. Understanding a problem, at least in the early stages of comprehension, involves knowing what to look at in a problem. Students might consider questions such as the following: (1) What are the parameters to be used? (2) What is important and what is "noise?" (3) What might a solution look like and what kind of information does one need to know in order to reach it? Some other questions that students need to ask at the problem comprehension stage are: (1) What kind of information/data am I dealing with? (2) What additional information or clarification is needed? (3) What resources are available, and what resources are needed? (4) Are there any transformations in the data to be made? (5) What does successive approximation to a solution look like? (6) Where should specific procedures be placed in a solution algorithm? When and how should they be implemented? etc.

Following an understanding of the problem--a solution may not be at all clear, but the student should have an idea of what needs to be done and a direction to pursue--it is important to generate hypotheses or ideas which would lead to a plan for project execution. A question that students ask at this stage is, "How could I do that?" For example, if a student were programming a computer to play the game of tic-tac-toe, he might be taught to divide the project into three parts: (1) a computer program to simply record events between two players; (2) a program to keep score and evaluate play; and ultimately, (3) a strategy playing program. What was achieved in earlier steps (parts) would be used in combination with later extensions leading to successful project completion. In short, the student would learn how to build extentions and solutions from smaller, more manageable parts. Papert (1971b) advocates this approach and provides some interesting examples from his research.

By dividing a project into parts, then, the student makes it more manageable and learns an important aspect of thinking. In turn, each sub-project (part) might involve even smaller parts so the process here is iterative. For instance, in the case of the first part of the tic-tac-toe game described in the preceding paragraph, the student might initially need to generate printout and mode of entry routines, each of which is a separate problem in itself. It is relatively easy to use sub-programs on a computer, incidentally, once a computer language is learned.

When hypotheses or ideas have been formed, experimentation is the next step. This execution phase is the basis for acceptance or rejection of ideas. Once it is completed, the student can determine if the project has been finished. In the case of computer programming, this is an iter-

ative phase in which various experiments and considerable hypothesis testing takes place. Eventually, either a program works or it does not. In the interim, debugging plays a major role. Also, the student can be taught to reformulate hypotheses or reinitialize processes.

Based on the experiments, the student could reflect back on the solution and refine, extend, or even generate new projects. As such, problem solving approximates "real life" situations where one works through solutions possibly many times over, vis-a-vis traditional "one-shot" solutions that are graded and returned by the teacher. If the extended or new projects have similarities with previous ones, the student exercises inductive reasoning.

Affective Outcomes

Expected affective outcomes in learner-controlled computing environments generally deal with values, attitudes, and interests. Some of the outcomes, as discussed below, deal with self-confidence, curiosity and exploratory behaviors, and motivation.

A certain degree of early intellectual mastery in a learning situation is important in building self-confidence. In computing, students can write simple programs that execute at their first sitting at a terminal. They are in control. That students perceive and, possibly, prefer this control is observable; or as one sixth grade student stated: "I like writing programs because its different than using programs [CAI] and also more fun."

Enhancement of self-confidence leads to increased experimentation in general. The computer certainly does not harass one for doing so. I would speculate that a non-threatening, learner-controlled environment

would serve to enhance self-confidence in much the same manner as the Hawthorne Effect increases productivity. In other words, by placing a child in a relatively different environment where he can experience control and mastery, we could expect to foster the development of his self-confidence.

Curiosity and exploratory behaviors such as frequent querying (of humans and computers), initiation of new ideas and rethinking old ones are natural outgrowths of increased self-confidence and favorable attitudes. They are also a function of the inherently interesting nature of computing. Although these behaviors can be encouraged, they are largely due to the challenging nature of instructional materials.

The self-produced changes inherent in learner-controlled computing are motivating. It follows that the students appear involved in their work. But then, students who are doing work of little interest to them also appear involved. I am referring to active involvement here, as opposed to passive responding, whereby students perservere and do so voluntarily, enthusiastically, and on a self-directed basis in the course of learning. For example, redesign and improvement of algorithms is not characteristic of typical "workbook problems."

Finally, creative behaviors such as independent study, and the potential for using varieties of insights or reactions, are fostered when students are given the opportunity to develop their <u>own</u> projects in learner-controlled environments.

* Perhaps a "meta-outcome" of the learner control in this context

is the acquisition of self-management behaviors. These include the what, when, and how of learning. What might be learned includes the generalized learning skills described above, specific subject matter insights, and, less obviously, assessment of competencies and setting of objectives. When learning takes place depends on student initiative, degree of involvement, as well as serendipitous experiences; implied is a direct relationship between innerent interest of material and student responsibility for learning it. How learning takes place is a function of many things including self-checking, subject-determined experiences, and teacher/student and student/peer interactions. As students learn to assess the validity of their programs, for example, they will internalize self-checking strategies and learn to know when they are right.

CONCLUDING REMARKS

A great deal of "lip service" is paid to the incorporation of "learning to learn" objectives. Yet very few, if any, exist in practice. The claim of behaviorists that these objectives can be devised in their "closed loop" systems remains unproven. One prescription is offered in this paper.

What are the implications, then, of all of this? For one, it should be clear that learner-controlled computing is a powerful resource in learning, and as such should be further exploited. In addition, it is hoped that the broad outcomes discussed, as well as others that certainly must exist, will be assimilated by educators along with the more limiting instructional objectives, but to a greater extent than presently. From a pragmatic and immediate standpoint, these outcomes and others might be used for evaluation, such as in the form of a "checklist" of learning outcomes.

Whatever the implications, research and development of learner-controlled computing will not be easily amenable to classical research designs nor even cost-benefit analyses. Part of the difficulty lies in the complexity and non-standard nature of the objectives. It certainly will not be easy to measure these outcomes according to present criteria (e.g., standardized tests). The actual programs written will clearly demonstrate knowledge. Observation and anecdotal record should also provide some initial data for us. For anyone who has observed students engaged in computing over any period of time, it it at least intuitively obvious that learning in this manner is promising. The challenge to actualize the promise is now.

REFERENCES

Bayer, T. "Heuristic Strategies for Using Computers to Enrich Education." International Journal of Man-Machine Studies 6, 1974 (to appear).

Feurzeig, W., Papert, S., Bloom, M., Grant, R., Solomon, S. Programming Languages as a Conceptual Framework for Teaching Mathematics. Project Report No. 1889. Bolt, Beranek and Newman, Cambridge, Mass., 1969.

Milner, S. The Effects of Computer Programming on Performance in Mathematics. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, La., February 1973.

Papert, S. Teaching Children Thinking. Massachusetts Institute of Technology Artificial Intelligence Laboratory, LOGO Memo No. 2, 1971a.

_____. Teaching Children to Be Mathematicians vs. Teaching About Mathematics. Massachusetts Institute of Technology Artificial Intelligence Laboratory, LOGO Memo No. 4, 1971b.

Polya, G. How to Solve It. (Second Edition). Princeton, N.J.: Princeton University Press, 1957.

Toffler, A. Future Shock. New York: Random House, 1970.