Comput. & Education, Vol. 1, pp. 193–197. Pergamon Press, 1977. Printed in Great Britain.

TEACHING PROGRAMMING: A SPIRAL APPROACH TO SYNTAX AND SEMANTICS

BEN SHNEIDERMAN

Department of Information Systems Management, University of Maryland, College Park, MD 20742, U.S.A.

(*Received* 12 *October* 1976)

Abstract—Coupling the recently proposed syntactic/semantic model of programmer behavior [1] with classic educational psychological theories yields new insights to teaching programming to novices. These new insights should make programming education more natural to students, alleviate "computer shock" (the analog of "math anxiety"[2]) and promote the development of widespread "computer literacy".

The spiral approach is the parallel acquisition of syntactic and semantic knowledge in a sequence which provokes student interest by using meaningful examples, builds on previous knowledge, is in harmony with the student's cognitive skills, provides reinforcement of recently acquired material and develops confidence through successful accomplishment of increasingly difficult tasks. The relationship of structured programming and flowcharts to the spiral approach is discussed.

INTRODUCTION

In the 1950's and 1960's programming education focussed on machine oriented issues including descriptions of the binary number system and machine language operation codes. In recent years the tendency has been to eliminate these low level details from introductory courses, while concentrating on high level language issues and problem solving. This approach appealed to a wider range of students and enrollments in introductory computer programming courses have increased substantially. In some exceptional forward looking colleges, notably Dartmouth University where BASIC was developed, computer literacy is expected of most students, but in the future, this will be the norm, not the exception. To design programming courses which are appealing to a wide variety of students we must understand the programming learning process and develop teaching techniques which are easy to follow and highly motivating.

Students in introductory programming courses have misconceptions about computers and the programming process which produce anxiety. By simultaneously confronting novice students with numerous details of keypunching, job submittal and program preparation, instructors compound the anxiety induced fear and generate "computer shock," a term coined by Kreitzberg and Swanson[2] in making an analogy to "math anxiety". Lectures about the wonders and importance of modern computers do little to relieve anxiety, but an opportunity to play computer games or write a simple program at an interactive terminal may do much to improve the student's attitude (these conclusions come from a pilot psychological study done at Indiana University designed to assess techniques for reducing student anxiety about computers).

The best way to overcome student anxiety is to present the material in a sequence which provokes the student's interest by using meaningful examples, builds on previous knowledge, is in harmony with his/her cognitive skills, provides reinforcement of recently acquired material and develops confidence through successful accomplishment of increasingly difficult tasks. This natural sequencing is difficult for many instructors to produce since their own experience of learning programming may have been traumatic and confusing. Their subsequent experiences may make it more difficult to appreciate the confusion of novice programmers.

Educational psychologists have produced a body of literature which provides generalized guidance in developing pedagogic strategies for computer programming. Two worthwhile sources are Bruner[3] and Ausubel[4] who offer theoretical frameworks for developing programming skills. Particularly appealing is Ausubel's notion of "anchoring" new material to a student's "ideational structure" through a process of "progressive differentiation". Bruner comments that "a curriculum should involve the mastery of skills that in turn lead to the mastery of still more powerful ones, the establishment of self-reward sequences".

More recently, educational psychologists have begun to examine specifically the programming learning process. Kreitzberg and Swanson[2] develop a cognitive psychological framework for structuring the introductory programming course. Mayer[5,6] has conducted educational psychological

experiments to determine the effect of computer models, flowcharts and programming languages on the learning process in novice programmers.

These developments are important contributions, but without a model of programming-related cognitive skills and knowledge it will be difficult to design a natural sequence for teaching programming. Responding to this need, Shneiderman and Mayer[7] developed the syntactic/semantic model of programmer behavior. This model was based on an earlier model[8] and numerous experimental results.

SYNTACTIC/SEMANTIC MODEL OF PROGRAMMER BEHAVIOR

The syntactic/semantic model of programmer behavior suggests that programming knowledge is of two kinds: semantic knowledge of program patterns and algorithms, and syntactic knowledge of a particular programming language (see Fig. 1). The semantic knowledge is hierarchically organized with concepts ranging from low level details such as the comparison of two values or the assignment operation; to middle level issues such as the pattern for finding the largest element of an array or zeroing out an array; to higher level operations such as the quicksort algorithm or the conversion of infix notation to prefix notation. There are even higher levels of semantic knowledge such as the compilation process or business related transaction processing which takes an old master file and a transaction file and produces a new master file. This semantic knowledge concerning algorithms is acquired through "meaningful learning", is resistant to forgetting and is language independent.

The second kind of knowledge that a programmer must acquire is syntactic knowledge of a particular programming language. This knowledge is not hierarchically organized and consists of multiple low level details such as the items in a FORTRAN FORMAT list, the syntax of the iteration statement or the names and arguments of built-in functions. This syntactic knowledge must be acquired through "rote learning", must be rehearsed frequently and is subject to forgetting.

This natural dichotomy is a common theme in the problem solving literature and in cognitive discussions of learning in numerous disciplines. The split into semantic and syntactic knowledge also reflects the frequent distinction made in discussions of introductory programming courses. One viewpoint is that programming education should concentrate on logical thinking and problem solving in a language independent process. Numerous introductory computer science texts published in the last few years often present algorithms in an informal notation or use flowcharts, thereby avoiding syntactic details of particular programming languages. The second viewpoint is that programming education should train students to be production programmers who have a thorough knowledge of a specific programming language, the diagnostic messages produced by a compiler and the job control language of an operating system.

In terms of the syntactic/semantic model, the problem solving approach is geared to semantic knowledge while the alternate pragmatic approach, almost vocational in nature, deals heavily with syntactic knowledge. Advocates of problem solving give tests which expect insightful solutions to programming situations and clever modular decompositions while the pragmatic professors require students to recognize the validity or invalidity of given statements, locate bugs in a program or hand simulate the execution of a program.

These descriptions are meant as exaggerations, most instructors choose a middle ground by haphazardly combining approaches. A sensible pedagogic technique is to explicitly acknowledge the exist-

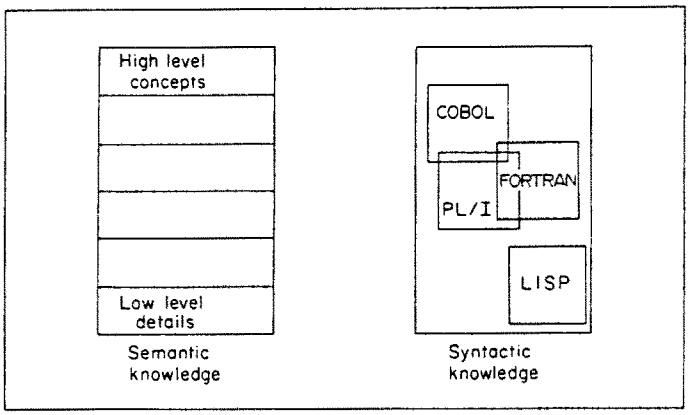

Fig. 1. Knowledge structure in long term memory.

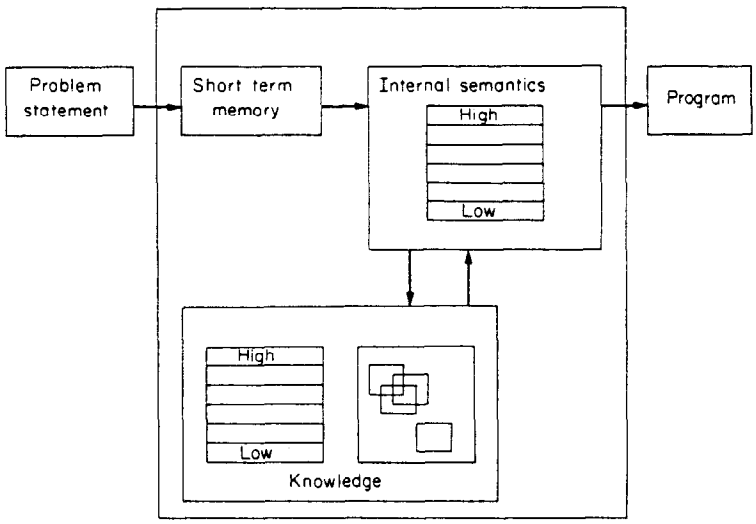

Fig. 2. The program composition process in the syntactic/semantic model.

ence of syntactic and semantic knowledge and to develop both in an orderly manner. The wrong approach is exemplified by the programming language manual which presents only the syntactic issues with few complete program examples or discussions of algorithms. Another poor teaching mode is the presentation of a moderate sized program on the first day of class (typically a program to find the roots of a quadratic equation) with the suggestion that the student should examine this program and somehow intuit proper syntax, relevant program control structures and logical design patterns.

Interpreting the program composition task, illustrated in Fig. 2, in terms of the syntactic/semantic model will help to clarify the terminology. The problem statement is processed through the short term memory as the programmer applies long term knowledge to construct an internal semantic structure. This structure is also hierarchically organized, paralleling long term semantic knowledge. A high level template for the program is established and the lower level semantic structures are filled in. To produce the program the programmer applies his/her knowledge of the syntactic details and generates a complete program. After the problem has been solved and the internal semantic structure developed, a programmer could easily construct the program in any language he/she was familiar with. This hypothesis assumes that the programming languages the programmer was familiar with have similar semantic structures.

THE SPIRAL APPROACH

An educational approach in harmony with the cognitive model is to present students with a small amount of syntactic and semantic knowledge which can be "anchored" to their "ideational structure". For example, a programming course might begin by teaching the semantics and syntax of free-format input and output statements, then progress to the simplest forms of the assignment statement and arithmetic expressions. At each step the new material should contain syntactic and semantic elements, should be a minimal addition to previous knowledge, should be related to previous knowledge, should be immediately shown in relevant, meaningful examples and should be utilized in the student's next assignment. This is the "spiral approach".

More complex syntactic forms should be developed after simple forms have been well established. For example, the elementary DO loop in FORTRAN which starts at an initial value of one and increments by one to a constant final value should be presented before variable limits or larger increments are introduced. Students, who would be overwhelmed if the general form were presented first, can absorb complex forms in a step by step process.

More complex semantic forms should be developed in a hierarchical manner: building complex ideas from combinations of simple ideas. For example, the PL/I DO–END loop can be presented as a combination of an initialization statement, decision statement and increment statement. Similarly, a sorting algorithm can be developed in a sequence of steps from a simpler algorithm which finds the largest element of an array. Not only is this teaching process appealing to students as they acquire new semantic structures, but it provides a natural basis for discussing top–down

design[9] and stepwise refinement[10]. Of course, the success of these learning and program design processes is attributable to the hierarchic structure of semantic knowledge as proposed in the syntactic/semantic model of programmer behavior.

Presenting programming in a spiral approach places a heavy burden on the instructor since texts are often organized for easy reference rather than for naturalness of presentation. Non-essentials must be stripped away so as to provide students with a minimum useful subset of the language which can be expanded gradually. Frequent tests and programming exercises must be given to anchor the new concepts. Informal analyses of universities which have adopted the spiral approach suggests that the drop-out rate decreases and that students are actually willing to do more work because of the positive reinforcement of successful mastery. The spiral approach seems to have been most helpful to the poorer students who were overwhelmed by previous pedagogic strategies.

STRUCTURED PROGRAMMING

Interpreting contemporary programming teaching issues in the context of the syntactic/semantic model and the spiral approach can give us insights to improved teaching techniques. Structured programming control structures, such as the IF–THEN–ELSE, DO–WHILE, REPEAT–UNTIL and CASE statements, have provided a challenge to programming instructors. The proliferation of FORTRAN pre-processors has tempted many to introduce structured control structures in first courses. Unfortunately the pre-processors and the textbooks do not support students and instructors who wish to follow this high-road approach. Students must take the low-road route and learn both the FORTRAN logical IF statement and the more advanced control structures. The confusion of multiple semantic structures and syntactic representations is a burden for novice programmers who are struggling to acquire meaningful conceptual templates for program creation.

When texts and compilers become widely available to support structured control structures, students will be able to go directly to these sophisticated approaches without the confusing translation to lower level syntax. In the interim, students should be given a solid grounding in the FORTRAN operators and should be encouraged to use them sensibly. Once students acquire these skills, the conceptual templates for constructing structured control structured syntax is introduced students will perceive it as natural and well-motivated. This gradual approach is in concert with the hierarchical structure of semantic knowledge and avoids "flooding" the student with a variety of low and high level syntactic and semantic issues.

FLOWCHARTS

Flowcharting, which can be traced back to the earliest days of programming, is staunchly supported by numerous programming managers and textbook authors but vocal critics have recently attacked this common documentation aid. Brooks[11] calls flowcharting "a curse", "a space-hogging exercise in drafting" and "a most thoroughly oversold piece of program documentation".

A series of experiments[12] has failed to show the utility of detailed flowcharts in composition, comprehension, debugging and modification of programs. These negative results suggest that flowcharts may not be an aid to learning programming either. Mayer[5,6] experimentally investigated the use of flowcharts in teaching novice programmers. His subjects, who had no programming experience, were taught a simplified version of FORTRAN. Some subjects used flowcharts, others did not. The flowchart groups did not do significantly better than the no-flowchart groups and for some tasks performed more poorly.

Further experiments will have to be done before we understand how people use flowcharts. The syntactic/semantic model suggests that since detailed flowcharts are merely an alternative syntactic representation of a program, we would not expect them to aid in situations where a program was available or was to be composed. One would not expect a French version of a recipe to aid a chef who had an English version and spoke both languages equally well. Macro or system flowcharts may be more helpful since they provide a hierarchical grouping of statements which should reveal the higher level semantic structure of the program. Other directions for research are investigations of individuals who feel that flowcharts are helpful. If it can be demonstrated that these individuals do perform better when using flowcharts, then a study of individual cognitive styles would be worthwhile. It would not be surprising to discover that some individuals perform better with pictures than with keywords. Other visual documentation aids, such as structured flowcharts[13] and HIPO charts, should be studied for their utility in teaching programming.

Instructors who choose to teach flowcharting should introduce the different symbols in an orderly sequence paralleling the presentation of related programming issues. If a special symbol is used

to indicate iteration, it should be presented when iteration is discussed, not when START, STOP, READ, WRITE and execution boxes are first introduced. Showing a large five page complex flowchart on the first day of class is as confusing as a large unintelligible program.

CONCLUSIONS

The syntactic/semantic model of programmer behavior is a useful tool in formulating pedagogic strategies in programming. It suggests that the spiral approach based on the educational psychological principles of Bruner and Ausubel is appropriate for teaching programming. Students should be taught small increments of syntactic and semantic knowledge in parallel. The syntactic knowledge must be frequently rehearsed and is anchored by repetition. The semantic knowledge is acquired through meaningful learning and is resistant to forgetting but it must be presented in small units which are either subtle variations or higher level organizations of previously acquired knowledge.

These reflections on the teaching of introductory programming to novices are, of course, not relevant for competent programmers who are students of a second programming language. Such students can transfer the bulk of their semantic knowledge and need only learn the syntax of the second language. If the second programming language has differing semantic structures such as parallel processing, recursion, pointer variables or disk access operations, then new semantic knowledge will have to be acquired.

The psychological study of programmer behavior and programming education is a new and exciting field[1]. It takes on critical importance since most college educated students will be expected to have some programming knowledge. By making it easier for all students to have access to powerful computational resources, we can expand their intellectual horizons by giving them a powerful tool which enables them to pursue challenging creative activities.

Acknowledgements—I wish to thank Stuart Milner of Catholic University and the referees for their constructive and supportive comments.

REFERENCES

1. Shneiderman B., *Proc. natn. Computer Conf.* pp. 653–656 AFIPS Press, Montvale, NJ (1975).
2. Kreitzberg C. B. and Swanson L. *Proc. natn. Computer Conf.* pp. 307–311 AFIPS Press, Montvale, NJ (1974).
3. Bruner J., *Towards a Theory of Instruction.* Norton, New York (1966).
4. Ausubel D. P., *Educational Psychology: A Cognitive Approach.* Holt, Rinehart & Winston, New York (1968).
5. Mayer R. *J. Educ. Psychol.* **67**, 725–734 (1975).
6. Mayer R., *J. Educ. Psychol.* **68**, 143–150 (1976).
7. Shneiderman B. and Mayer R. submitted for publication.
8. Shneiderman B., *Int. J. Computer Inf. Sci.* **5**, 123–143 (1976).
9. Mills H., in *Debugging Techniques in Large Systems*, R. Rustin (Ed.) pp. 41–55. Prentice-Hall, Englewood Cliffs (1971).
10. Wirth N., *Communs ACM* **14**, 1971.
11. Brooks F., Jr., *The Mythical Man-Month.* Addison Wesley, Reading, MA (1974).
12. Shneiderman B., Mayer R., McKay D. and Heller P., *Communs. ACM*, **20**, 373–381 (1977).
13. Nassi I. and Shneiderman B. *SIGPLAN Notices* **8**, 12–26 (1973).