International Journal of Computer and Information Sciences, Vol. 8, No. 3, 1979

Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results

Ben Shneiderman[1] and Richard Mayer[2]

Received January 1977; revised November 1978

This paper presents a cognitive framework for describing behaviors involved in program composition, comprehension, debugging, modification, and the acquisition of new programming concepts, skills, and knowledge. An information processing model is presented which includes a long-term store of semantic and syntactic knowledge, and a working memory in which problem solutions are constructed. New experimental evidence is presented to support the model of syntactic/semantic interaction.

KEY WORDS: Programming; programming languages; cognitive models; program composition; program comprehension; debugging; modification; learning; education; information processing.

1. INTRODUCTION

Recent research in programming and programming languages has begun to focus more heavily on human factors and to separate human-centered issues from the machine-centered issues. This natural division enables us to study programmer behavior without concern for implementation issues such as parsing ease, execution speed, storage economy, available character sets, etc.

[1] Department of Information Systems Management, University of Maryland, College Park, Maryland.

[2] Department of Psychology, University of California at Santa Barbara, Santa Barbara, California.

Stimulated by Weinberg's text[29] and the improvements promoted by structured programming advocates, researchers have begun to deal with the cognitive processes of programmers. This research has taken the form of controlled experiments, protocol analyses, and case studies on individuals or groups.[9,16,19,21,23–25,27,28,30,31,35] The tasks studied have included program composition, comprehension, debugging, modification, and the learning of new programming skills. A wide range of subjects, from nonprogrammers to professional programmers, have been tested, mostly on short- or medium-length programs, but occasionally on longer, more complex programs.

Other material on programmer behavior is contained in the publications of the ACM Special Interest on Computer Personnel Research. Interesting personal reflections have appeared in books by Joel Aron[1] and Frederick Brooks, Jr.[4]

A final area of importance is programming education. Research on this topic is covered by the ACM Special Interest Group on Computer Science Education, which publishes the proceedings of an annual conference. Educational psychologists have recently begun to probe the acquisition of programming skills,[12,14] providing a new and valuable viewpoint.

Unfortunately this work is fragmented; nowhere is there a unified approach or theory to account for the results that are beginning to appear. Each paper focuses on a particular problem, issue, task, or aspect of the programming process without producing a broader model that explains the wide range of programmer behavior. A unified cognitive model of the programmer would guide us in future experiments and suggest new programming techniques while accounting for observed behavior. Such a model becomes necessary as we move into an era of more widespread computer literacy, in which an increasingly diverse population interacts with computers. The intuitions and experience of expert programmers and programming language designers are no longer appropriate for developing facilities to be used by novices with varied backgrounds.

In Section 2 we present our model of programmer behavior. In Section 3 the experiments that led to this model are presented and future experiments are proposed. Section 4 is a summary with conclusions.

2. A COGNITIVE VIEW OF PROGRAMMER BEHAVIOR

Any model of programmer behavior must be able to account for five basic programming tasks:

- composition: writing a program,
- comprehension: understanding a given problem,
- debugging: finding errors in a given program,

- modification: altering a given program to fit a new task,
- learning: acquiring new programming skills and knowledge.

In addition, a cognitive model must be able to describe these tasks in terms of

- *the cognitive structures* that programmers possess or come to possess in their memory, and
- *the cognitive processes* involved in using this knowledge or in adding to it.

Recent developments in the information processing approach[10] to the psychology of learning, memory, and problem solving have suggested a framework for discussing the components of memory involved in programming tasks (see Fig. 1). Information from the outside world, to which the programmer pays attention, such as descriptions of the to-be-programmed problem, enters the cognitive system into *short-term memory*, a memory store with a relatively limited capacity (Miller[15] suggests about seven chunks) and which performs little analysis on the input information. The programmer's permanent knowledge is stored in *long-term memory*, with unlimited capacity for organized information. The component labeled *working memory* by Feigenbaum[8] represents a store that is more permanent than short-term but less permanent than long-term memory, and in which information from short-term and long-term memory may be integrated and built into new structures. During problem solving (e.g., generation of a program) new information from short-term memory and existing relevant concepts from long-term memory are integrated in working memory, and the result is used to generate a solution or, in the case of learning, is stored

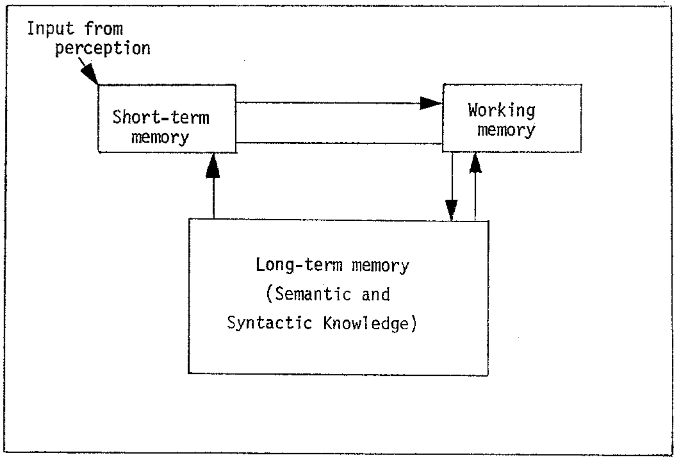

Fig. 1. Components of memory in problem solving.

in long-term memory for future use. Two questions are posed by this mode What kind of knowledge (or *cognitive structures*) is available to the programmer in long-term memory? What kind of processes (*or cognitive processes*) does the programmer use in building a problem solution in working memory?

2.1. Multileveled Cognitive Structures

The experienced programmer develops a complex multileveled body of knowledge—stored in long-term memory—about programming concepts and techniques. Part of that knowledge, called *semantic knowledge*, consists of general programming concepts that are independent of specific programming languages. Semantic knowledge may range from low-level notions of what an assignment statement does, what a subscripted array is, or what data typas are; to intermediate notions such as interchanging the contents of two registers, summing up the contents of an array, or developing a strategy for finding the larger of two values; to higher level strategies such as binary searching, recursion by stack manipulation, or sorting and merging methods. A still higher level of semantic knowledge is required to solve problems in application areas such as statistical analysis of numerical data, stylistic analysis of textual data, or transaction handling for an airline reservation system. Semantic knowledge is abstracted through programming experience and instruction, but it is stored as general, meaningful sets of information that are more or less independent of the syntactic knowledge of particular programming languages or facilities such as operating systems languages, utilities, and subroutine packages.

Syntactic knowledge is a second kind of information stored in long-term memory; it is more precise, detailed, and arbitrary (hence more easily forgotten) than semantic knowledge, which is generalizable over many different syntactic representations. Syntactic knowledge involves details concerning the format of iteration, conditional or assignment statements valid character sets; or the names of library functions. It is apparently easier for humans to learn a new syntactic representation for an existing semantic construct than to acquire a completely new semantic structure. This is reflected in the observation that it is generally difficult to learn the first programming language, such as FORTRAN, PL/1, BASIC, and PASCAL, but relatively easy to learn a second one of these languages. Learning a first language requires development of both semantic concepts and specific syntactic knowledge, while learning a second language involves learning only a new syntax, assuming the same semantic structures are retained. Learning a second language with radically different semantics (i.e., underlying

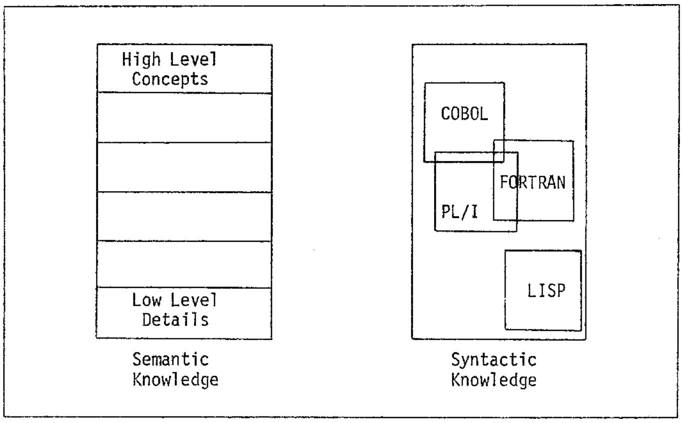

Fig. 2. Long-term memory.

basic concepts) such as LISP or MICRO-LANNER may be as hard or harder than learning a first language.

The distinction between semantic and syntactic knowledge in the programmers' long-term memories is summarized in Fig. 2. The semantic knowledge is acquired largely through intellectually demanding, meaningful learning, including problem solving and expository instruction, which encourages the learner to "anchor" or "assimilate" new concepts within existing semantic knowledge or "ideational structure."(2) Syntactic knowledge is stored by rote, and is not well integrated within existing systems of semantic knowledge. The acquisition of new syntactic information may interfere with previously learned syntactic knowledge, since it may involve *adding* rather than *integrating* new information. This kind of confusion is familiar to programmers who develop skills in several languages and find that they interchange syntactic constructs among them. For example, PASCAL students with previous training in FORTRAN find assignment statements simple, but often err while coding by omitting the colon in the assignment operator and the semicolon to separate statements.

Our discussion of the two kinds of knowledge structures involved in computer programming parallels similar distinctions in mathematics learning. The gestalt psychologists distinguished between "structural understanding" and "rote memory,"(32) between "meaningful apprehension of relations" and "senseless drill and arbitrary associations,"(11) between knowledge which fostered "productive reasoning" and "reproductive reasoning."(13,32) The flavor of the distinction is indicated by an example cited by Wertheimer,(32) suggesting two kinds of knowledge about how to find the area of a parallelogram: knowledge of the memorized formula, $A = h \times b$; and structural understanding of the fact that a parallelogram may be converted into a rectangle by cutting off a triangle from one end and placing it on the other. Similarly, Brownell[5] distinguished between "rote" knowledge of arithmetic acquired through memorizing arithmetic facts (e.g., $2 + 2 = 4$) and "meaningful" knowledge such as relating these facts to number theory by working with physical bundles of sticks. More recently, Polya[18] has distinguished between "know how" and "know what," Greeno[10] has made a distinction between "algorithmic" and "propositional" knowledge used in problem solving, and Ausubel[2] distinguished between "rote" and "meaningful" learning outcomes. Although these distinctions are vague, they reflect a basic distinction, similar to our concept of syntactic and semantic knowledge, that is relevant for computer programming. In his parody of the "new math," Tom Lehrer made a distinction between "getting the right answer" and "understanding what you are doing" (with new math emphasizing the latter). In both mathematics and computer science, however, it seems clear that a compromise is needed between syntactic knowledge and knowledge that provides direction for creating strategies of solution, i.e., semantic knowledge.

2.2. Multi-Leveled, Funneled Cognitive Processes

To complete the model we must examine the processes involved in problem-solving tasks, such as program composition. The mathematician George Polya[18] suggested that problem solving involves four stages:

1. Understanding the problem, in which the solver defines what is given (initial state) and what is the goal (goal state).
2. Devising a plan, in which a general strategy of solution is discovered.
3. Carrying out the plan, in which the plan is translated into a specific course of action.
4. Checking the result, in which the solution is tested to make sure it works.

2.2.1. *Program Composition*

When a problem is presented to a programmer, we assume it enters the cognitive system and arrives in "working memory" by way of short-term memory, and that in working memory the problem is analyzed and represented in terms of the "given state" and "goal state."[33] Similarly, general information from the programmer's long-term memory (both syntactic and semantic) is called and transferred to working memory for further analysis.

These two steps—transferring, to working memory, a description of the problem from short-term memory, and general knowledge from long-term memory—constitute the first step in program composition.

The second step, devising a general plan for writing the program, follows a pattern described by Wirth(34) as stepwise refinement. At first the problem solution is conceived of in general programming strategies and application-related domains such as graph theory, business transaction processing, orbital mechanics, chess playing, etc. We refer to the programmer's general plan as "internal semantics," and suggest that this internal representation progresses from a very general, to a more specific plan, to a specific generation of code focusing on minute details. This "funneling" view of problem solving from the general to the specific was first popularized by the gestalt psychologist Carl Duncker,(7) based on asking subjects to solve a complex problem "aloud." General approaches occurred first, followed by "functional solutions" (i.e., more specific plans), followed by specific solutions.

A top-down implementation of the internal semantics for a problem would demand that the highest (most general) levels be set first, followed by more detailed analysis. This process, suggested by Polya and Wickelgren as "working backwards" or "reformulating the goal" (from the general goal to the specifics) is one technique used by humans in problem solving. A bottom-up implementation would permit low-level code to be generated first, in an attempt to build up to the goal. This process, referred to as "working forward" or "reformulating the givens," where the "givens" include the permissible statements of the language, is another problem-solving technique. Apparently, some types of problems are better solved by one or the other, or both of these techniques.

Structured programming, and particularly the idea of modularization, is another technique that aids in the development of the internal semantics.(6,17,36) Polya and Wickelgren refer to this technique as making "subgoals."

Each of these techniques leads to a funneling of the internal semantics from a very general to a specific plan. Then code may be written, and the program run, as a test. These steps are summarized in Fig. 3. Shneiderman(22) describes design processes and implementation approaches for programs and data.

This model of program composition suggests that once the internal semantics have been worked out in the mind of the programmer, the construction of a program is a relatively straightforward task. The program may be composed easily in any programming language with which the programmer is familiar, and which permits similar semantic constructs. An experienced programmer fluent in multiple languages will find it of approximately equal

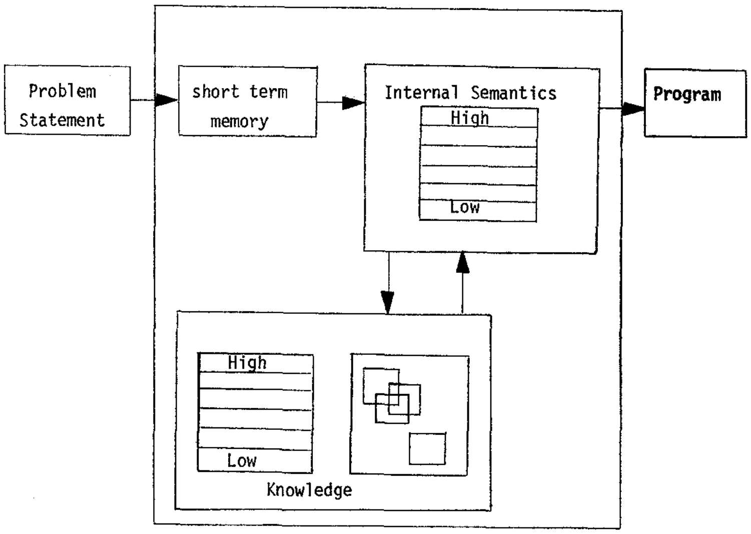

Fig. 3. Program composition process.

ease to implement a table look-up algorithm in PASCAL, FORTRAN, PL/1, or COBOL.

2.2.2. *Program Comprehension*

The program comprehension task is a critical one because it is a subtask of debugging, modification, and learning. The programmer is given a program and is asked to study it. We conjecture that the programmer, with the aid of his or her syntactic knowledge of the language, constructs a multileveled internal semantic structure to represent the program. At the highest level the programmer should develop an understanding of what the program does: for example, this program sorts an input tape containing fixed-length records, prints a word frequency dictionary, or parses an arithmetic expression. This high-level comprehension may be accomplished even if low-level details are not fully understood. At lower semantic levels the programmer may recognize familiar sequences of statements or algorithms. Similarly, the programmer may comprehend low-level details without recognizing the overall pattern of operation. The central contention is that programmers develop an internal semantic structure to represent the syntax of the program, but that they do not memorize or comprehend the program in a line-by-line form based on the syntax.

The encoding process by which programmers convert the program to internal semantics is analogous to the "chunking" process first described by George Miller in his classic paper, "The Magical Number Seven Plus or Minus Two."[15] Instead of absorbing the program on a character-by-character basis, programmers recognize the function of groups of statements, and then piece together these chunks to form ever larger chunks until the entire program is comprehended. This chunking or encoding process is most effective in a structured programming environment, where the absence of arbitrary GOTOs means that the function of a set of statements can be determined from local information only. Forward or backward jumps would inhibit chunking, since it would be difficult to form separate chunks without changing attention to various parts of the program.

Once the internal semantic structure of a program is developed by a programmer, this knowledge is resistant to forgetting and accessible to a variety of transformations. Programmers could convert the program to another programming language or develop new data representations or explain it to others with relative ease. Figure 4 represents the comprehension process.

2.2.3. *Debugging and Modification*

Debugging is a more complex task, since it is an attempt to locate an error in the composition task. We exclude syntactic bugs which are recognizable by a compiler, since these bugs are the result of a trivial error in the

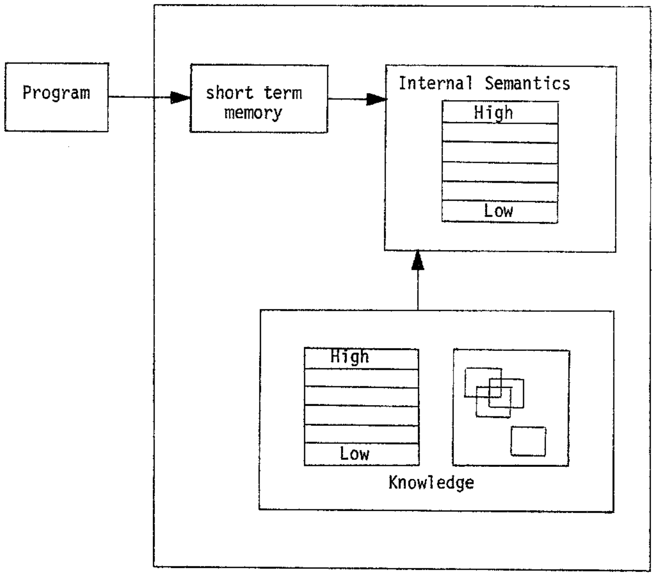

Fig. 4. Program comprehension process: the formation of internal semantics for a given program.

preparation of a program or of erroneous syntactic knowledge that can be resolved by reference to programming manuals. We are left with two further types of bugs: those that result from an incorrect trasnformation from the internal semantics to the program statements, and those that result from an incorrect transformation from the problem solution to the internal semantics.

Errors that result from erroneous conversion from the internal semantic to the program statements are detectable from debugging output which differs from the expected output. These errors can be caused by improper understanding of the function of certain syntactic constructs in the programming language, or simply by mistakes in the coding of a program. In any case sufficient debugging output will help to locate these errors and resolve them.

Errors that result from erroneous conversion from the problem solution to the internal semantics may require a complete reevaluation of the programming strategy. Examples include failure to deal with out-of-range data values, inability to deal with special cases such as the average of a single value, failure to clear critical locations before use, and attempts to merge unsorted lists.

In the modification task the first step is the development of internal semantics representing the current program. The statement of the modification must be reflected in an alteration to the internal semantics, followed by an alteration of the programming statements. The modification task requires skills gained in composition, comprehension, and debugging.

2.2.4. Learning

Finally, we examine the learning task, the acquisition of new programming knowledge. We start with the training of nonprogrammers in their much debated "first course in computing" (*SIGCSE Proceedings*). The classic approach focused on teaching the syntactic details of a language, and used language reference manuals as a text. Much attention was paid to exhaustive discussions of the details of each syntactic structure, with minimal time spent on motivational material or problem solving. Tests focused on statement validity and the determination of what output was produced by a tricky program fragment which exploited obscure features.

By contrast, the problem-solving approach suggested that high-level, language-independent problem solving was the course goal and that coding was a trivial detail not worth the expense of valuable thinking time. Tests in these courses required students to cleverly decompose problems and produce insightful solutions to highly abstract and unrealistic problems.

Of course, both of these descriptions are caricatures of reality, but they point out the differences in approaches. The classic approach concentrated on the development of syntactic knowledge and produced "coders" while the problem-solving approach concentrated on the development of semantic knowledge and produced high-level problem solvers who were unsuited to a production environment. Neither of these approaches is incorrect, they merely have different goals. A reasonable middle ground, the development of syntactic and semantic knowledge in parallel, is pursued by most educators.(26)

Education for advanced programmers also has the syntactic/semantic dichotomy. Courses in the design of algorithms focus on semantic knowledge and attempt to isolate syntactic details in separate discussions or omit them completely. Courses on second or third programming languages can concentrate on the syntactic equivalents of already understood semantics. This makes it unwieldy to teach nonprogrammers and programmers a new language in the same course.

Learning a language that has radically different semantic structures may be difficult for an experienced programmer, since previous semantic knowledge can interfere with the acquisition of a new language. Learning a new language that has similar semantic structures, such as FORTRAN and BASIC, is relatively easy, since most of the semantic knowledge can be applied directly, although confusion of syntax may interfere.

In summary, we conjecture that the semantic knowledge and syntactic knowledge form two separate classes, but that there is a close relationship between them. The multilevel structure of semantic knowledge, acquired largely through meaningful learning, is replicated in the multilevel approach to the development of internal semantics for a particular problem. The syntactic knowledge, acquired largely by rote learning, is compartmentalized by language. The semantic knowledge is essential for problem analysis, while syntactic knowledge is useful during the coding or implementation phase.

Machine-related details such as range of integer values and execution speed of certain instructions, and compiler-specific information such as experience with diagnostic messages are more closely tied to the language-specific syntactic information. This information is highly detailed, learned by repeated experience, and easily forgotten.

3. NEW EXPERIMENTAL EVIDENCE

The model was developed after examining the evidence acquired from a series of experiments briefly described in this section. Our original motivation in pursuing controlled psychological experiments in programming was to assess programming language features, develop standards for stylistic considerations (such as meaningful variable names and commenting), and to validate the design techniques that have been so vigorously debated (top-down design, modularity, and flowcharting) (see Ref. 21 for a discussion with references, also Ref. 9, 16, 22, 30, and 31).

As a result of our experiments and other research we have formulated the model presented in the preceding section, and now have a hypothesis on which to organize future experiments. We hope that future work will not only refine our notion of programmer behavior, but also lead to improved languages, proper stylistic standards, practical design methodologies, new debugging techniques, programmer aptitude tests, programmer ability measures, metrics for problem and program complexity, and improved teaching techniques.

3.1. Arithmetic vs. Logical IF

Our first two experiments (see Shneiderman[23] for a detailed statistical report), carried out by Mao-Hsian Ho, were simple and had modest goals. In one we sought to compare the comprehensibility of arithmetic and logical IF statements in short FORTRAN programs. Our subjects were 24 first-term programming students who had been taught both forms, and 24 advanced programmers who were expected to be familiar with both forms. The novices did better with the logical IF statements, as measured by multiple choice and fill-in-the-blank-type questions, but the advanced programmers did equally well with both forms. We felt that the novices were struggling with the greater syntactic complexity of the arithmetic IF, but that the advanced subjects could easily convert the syntax of the arithmetic IF into the internal semantic form. The advanced students apparently thought about the program on a more general level than did the novices. This was confirmed by discussions with the subjects, and agrees with reports from other sources. The syntactic form of the logical IF seems to be close to the internal semantic form that most programmers perceive. Recent texts support this contention, and sometimes blatant attacks have been made on the use of the arithmetic IF.[37] Still, older programmers who were first taught the arithmetic IF stick to it, finding that they can easily switch from their internal semantic form to the syntactic representation with an arithmetic IF. An experiment with longer, more complex programs would be useful in determining whether the easy conversion breaks down in more difficult situations.

3.2. Memorization

Our second experiment[23] carried out by Mao-Hsian Ho, was a memorization task. Two short programs, about 20 FORTRAN statements, were keypunched, and the first program was listed on a printer. The second program was shuffled and listed. Seventy-nine subjects, ranging from nonprogrammers to experienced professionals, were asked to memorize the two listings, one at a time, and to write back what they could remember. The nonprogrammers did approximately equally poorly on both listings, while the professionals performed poorly on the shuffled program but excelled in recalling the proper executable program. Programmers with greater experience tended to perform better on the proper executable program. Our interpretation was that the advanced programmers attempted to convert specific code into a more general internal semantic representation during program comprehension, while novices focused more on specific code. Advanced subjects constructed a multileveled internal semantic structure to represent the proper executable program, but could not perform this process on the shuffled program; and novices lacked the semantic knowledge to perform this process. This was confirmed by reports from the advanced subjects, who indicated that they could describe the function of the entire program, and that they remembered by realizing that a segment of the program tested a value and then incremented a pair of locations to accumulate sums and counts. Further support for our internal sementics model was gained by studying the written forms. Advanced subjects would recreate semantically equivalent programs that had syntactic variations such as interchanged order of statements, consistent replacement of format numbers, consistent replacement of statement labels, and consistent replacement of variable names. Recall errors of advanced programmers tended to retain the meaning of the program but not the syntax, a finding consistent with human memory for English prose.(3,20) It was these facts that first led us to propose that subjects were not really memorizing the program, but were constructing internal semantics to represent the program's function. When asked to recall the program, they applied their knowledge of FORTRAN syntax and converted their internal semantics back into a FORTRAN program.

3.3. Commenting and Mnemonic Variable Names

Two other experiments, carried out by Ken Yasukawa and Don McKay, sought to measure the effect of commenting and mnemonic variable names on program comprehension in short, 20–50 statement FORTRAN programs. The subjects were first- and second-year computer science students. The programs using comments (28 subjects received the noncommented version, 31 the commented) and the programs using meaningful variable names (29 subjects received the mnemonic form, 26 the nonmnemonic) were statistically significantly easier to comprehend as measured by multiple choice questions. This experiment confirms common practice, but gives no insight into which kind of comments or mnemonic names are helpful and which are not. Further experiments to develop proper standards would be useful.

Our interpretation in terms of the model are that the mnemonic names simplified the conversion from the program syntax to the internal semantic structure of the program. Nonmeaningful variable names place an extra burden on the programmer to encode the meaning of the variable, and add complexity to the conversion process. The internal semantics relate to the meaning and use of a variable, not to the particular variable name, but a meaningful variable name that conveys the variable's function simplifies the programmer's task. The comments serve a somewhat different function. Again, the comments are not stored in the internal semantic structure, but they facilitate the conversion by describing the function of a statement or group of statements. This notion conforms to programming practice, which urges functional descriptive comments, not low-level comments that reiterate the operation of a particular statement. For example, a bad comment for the statement $I = I + 1$ would be "ADD ONE TO THE VARIABLE I." Useful comments are those that facilitate the construction of the internal semantics by describing the meaning of a group of operations such as "SEARCH FOR THE LARGEST VALUE IN THE TABLE." Better still would be application domain comments such as "BINARY SEARCH FOR STUDENT WITH HIGHEST GRADE."

In a debugging task which followed the commenting experiment, 12 out of 28 (42 %) subjects located a bug in a commented program, while only 10 out of 30 (33 %) subjects (one subject dropped out) located the same bug in an uncommented version.

Although this result was not statistically significant, it favored the commented form. Comments should facilitate the construction of an internal semantic structure to describe what the program is supposed to do. The expected internal semantic structure can then be compared to the actual program.

3.4. Modularity

The next area of study for our experiments was modular program design, investigated by Robert Kinicki and Mary Ramsey. The subjects were assembly language students in two groups: those learning the Texas Instrument 980A machine, and those learning the COMPASS assembly language for the Control Data 6600 computer. The 30 TI980A students were divided into three groups of 10 subjects who received the same program written in different forms:

1. Modular—each module has an explicit function (10-line main program and three subroutines: 13, 13, and 22 lines).

2. Nonmodular—unseparated sequential code (54 lines).
3. Random modular—a program broken into subroutines without clear function (8-line main program, four subroutines: 10, 17, 8, and 19 lines).

All subjects took a comprehension test, which produced the following average scores (100 was a perfect score): modular, 89.5; nonmodular, 77.3; random modular, 67.9. An analysis of variance indicated group differences significant at the .08 level. This expected result confirmed the popular statements about the utility of modular programming, but underlined the importance of the proper selection of modules.

Poor decomposition can make a program more difficult to comprehend. A closer, informal examination of the data showed that some of the best students in the class were assigned to the random modular group, and they were capable of achieving high scoress in spite of the difficulty of the progrm. Excellent programmers can perform surprisingly well even in adverse conditions.

The results with the COMPASS students on the modularity experiment were less clear-cut. The three test forms of the COMPASS program were distributed to the 39 students in three groups of 13 each. The averages on the comprehension test were: modular, 47.8; nonmodular, 60.8; random modular, 57.8. The generally poorer scores and lack of significant differences among the groups were attributed to the differences in teaching techniques and the added complexity of subroutines in COMPASS. Apparently the instructor in this course had not emphasized subroutines, and had not required subroutines in homework problems. As a result, subjects suffered from the added complexity of subroutine invocation and argument passing.

This experiment reinforces our belief that modular program construction can be more difficult if programmers have not had adequate training, but that modularity is helpful to experienced programmers for program comprehension. Programmers who have not developed the syntactic and semantic knowledge to support modular programming have an extremely difficult time in developing the proper internal semantic structure for program comprehension. Experienced programmers who understand modular programming can make good use of this technique in developing the internal semantic structure necessary for program comprehension. Modular program design facilitates the chunking process, allowing the programmer to concentrate on a small portion of the program and to encode that portion into higher level concepts. The random modular program is just a sequence of statements that perform no obvious coherent function and cannot be encoded into a higher level chunk.

3.5. Flowcharting

A more recent series of experiments carried out by Peter Heller and Don McKay[24] were designed to test the utility of detailed flowcharts in program composition, comprehension, debugging, and modification. Although flowcharts have long been a staple of programming practice and education, there are now an increasing number of critics. One of the strongest attacks was by Brooks,[4] who wrote that, "The flow chart is a most thoroughly oversold piece of documentation... the detailed blow-by-blow flow chart... is an obsolete nuisance." Our experiments were conducted both with Indiana University students—whose training did not emphasize the use of flowcharts—and with Purdue students—whose training did emphasize the use of flowcharts. For comprehension and debugging tasks there was no overall difference in performance between students given microflowcharts, macroflowcharts, and no flowcharts. However, a closer analysis revealed an interaction in which Indiana students performed worse with flowcharts as compared with no flowcharts, but the Purdue student performance was better with flowcharts as compared with no aids. In a modification task using a longer program a similar pattern was found for the second of two problems.

These results indicate that flowcharts may be an aid in some situations and a hindrance in others. Apparently, a flowchart may serve either as an aid in the translation process from syntax to semantics (as the Purdue students hinted) or merely as an alternative syntactic representation of the program and as such may actually interfere with the creation of the internal semantic structure (as the Indiana students hinted). The resolution of the "flowchart question" seems to depend not on flowcharts *per se*, but on the larger question of what types of supplementary representation help programmers build the internal semantics.

3.6. Commenting

An often expressed belief is that the more comments a program has the better it is. In a recent experiment[25] we tested this idea within the framework of our cognitive model, which suggests that high-level commen s help develo the hierarchical internal semantic structure while low-level comments might interfere with comprehension since they are merely alternate syntactic representations of the function of a single statement. A 26-line FORTRAN program was prepared in two versions: one with only a five-line, high-level comment block at the beginning, and the other with numerous interspersed low-level comments describing the function of the following one or two lines. Two groups of 30 students each were given the listing and asked to make three independent modifications to the program. Following this segment of the experiment, all subjects were asked to study the program for 5 min more and were given 10 min to write back as many of the program statements (not comments) as they could remember. Finally, the students were asked to make subjective judgments as to the difficulty of the task. The final grades of each of the subjects, who were just completing their first course in programming, were also available.

The modifications were graded by an experienced graduate teaching assistant, and the memorization scores were based on the number of lines that were attempted and number of lines that were precisely correct. The results showed statistically significant differences between the two groups. Both the modification and the memorization scores were better for the high-level comment group. Performance on the modifications was strongly correlated with course performance, indicating that the modification task required programming skills developed in the course.

The three modification tasks were designed to be increasingly difficult. There was no statistically significant difference in group performance for the first modification, which was the easiest. The two more difficult modifications showed clear differences, suggesting that, as task complexity increases, commenting techniques play a more important role.

The results of the memorization experiment reinforce our belief that ability to memorize and recall a program is a strong correlate of program comprehension—this experiment utilized a modification task as a measure of program comprehension. Furthermore, we feel that memorization and comprehension are accomplished by a hierarchical chunking process that organizes several statements into a functional unit. These units can then be organized into still higher level units which convey the overall operation of the program. Within this framework the high-level comments facilitate this organizing process, while the low-level comments inhibit it by distracting the reader and simply offering a repetition of the statements whose function is clear to a knowledgeable programmer.

4. FUTURE RESEARCH

We have attempted to present a cognitive model of programmer behavior that was developed in response to controlled psychological experiments. This cognitive model separates the syntactic knowledge from semantic knowledge, and emphasizes the internal representation created by the programmer in the programming tasks of composition, comprehension, debugging, modification, and learning.

Future experiments must focus on the verification of this model. In particular we are interested in trying to study the components of the internal semantic model and the encoding process across a range of subject experience. It is important to find out what encodings are used by different programmers with different amounts of experience in different languages. Results of this kind of research would be significant for programming-language designers and for educators.

One possible experiment is to ask subjects with varying experience to take a program and mark the segments of the program with a series of nested brackets on the lefthand side of the page. We expect that, as subject experience increases, an increased depth of nesting will occur, since more knowledgeable subjects have a greater capacity to organize the program into a hierarchical semantic structure.

Another approach is to ask subjects to insert blank cards in a program to clarify the structure of a program. This kind of experiment may help give us some insight into what encodings the subjects perceive. A cross-sectional study of subjects with different levels of experience might reveal patterns of skill acquisition.

A promising direction of research is the use of memorization and recall experiments. First, more work must be done to validate the hypothesis that recall is a measure of comprehension. Then memorization and recall can be used to study stylistic issues (such as commenting, mnemonic variable names, and indentation), language features (such as recursion/iteration, block structuring, data types, and control structures), and design techniques (such as modularity, top-down design, and bottom-up design).

In the future we look to a clarification of the cognitive processes in problem solving and programming. Such an understanding would lead to improved programming languages whose syntactic structure more closely reflectes internal semantic structures, thereby easing the programming process. Machine efficiency issues must be temporarily ignored while programming is studied from a purely human viewpoint. Then we can discuss efficient implementations of what programmers consider convenient semantic structures.

Simplifying the programming process and making it easier for a wider range of people to use computers are the ultimate aims of this research direction. Computer scientists should welcome the contributions of and cooperation with cognitive psychologists. Interaction between the two disciplines will benefit both.

REFERENCES

1. J. D. Aron, *The Program Development Process: Part 1, The Individual Programmer* (Addison-Wesley, Reading, Massachusetts, 1974).
2. D. P. Ausubel, *Educational Psychology: A Cognitive Approach* (Holt, Rinehart and Winston, New York, 1968).

3. J. Bransford and J. Franks, "Abstraction of linguistic ideas," *Cognit. Psychol.* **2**:331–350 (1971).
4. Frederick Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, Reading, Massachusetts, 1975).
5. W. A. Brownell, "Psychological Considerations in the Learning and Teaching of Arithmetic," in *The Teaching of Arithmetic: Tenth Yearbook of the National Council of Teachers of Mathematics*, Bureau of Publications, Teachers College, Columbia University, New York (1935), pp. 1–35.
6. O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming* (Academic Press, London–New York, 1973).
7. K. Duncker, "On problem solving," *Psychol. Monogr.* **58**:270 (1945).
8. E. A. Feigenbaum, "Information Processing and Memory," in *Models of Memory*, D. A. Norman, Ed. (Academic Press, New York, 1970), pp. 451–469.
9. J. D. Gannon and J. J. Horning, "The Impact of Language Design on the Production of Reliable Software," *IEEE Trans. on Software Engineering*, **1** (1975).
10. J. G. Greeno, "The Structure of Memory and the Process of Problem Solving," in *Contemporary Issues in Cognitive Psychology*, R. Solso, (Winston, Washington, 1973).
11. G. Katona, *Organizing and Memorizing* (Columbia University Press, New York, 1940).
12. C. Kreitzberg and L. Swanson, "A Cognitive Model for Structuring an Introductory Programming Curriculum," *AFIPS Proceedings of the National Computer Conference* (AFIPS Press, Montvale, New Jersey, 1974).
13. N. R. F. Maier, "Reasoning in humans, I, On direction," *J. Comp. Psychol.* **12**:115–143 (1930).
14. R. E. Mayer, "Different problem-solving competencies established in learning computer programming with and without a meaningful model," *J. Educ. Psychol.* **68**:143 150 (1976).
15. G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychol. Rev.* **63**;81–97 (1956).
16. L. Miller, "Programming by Non-Programmers," IBM Research Report RC4280 (1973).
17. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM* **15**:1053–1058 (December 1972).
18. G. Polya, *How to Solve It* (Doubleday, New York, 1957).
19. P. Reisner, R. F. Boyce, and D. D. Chamberlin, "Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL," *Proceedings of the National Computer Conference* (AFIPS Press, Montvale, New Jersey, 1975).
20. J. Sachs, "Recognition memory for syntactic and semantic aspects of connected discourse," *Percept. Psychophys.* **2**:437–442 (1967).
21. B. Shneiderman, "Experimental Testing in Programming Languages, Stylistic Considerations and Design Techniques," *Proceedings of the National Computer Conference* (AFIPS Press, Montvale, New Jersey, 1975).
22. B. Shneiderman, "A review of design techniques for programs and data," *Software—Pract. Exper.* **6**:555–567 (1976).
23. B. Shneiderman, "Exploratory experiments in programmer behavior," *Int. J. Comput. Inf. Sci.* **5**(2):123–143 (June 1976).
24. B. Shneiderman, R. Mayer, D. McKay, and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Commun. ACM* **20**:373–381 (1977).

25. B. Shneiderman, "Measuring computer program quality and comprehension," *Int. J. Man-Mach. Stud.* **9**:465–478 (1977).
26. B. Shneiderman, "Teaching programming: A spiral approach to syntax and semantics," *Comput. Educ.* **1**:193–197 (1977).
27. M. Sime, T. Green, and D. Guest, "Psychological evaluation of two conditional constructions used in computer languages," *Int. J. Man-Mach. Stud.* **5**:105–113 (1973).
28. J. C. Thomas and J. D. Gould, "A Psychological Study of Query by Example," *Proceedings of the 1975 National Computer Conference* (AFIPS Press, Montvale, New Jersey, 1975).
29. G. M. Weinberg, *The Psychology of Computer Programming* (Van Nostrand–Reinhold, New York, 1971).
30. L. Weissman, "Psychological Complexity of Computer Programs: An Initial Experiment," Technical Report CSRG-26, Computer Systems Research Group, University of Toronto, Toronto, Canada (1973).
31. L. Weissman, "Psychological complexity of computer programs: An experimental methodology, *SIGPLAN Not.* **9**:25–36 (June 1974).
32. M. Wertheimer, *Productive Thinking* (Harper & Row, New York, 1959).
33. W. Wickelgren, *How to Solve Problems* (W. H. Freeman, San Francisco, 1974).
34. Niklaus Wirth, "Program development by stepwise refinement," *Commun. ACM* **14**:4 (April 1971).
35. E. A. Youngs, "Human errors in programming," *Int. J. Man-Mach. Stud.* **6**:361–376 (1974).
36. H. Mills, "Top Down Programming in Large Systems," in R. Rustin (Ed.), *Debugging Techniques in Large Systems* (Prentice-Hall, Englewood Cliffs, N. J., 1971).
37. D. McCracken, *A Simplified Guide to FORTRAN IV Pragramming*, (John Wiley and Sons, New York, 1974).