The problem

Introduction

From its inception the electronic digital computer has been involved in education, although its role has been the subject of some debate. Academically, the computer has been used as a device for conducting or augmenting instruction, as a calculating device adjunctive to courses in engineering and the sciences, and most recently as an object for study in its own right. Increasing attention is now being focused on undergraduate training in computer use, partly as a result of the recommendation of the President's Science Advisory Committee that computing education be provided to all college undergraduates. We can expect that the number of students who undertake incidental study of programming as a part of their undergraduate curricula will continue to increase rapidly.

While the demand for programming instruction is increasing, little research has been conducted on how that instruction might be improved. There have been some investigations of programming behavior, but these have mainly been concerned with how experienced programmers behave rather than how programming skills are acquired or how learning these skills may be facilitated. Many instructors express an uneasy feeling that the objectives of the introductory programming course are often not met; that many students fail to develop programming competency.

It is, therefore, important that we look for the sources of learning difficulties so that the courses we teach may be more effective. Motivational and instructional pacing problems, resulting from the frequent occurrence of students of unequal aptitude and experience heterogeneously grouped in large classes, clearly are factors. However, theories founded on educational psychology seem to suggest that more fundamental problems, arising from the absence of certain requirements for effective learning, are at play. It is the objective of this paper to propose a theory explaining some of these more fundamental problems, and to suggest the need for research supporting a methodology for overcoming them.

Rote and meaningful learning

Educational psychologists classify learning along several dimensions. One of the most important of these dimensions is the rote-meaningful continuum. Learning which is rote is verbatim memorization. In contrast, meaningful learning is non-verbatim; that is, the learner is able to restate the material in his own terms. Material which is meaningfully learned may be reformulated and used by the learner. In order to learn material meaningfully, the learner must integrate the material with ideas which he already understands. Thus the background of the student must be adequate if the material is to be meaningfully learned.

Rote learning and meaningful learning are not absolutes. Learning may be more or less meaningful or rote, depending upon a number of factors. The precise degree of meaningfulness will vary with the individual student. Factors which lead to rote learning tendencies are: material which is verbatim by nature (such as most foreign language vocabulary), lack of background on the part of the student, poor presentation of the material, and anxiety on the part of the student.

A complete discussion of the rote-meaningful dimension of learning may be found in Ausubel. Numerous studies have shown that meaningfully learned material is remembered far longer, recalled with less difficulty, and utilized more effectively than is material learned in a rote manner. However, in order for a student to learn a subject meaningfully each fact to be learned must be integrated with the student's previous knowledge. Each new idea must be either an elaboration, an addition, a contradiction, or an example of an idea already in the student's background.
Learning to program is not typical of other learning because it has decidedly rote and decidedly meaningful components. The syntactical elements of the language cannot easily be presented in a meaningful manner, or related substantively to other constructs in the beginning student's experience. For example, it is difficult to see how the fact that a FORTRAN statement must begin in (or beyond) column 7 can be presented in a meaningful manner. It is an arbitrary fact which must be memorized by the student. However, the assembly of statements into a program requires a skill quite beyond the recall of arbitrary facts. It is impossible to learn how to program in a "verbatim" manner because programming is fundamentally a process of selecting and structuring statements chosen from those available within the language. Even general forms must be adapted to the particular requirements of the problem before they can be used in a program. For this reason it is important that the learning of programming be meaningful rather than rote.

We have pointed out that there are several factors which may cause material which could be learned meaningfully to be rote learned by the student. One of these factors is the lack of adequate background for the material, or insufficient context. Another factor which may inhibit meaningful learning is insufficient vertical transfer—an inability on the part of the student to transfer his understanding of a concept to an application of that concept in a program. A third inhibitory factor in the meaningful acquisition of concepts is a state of anxiety on the part of the student, which in this context we have called computer shock.

It must be recognized that these factors are not absolute. Material is rarely learned either completely or completely meaningfully. In general, the rote-meaningful dimension is a continuous one, and the inhibitory factors to meaningful learning may operate to a greater or lesser extent depending upon factors unique to the individual student or to the learning situation. However, to the extent that these factors are operative they will tend to make it more difficult for the student to learn programming. Therefore, a programming curriculum which minimizes the effects of these factors should be more successful in meeting its objectives than alternative approaches. Before recommending a curriculum which minimizes these factors we will elaborate on them.

**Insufficient context**

Insufficient context results from a lack of related ideas in the student's background which can serve as a base ("anchor") for the new programming concepts and techniques being learned. Context derived from internalized concepts and ideas can provide substantive links between existing cognitive structure and the new meanings being encountered. For the beginning student context must be derived from prior knowledge and experience; as the student progresses context can be constructed from the set of basic and unifying principles that underlie the programming discipline, and as a result comprehension of more subtle concepts is possible.\(^4\)

The lack of relevant prior knowledge on the part of the beginning student makes learning and retention more difficult. The presence of a sufficiently inclusive context makes it possible for new ideas to be reconciled with existing ideas and integrated into the cognitive structure more rapidly and efficiently. Braunsford and Johnson have shown dramatically that establishment of context prior to exposing students to material to be learned facilitates both comprehension and retention of the material.\(^5\)

Beginning students usually have few relevant anchoring ideas that could support efficient learning of programming. The study of a programming language is in this sense quite different from the study of a second verbal language. Bernard argues that the latter "consists fundamentally in the acquisition of an additional set of symbols for old, familiar meanings."\(^7\) In acquiring a second language the student has already mastered basic concepts and syntactic code and thus has the necessary relevant context as part of his prior knowledge. In programming, however, the student has no easy referents for such concepts as the interchange of two variables or the searching of a matrix. These concepts thus form basic meanings whose learning must precede the purely syntactical concerns and provide the anchorage for subsequent learning. This explains why a programmer proficient in a single language can usually learn a second programming language very rapidly. The "vocabulary" and syntax of a programming language is usually quite simple once a basic set of constructs is learned, and the student has an appropriate context into which he can integrate the new set of symbols.

The lack of adequate context for the beginning programmer stems from the fact that a computer is unlike anything which the student has previously encountered, nor is programming very much like any problem-solving behavior in which the student has previously engaged. Yet computers are designed as they are for very good reasons—to provide them with certain well-defined capabilities. Unfortunately, it seems that most students are not presented with the relevant anchoring ideas that would help them see the language requirements as a necessary concomitant of the computer's functions.

Unless sufficiently inclusive unifying context is provided, a beginning student without the relevant anchoring ideas must either learn in a rote manner or press into service less relevant ideas, relying on his own ability to reconcile the new ideas with existing ideas in his cognitive structure. In the latter case the new ideas being encountered will be less firmly anchored and more quickly forgotten. If the student must learn by rote, deferring integration until he has acquired more background, then reconciliation and synthesis of ideas becomes difficult and his ability to apply his new "knowledge" is correspondingly impaired.

**Insufficient vertical transfer**

The second hypothesized source of difficulty in learning programming is insufficient "vertical transfer." Transfer is the facilitating effect of previous learning on new learning.
Transfer is usually defined as “horizontal” or “vertical.” Horizontal transfer occurs when a previously learned task, A, facilitates the learning of a new (and different) task, B. Vertical transfer occurs when rules or concepts learned at a “lower” cognitive level (such as concept learning) are applied at a “higher” cognitive level (such as problem solving). In the case of programming, the objective of instruction is to help the student overcome the problem of knowledge (of syntax) to the level of application, which is generally regarded as a higher cognitive level. In another context, the ability to play a musical instrument represents a (vertical) transfer from the level of knowledge (of musical notation and instrument mechanics) to the level of skill or application. In this case the skill is partly motor and partly intellective.

Transfer from the intellective to the skill level is essential when applying knowledge of a programming language to a specific problem. In most cases the beginning student’s knowledge of the language consists primarily of vocabulary and syntax. Even if the solution to a specific problem is given to him in the form of a verbal algorithm—indeed, even if the implementation itself is shown to him—he has difficulty accomplishing the transfer because the gap between knowledge of vocabulary and syntax and problem solution is too great.

Gagné has said that “the components which appear to make problem solving possible are the rules that have previously been learned. Problem solving may be viewed as a process by which the learner discovers a combination of previously learned rules that he can apply to achieve a solution for a novel problem situation.” (Reference 8, page 214)

Thus a proficient programmer, working on a problem that is within his experience, is hypothesized to operate primarily at a synthetic level. That is, he has learned or discovered the techniques or “meta-rules” that might be applicable to his specific problem, and applies these techniques in a relatively straightforward manner. It is precisely these meta-rules which permit him to operate at the level of synthesis much of the time, using analytic skills only when new constructs or techniques are required.

The beginning student, however, must try to break his problem down into a much finer set of subtasks that are recognizable to him in terms of his knowledge of the basic vocabulary and syntactic rules of programming before he can begin to develop the larger structures. In effect, he is constructing new techniques for solving non-intuitive problems, and this is an intellectually difficult task. Moreover, he must hold all potentially relevant rules in mind at one time when searching for those which might be applied to his problem.

The specific rules may not be obvious to the student because there are fundamental differences between human problem-solving techniques and computer solutions. People tend to solve problems by a “top down” approach, seeking patterns which may then be considered as problems of reduced complexity. Computers, on the other hand, have almost no ability to consider patterns, but rather operate in a sequential manner. As an example of this, consider the problem of determining which of the following numbers is largest:

1 12396 14

A person scanning this list would automatically and effortlessly pick out the second number as the largest, not because of its value but because of its length. A computer program to perform this same task would need to be written as a complex series of sequential operations; the following algorithm is an example:

\[
i := 1; j := a[1];
\]

\[
for k := 2 step 1 until n do
\]

\[
begin
\]

\[
if a[k] > j then
\]

\[
begin i := k; j := a[k]
\]

\[
end;
\]

\[
end;
\]

Such algorithms, while they are “natural” to the experienced programmer, are highly non-intuitive to the beginner who does not recognize the relevant rules. To expect the student to be able to create such algorithms without careful preparation is unreasonable; yet attempts are often made to teach this algorithmic approach by showing the student a complete program or two and expecting him to somehow absorb these non-intuitive concepts.

Even when a student is shown the solution to a problem he has difficulty transferring that solution to other situations. A stimulus-response chain may be created, in which the student learns to respond in certain ways to certain problems without understanding the underlying rationale for the response. When asked to generalize from the solution, or solve related but perhaps more complex problems, he is at a loss.

The problem of learning from a given solution, or constructing generalizations of it, is particularly acute when the solution is complex or counter-intuitive, as for example in a sort routine. A study by Newsted suggests that there is a critical point of program complexity beyond which the student is unable to grasp the solution as a unit and requires additional documentary aids.

It therefore seems reasonable to conclude that the learning of a programming language can be significantly improved if the student is encouraged to operate primarily at the level of rule application. To do this he must draw on knowledge of larger (non-intuitive) constructs than those represented simply by the vocabulary and syntax of the language. He must be able to subordinate his problem under such generalized techniques or meta-rules as sorting, interchanging, searching, etc. In a sense he must be presented with an extended “syntax” which includes these basic techniques or processes.

Computer shock

Beginning programming students are particularly susceptible to a phenomenon similar to one which in mathe-
The problem of facilitating vertical transfer, so that the student is able to utilize the concepts which he learns, can be resolved by judicious selection of exercises which implement fundamental programming structures. These exercises should drill the student in such areas as variable assignment, exchanging values, statement repetition, and basic program structure. The techniques which underlie the selection of these exercises can be found by inspection of professionally written programs to isolate common program structures utilized.

We have postulated the existence of "meta-rules" which are basic constructs which programmers use to build programs. An example of such a meta-rule in FORTRAN might be the zeroing of an array by use of a DO loop:

\[
\text{DO 10 } I = 1, N \\
10 \ A(I) = 0.
\]
or the interchange of two variables:

\[
\begin{align*}
\text{TEMP} &= A \\
A &= B \\
B &= \text{TEMP}
\end{align*}
\]

If such meta-rules can be identified and taught to the student he should be able to develop programs far more rapidly and with greater ease, since he no longer needs to invent these constructs each time he needs them.

Computer Shock

The effects of computer shock may be minimized in several ways. First, the mystery which tends to surround the word "computer" should be dispelled as quickly as possible by describing the simple, readily understandable functions it is designed to perform and the manner in which these functions are built upon to create the more elaborate structures associated with actual applications. This will provide the student with a better sense of the capabilities and limitations of the computer, and place in perspective the reasons it is designed as it is.

Second, the basic operations the computer performs should be described in terms of similar operations with which the student is already familiar. The operation of an adding machine, for example, provides a good analogy for introducing the concepts of input, output, and program control.

Finally, written materials used in teaching should be as attractive and well-organized for instruction as possible. The materials typically found in reference manuals, for example, tend to be imposing to the student and are therefore unsuitable for instructional use.

SUMMARY

This paper has hypothesized three factors which inhibit the acquisition of programming skills. We expect that curricula which minimize the effects of these factors will be the most successful in teaching introductory programming.

Empirical evidence is required to support these hypotheses. Among the research which should be carried out are investigations into the specific concepts required to provide adequate context, identification of the meta-rules which will enhance vertical transfer, and construction of exercises which will facilitate the use and adaptation of the meta-rules in various contexts. Such research is much needed in a society in which "computer literacy" is rapidly becoming a requirement of the educated person.

ACKNOWLEDGMENT

The authors would like to thank Professor Ben Shneiderman for his contribution to some of the ideas formulated in this paper.

REFERENCES

11. Newsted, P. Unpublished manuscript.