TOWARDS A COGNITIVE MODEL OF PROGRAMMER BEHAVIOR

Ben Shneiderman

Computer Science Department

Richard Mayer

Department of Psychology

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT No. 37

TOWARDS A COGNITIVE MODEL OF PROGRAMMER BEHAVIOR

BEN SHNEIDERMAN

RICHARD MAYER

AUGUST, 1975

Towards a Cognitive Model of Programmer Behavior

Ben Shneiderman

Computer Science Department

Richard Mayer

Department of Psychology

Indiana University

Bloomington, Indiana 47401

Abstract

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.

Keywords

programming, programming languages, cognitive models, program composition, program comprehension, debugging, modification, learning, education, information processing.

CR Categories

1.5, 4.0, 4.2

1. Introduction

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

Stimulated by Weinberg's insightful text, The Psychology of Computer Programming (1971), and the improvements promoted by structured programming advocates, researchers have begun to deal with the cognitive processes of programmers. This research has been in the form of controlled experiments, protocol analyses and case studies on individuals or groups (Sime, Green and Guest, 1973; Miller, 1973; Reisner, Boyce, and Chamberlin, 1975; Shneiderman, 1975a, 1975b; Thomas and Gould, 1975; Weissman, 1973, 1974; Young, 1974; and Gannon and Horning, 1975). The tasks studied have included program composition, comprehension, debugging, modification and the learning of new programming skills. A wide range of subjects, from non-programmers to a 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 recently appeared in books by Joel Aron, The Program Development Process Part I (1974), and Fredrick Brooks, Jr., The Mythical Man (1975).

A final area of importance is programming education. Research in 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 (Mayer, 1975; and Kreitzberg and Swanson, 1974) and provide 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 which 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 which led to this model are presented and future experiments are proposed. In Section 4 is a summary with conclusions.

2. A Cognitive Theory of Programming Behavior

Any theory 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 theory must be able to describe these tasks in terms of

the cognitive structures that programmer possesses or comes to possess in his memory, and

the congitive processes involved in using this knowledge or in adding to it.

Recent developments in the information processing approach (Greeno, 1974) to the psychology of learning, memory and problem solving have suggested a framework for discussing the components of memory involved in programming tasks (see Figure 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, 1956, 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 (Feigenbaum, 1974) 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 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 in long-term memory for future use. Two main questions posed by the model summarized in Figure 1 are, what kind of knowledge (or cognitive structures) are available to the programmer in long-term memory, and what kind of processes (or cognitive processes) does the programmer use in building a problem solution in working memory.

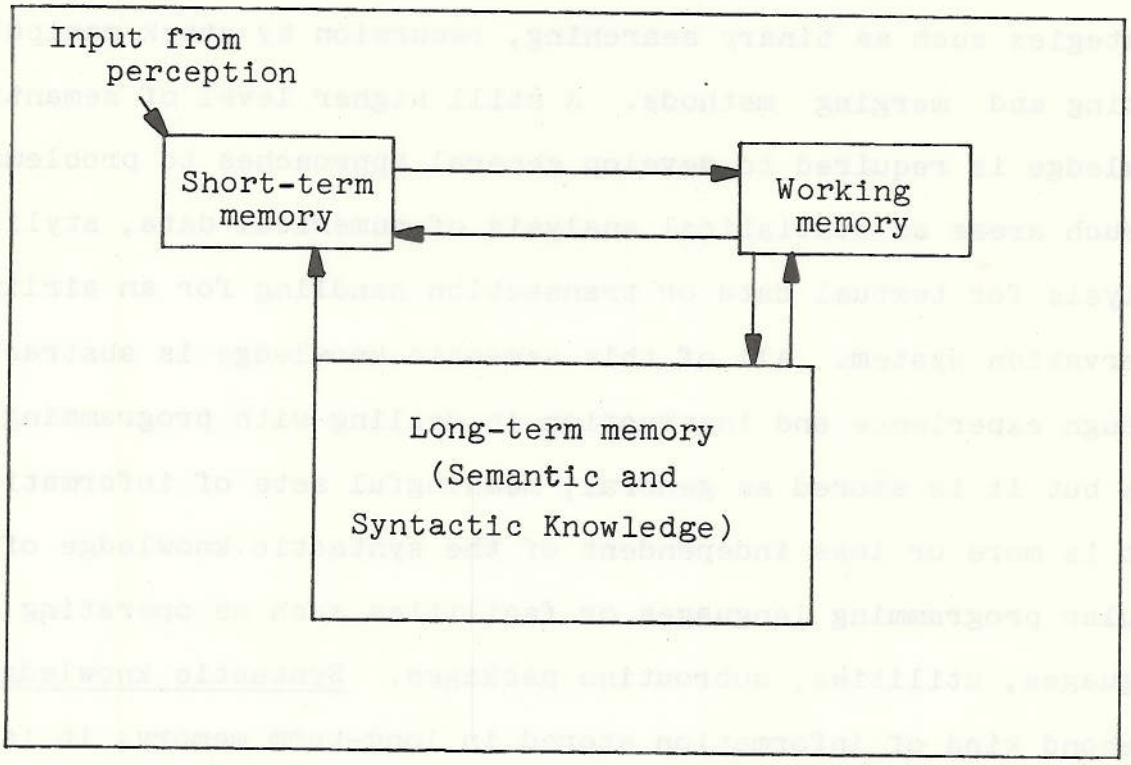

Figure 1: Components of Memory in Problem Solving

2.1 Cognitive Structures are Multi-Leveled

The experienced programmer has developed a complex multi-leveled body of knowledge---stored in long-term memory---about programming concepts and techniques. Part of that knowledge---we will refer to as semantic knowledge---has to do with general concepts important for programming but which are independent of any specific programming language. Semantic knowledge may range from low-level notions of what an assignment statement does, what a subscripted array is, what data types are; to intermediate notions such as interchanging the contents of two registers, summing up the contents of an array, a strategy for finding the larger of two values; to higher level strategies such as binary searching, recursion by stack manipulation, sorting and merging methods. A still higher level of semantic knowledge is required to develop general approaches to problems in such areas as statistical analysis of numerical data, stylistic analysis for textual data or transaction handling for an airline reservation system. All of this semantic knowledge is abstracted through experience and instruction in dealing with programming problems but it is stored as general, meaningful sets of information that is more or less independent of the syntactic knowledge of particular programming languages or facilities such as operating systems languages, utilities, 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, like FORTRAN, PL/1, COBOL, BASIC, PASCAL, etc., 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 a new syntax, assuming the same semantic structures are remained. Learning a second language with radically different semantics (i.e., underlying basic concepts) such as LISP or MICROPLANNER 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 Figure 2.

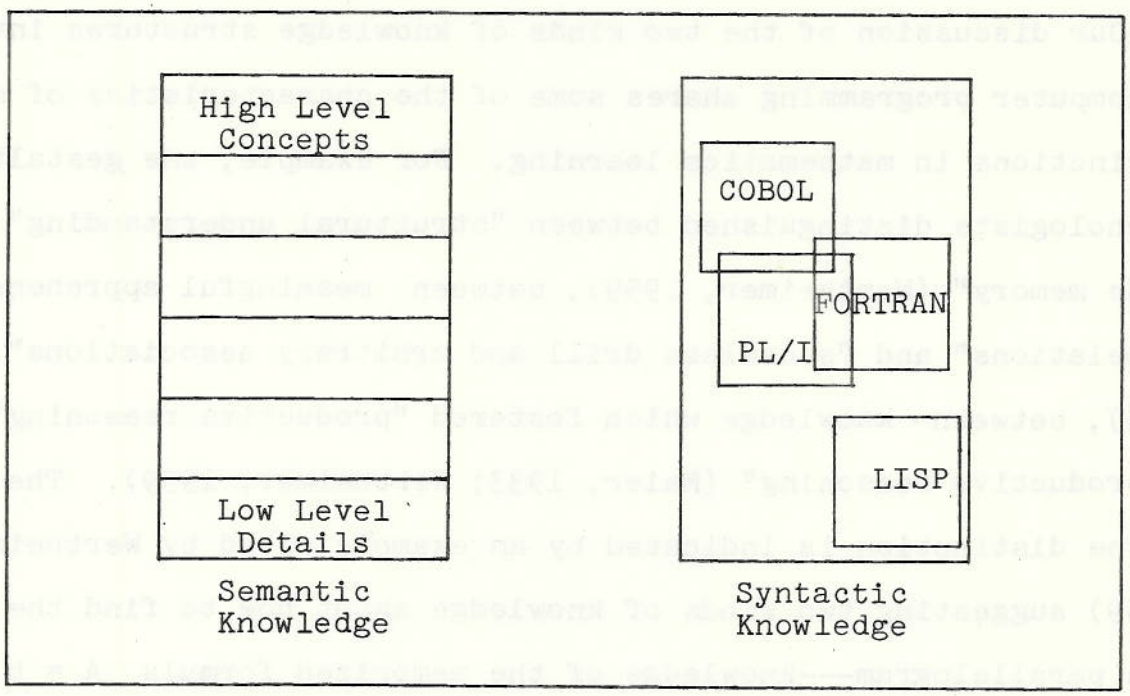

Figure 2: Long-Term Memory

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" (Ausubel, 1968). 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 semi-colon to separate statements.

Our discussion of the two kinds of knowledge structures involved in computer programming shares some of the characteristics of similar distinctions in mathematics learning. For example, the gestalt psychologists distinguished between "structural understanding" and "rote memory" (Wertheimer, 1959), between meaningful apprehension of relations" and "senseless drill and arbitrary associations" (Katona, 1940), between knowledge which fostered "productive reasoning" and "reproductive reasoning" (Maier, 1933; Wertheimer, 1959). The flavor of the distinction is indicated by an example cited by Wertheimer (1959) 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 (1935) distinguished between "rote" knowledge of arithmetic acquired through memorizing arihhmetic 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 (1968) has distinguished between "know how" and "know what", Greeno (1974) has made a distinction between "algorithmic" and "propositional" knowledge used in problem solving, and Ausubel (1968) distinguished between "rote" and "meaningful" learning outcomes. Although these distinction are vague and not fully understood, they do seem to reflect a basic distinction, such as 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 which provides direction for creating strategies of solution, that is, semantic knowledge.

The area of foreign language seems, also, related to computer science. However, programming learning is different from language learning, such as an English-speaker learning French, German, or Spanish, since many of the same underlying concepts (semantics) are involved in these languages and instruction focuses mainly on syntax (e.g., vocabulary memorization, conjugations, formating). To the extent that new languages involve new concepts (such as the subjective tense) or are not "standard" European languages (e.g., Chinese) emphasis on both semantics and syntax is required.

2.2 Cognitive Processes are Multi-Leveled and Funneled

To complete the model we must examine the processes involved in problem solving tasks, such as program composition. The mathematician, George Polya (1966) 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.

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" (Wickelgren, 1974). 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 (1971) as step-wise refinement. At first the problem solution is conceived of in general terms such as general programming strategies and other relevant knowledge such as graph theory, business transaction processing, orbital mechanics, chess playing, etc. We will 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 focussing on minute details. This "funneling" view: of problem solving from the general to the specific was first popularized by the gestalt psychologist Carl Duncker (1945) based on asking subjects to solve 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 Wicklgren 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, refered to as "working forward" or "reformulating the givens" where the "givens" includes 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 to aid in the development of the internal semantics (Dahl, Dijkstra and Hoare, 1973; Mills, 1974; and Parnas, 1972). Polya and Wickelgren refer to this technique as making "subgoals".

Each of these techniques leads to a funneling of the internal semantics from very general to a specific plan. Then code may be written, and the program run, as a test. These steps are summarized in Figure 3.

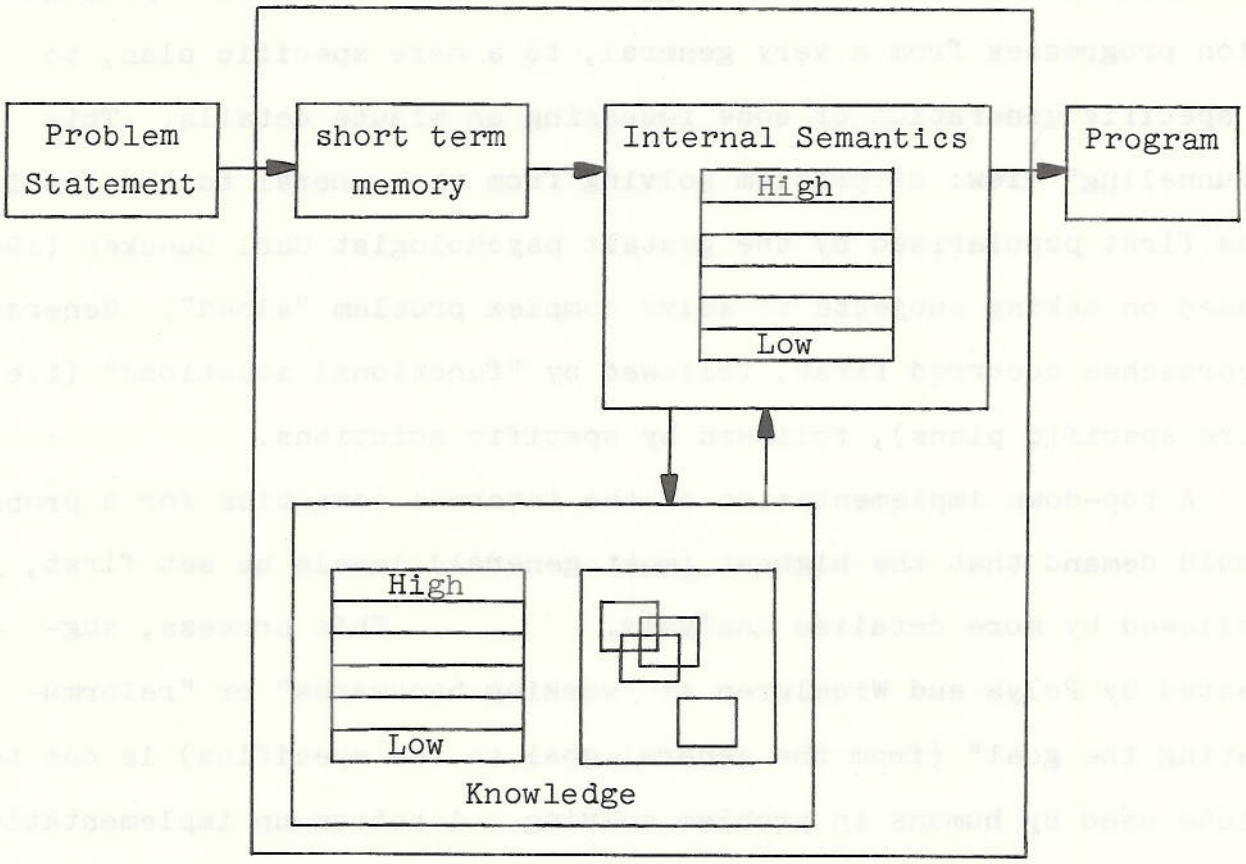

Figure 3: Program Composition Process

This model of program composition represents what we have all known: 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 which the programmer is familiar with and which permits similar semantic constructs. An experienced programmer fluent in multiple languages will find it approximately equally easy to implement a table look-up algorithm in assembly language, FORTRAN, PL/I or COBOL.

The program comprehension task is a critical one since 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/her syntactic knowledge of the language,constructs a multi-leveled 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 based on the "chunking" process first described by George Miller in his classic paper, "The Magical Number Seven Plus or Minus Two" (Miller, 1956). 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 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.

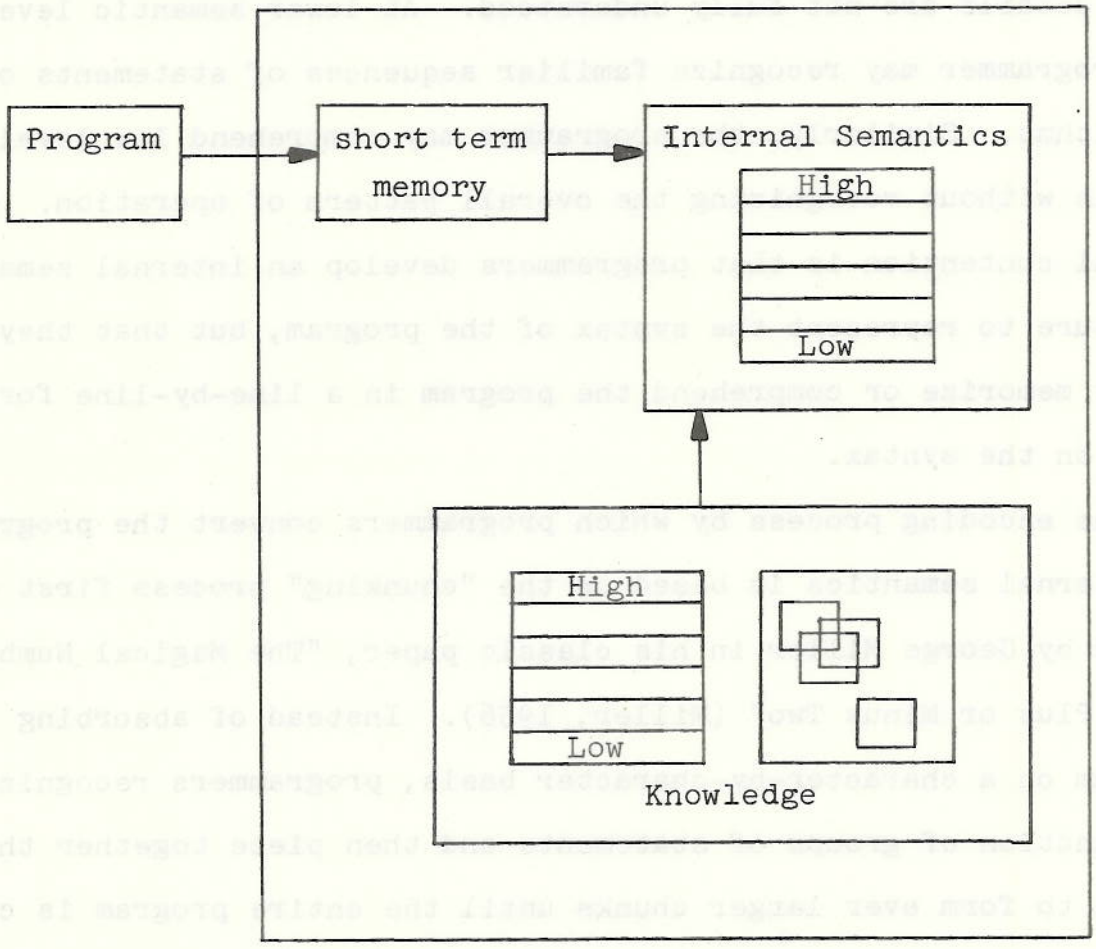

Figure 4: Program Comprehension Process: the formation of internal semantics for a given program

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 a result of a trivial error in the preparation of a program or of erroneous syntactic knowledge which can be resolved by reference to programming manuals. We are left with two further types of bugs: those that result from an incorrect transformation 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 semantics 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 re-evaluation 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 or 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.

Finally, we examine the learning task, the acquisition of new programming knowledge. We start with the training of non-programmers in their much debated "first course in computing" (SIGCSE Proceedings). The classic approach focused on teaching the syntactic details of a language and used manufacturer distributed 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 the validity or invalidity of certain statements and the ability to determine what output was produced by a tricky program fragment which exploited one of the minute details concerning a particular statement.

By contrast the problem solving approach suggested that high level language independent problem solving was the goal of the course and that the actual coding of problems 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 the reality but they point up the differences in approaches. The classic approach concentrated on the development of syntactic knowledge and produced "coders" while the problem solving approach contentrated on the development of semantic knowledge and produced high-level 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.

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 which teach second or third programming languages can concentrate on the syntactic equivalents of already understood semantics. This makes it unwieldy to teach non-programmers and programmers a new language in the same course. Learning a language which has radically different semantic structures is possibly more difficult for an experienced programmer than a non-programmer. The proactive inhibition created by previous semantic knowledge can interfere with the acquisition of a new language. Learning a new language which has similar semantic structures, such as FORTRAN and BASIC, is relatively easy since most of the semantic knowledge can be applied directly.

In summary, we conjecture that the semantic knowledge and syntactic knowledge are independent but that there is a close relationship between them. The multi-level structure of semantic knowledge, acquired largely through meaningful learning, is replicated in the multi-level 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 or execution speed of certain instructions and compiler specific information such as experience with diagnostic messages is more closely tied to the language specific syntactic information. This information is highly detailed, learned by repeated experience and easily subject to forgetting.

3. New Experimental Evidence

The evidence for this model was acquired through a series of experiments conducted during the past year. 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 Shneiderman, 1975a] for a discussion with references; Weissman, 1973, 1974; Gannon and Horning, 1975; Miller, 1973; and Shneiderman, 1975b].

As a result of our experiments and other research we have formulated the model presented in the last 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 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.

Our first two experiments, 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 first term programming students who had been taught both forms and 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-blanks 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 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 have blatant attacks on the use of the arithmetic IF (McCracken, 1974). Still, older programmers who were first taught the arithmetic IF stick to it and find 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 to determine if the easy conversion breaks down in more difficult situations.

Our second experiment, 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. Subjects ranging from non-programmers to experienced professionals were asked to memorize the two listings, one at a time, and to write back what they could remember. The non-programmers 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 multi-leveled 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 semantics model was gained by studying the written forms. Advanced subjects would recreate semantically equivalent programs which 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 consistant with human memory for English prose (Bransford and Franks, 1974; Sachs, 1968). It was these facts which 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.

Two other experiments, carried out by Ken Yasukawa and Don McKay, sought to mearure the effect of commenting and mnemonic variable names on program comprehension in short, twenty to fifty FORTRAN statement, programs. The subjects were first-and second-year computer science students. The programs using comments (28 subjects received noncommented version, 31 the commented) and the programs using mean-

ingful variable names (29 subjects received mnemonic form, 26 the non-mnemonic) were statistically significantly easier to comprehend as measured by multiple choice questions. This experiment confirms common practice but gives no insight into what 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. Non-meaningful 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 which conveys the function of the variable, 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 which 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 which 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".

In a debugging task which followed the commenting experiment, 12 out of 28 subjects located a bug in a commented program while only 10 out of 30 subjects located the same bug in an uncommented version

of the bug. 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.

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 (TI980A) and those learning the COMPASS assembly language for the Control Data 6600 computer. The thirty TI980A students were divided into three groups of ten subjects which received the same program written in different forms:

1. modular - each module has an explicit function (10-line main program and 3 subroutines: 13, 13 and 22 lines)

2. non-modular - unseparated sequential code (54 lines)

3. random modular - a program broken into subroutines without clear function (8-line main program, 4 subroutines: 10, 17, 8 and 19 lines).

All of the subjects took a comprehension test which produced the following average scores (100 was a perfect score):

Modular 89.5

Non-modular 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 underlines the importance of the proper selection of modules.

Poor decompositions 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 scores inspite of the difficulty of the program. 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 subjects in three groups of 13 each. The averages on the comprehension test were:

Modular 47.8

Non-modular 60.8

Random modular 57.8

The generally poorer scores and lack of significant differences a 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 unless programmers have had adequate training, but that modularity is helpful for program comprehension when used by experienced programmers. 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 comprehsnion. 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 encode that portion into higher level concepts. The random modular program is just a sequence of statements which perform no obvious coherent function and cannot be encoded into a higher level chunk.

A more recent series of experiments carried out by Peter Heller and Don McKay were designed to test the utility of detailed flowcharts in program composition, comprehension, debugging and modification. Although flowcharts have long been a staple of the programming practice and education, there are now an increasing number of critics. One of the strongest attacks was by Brooks (1975) who wrote that "the flow chart is a most thoroughly oversold piece of documentation...the detailed blow-by-blow flow chart, however, is an obsolete nuisance." Our experiment 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 micro-flowcharts, macro-flowcharts 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 students 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 may be a hindrance in others. Apparently, flowcharts

may serve either as an aid in the translation process from syntax to semantics (as the Purdue students hinted) or the flowchart may serve merely as an alternative syntactic representation of the program and as such may actually interfer with the creation of the internal semantic structure (as our 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.

4. Summary

We have attempted to present a cognitive model of programmer behavior which 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.

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

In the future, we look to a clarification of the semantic structures and of the chunking process used for various tasks. Such an understanding would lead to improved programming languages whose syntactic structure more closely reflected the semantic structures and thereby eased the programming process. Machine efficiency issues must be temporarily ignored while programming is studied from a purely human factors 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 is the ultimate aim of this research direction. Computer scientists should welcome the contributions of and cooperation with cognitive psychologists. Interaction between two groups will benefit both disciplines.

References

1. Aron, J.D. The Program Development Process: Part 1 The Individual Programmer, Addison-Wesley Publishing Co., Reading, MA, (1974).

2. Ausubel, D.P. Educational Psychology: A Cognitive Approach, New York, Holt, Rinehart & Winston (1968).

3. Bransford, J., and Franks, J. Abstraction of linguistic ideas. Cognitive Psychology 2 (1972), 331-350.

4. Brooks, Frederick, Jr. The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley Publishing Co., Reading, MA, (1975).

5. Brownell, W.A. 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), 1-35.

6. Dahl, O.-J.; E.W. Dijkstra; and C.A.R. Hoare. Structured Programming, Academic Press, London and New York (1973).

7. Duncker, K. On problem solving. Psychological Monographs 58, Whole No. 270 (1945).

8. Feigenbaum, E.A. Information processing and memory. In Models of Memory, D.A. Norman (Ed.), Academic Press, New York (1970), 451-469.

9. Gannon, J.D., and Horning, J.J. The impact of language design on the production of reliable software. Proc. 1975 International Conference on Reliable Software.

10. Greeno, J.G. The structure of memory and the process of problem solving. In Contemporary Issues in Cognitive Psychology, R. Solso (Ed.), Winston, Washington (1973).

11. Katona, G. Organizing and Memorizing, Columbia University Press, New York (1940).

12. Kreitzberg, C., and Swanson, L. A cognitive model for structuring an introductory programming curriculum. AFIPS Proc. National Computer Conference (1974).

13. Maier, N.R.F. Reasoning in humans. I. On direction. Journal of Comparative Psychology 12 (1930), 115-143.

14. Mayer, R.E. Different problem-solving competencies established in learning computer programming with and without a meaningful model. Journal of Educational Psychology, in press.

15. Miller, G.A. The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychological Review 63 (1956), 81-97.

16. Miller, L. Programming by Non-Programmers, IBM Research Report RC4280 (1973).

17. Parnas, D.L. On the criteria to be used in decomposing systems into modules. Comm. ACM 15, 12 (December, 1972).

18. Polya, G. How to Solve It, Doubleday, New York (1957).

19. Reisner, P.; Boyce, R.F.; and Chamberlin, D.D. Human factors evaluation of two data base query languages: SQUARE and SEQUEL. Proc. National Computer Conference, AFIPS Press, Montvale, NJ, (1975).

20. Sachs, J. Recognition memory for syntactic and semantic aspects of connected discourse. Perception and Psychophysics 2 (1967), 437-442.

21. Shneiderman, B. Experimental testing in programming languages, stylistic considerations and design techniques. Proc. National Computer Conference, AFIPS Press, Montvale, NJ (1975).

22. Shneiderman, B.; Mayer, R.; McKay, D.; and Heller, P. Experimental investigations of the utility of flowcharts in programming. Technical Report No. 36, Computer Science Department, Indiana University, Bloomington, IN (1975).

23. Sime, M.; Green, T.; and Guest, D. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies 5, 1 (1973).

24. Thomas, J.C., and Gould, J.D. A psychological study of query by example. Proc. 1975 National Computer Conference, AFIPS Press, Montvale, NJ (1975).

25. Weissman, L. Psychological complexity of computer programs: an initial experiment. Technical Report CSRG-26, Computer Systems Research Group, University of Toronto, Toronto, Canada (1973).

26. ----------. Psychological complexity of computer programs: an experimental methodology. SIGPLAN Notices 9, 6 (June, 1974).

27. Wertheimer, M. Productive Thinking, Harper & Row, New York (1959).

28. Wickelgren, W. How to Solve Problems, Freeman, San Francisco (1974).

29. Wirth, Niklaus. Program development by stepwise refinement. Comm. ACM 14, 4 (April, 1971).

30. Young, E.A. Human errors in programming. International Journal of Man-Machine Studies 6 (1974), 361-376.