COGNITIVE PSYCHOLOGY 19, 295–341 (1987)

Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs

NANCY PENNINGTON

Graduate School of Business, University of Chicago

Comprehension of computer programs involves detecting or inferring different kinds of relations between program parts. Different kinds of programming knowledge facilitate detection and representation of the different textual relations. The present research investigates the role of programming knowledge in program comprehension and the nature of mental representations of programs; specifically, whether procedural (control flow) or functional (goal hierarchy) relations dominate programmers' mental representations of programs. In the first study, 80 professional programmers were tested on comprehension and recognition of short computer program texts. The results suggest that procedural rather than functional units form the basis of expert programmers' mental representations, supporting work in other areas of text comprehension showing the importance of text structure knowledge in understanding. In a second study 40 professional programmers studied and modified programs of moderate length. Results support conclusions from the first study that programs are first understood in terms of their procedural episodes. However, results also suggest that a programmer's task goals may influence the relations that dominate mental representations later in comprehension.

Computer programming is a complex cognitive task composed of a variety of subtasks and involving several kinds of specialized knowledge (Pennington, 1985). A skilled computer programmer must understand the problem to be solved, design a solution, code the solution into a programming language, test the program's correctness, and be able to comprehend written programs. These different aspects of programming require

This research was sponsored by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract N00014-82-K-0759, Contract Authority Identification Number 667-503. Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government. Beatrice Grabowski, Paul Harvell, Lori Hunsaker, John Keating, and Helen Szepe worked on the project as research assistants. Thanks are due to Sargent and Lundy, and Argonne Laboratories for encouraging their programmers to participate in the research, and to many other professional programmers who came to our lab to participate. Thanks are also due to Reid Hastie, Beatrice Grabowski, and Lance Rips for helpful comments on the manuscript and to Gail McKoon for advice on methods. Correspondence, including requests for reprints, should be directed to Nancy Pennington, Center for Decision Research, Graduate School of Business, University of Chicago, 1101 E. 58th Street, Chicago, IL 60637.

0010-0285/87 $7.50

knowledge of the real world problem domain, such as statistics, banking, or physics; knowledge of design strategies and useful design components; knowledge of programming language syntax, text structure rules, and programming conventions; knowledge of computer features that impact program implementation; and knowledge of the user of the program. Central questions in the study of cognitive skills in general and of programming in particular concern the nature of expert knowledge and how various types of knowledge influence skilled performances (Bisanz & Voss, 1981; Chi, Glaser, & Rees, 1982; Kieras, 1985; Miller, 1985).

The present research focuses on the subtask of computer program comprehension, an important part of computer programming skill from both practical and theoretical perspectives. It is estimated that more than 50% of all professional programmer time is spent on "program maintenance" tasks that involve modifications and updates of previously written programs. Because the programs are most often written by other programmers, comprehension plays a central role in this endeavor. From a theoretical perspective, comprehension involves the assignment of meaning to a particular program, an accomplishment that requires the extensive application of specialized knowledge. Thus the study of program comprehension provides an effective means for studying the role of particular kinds of knowledge in cognitive skill domains.

The general approach employed in the present research is to regard a computer program as a text. Because programs are instructions to a computer, the closest analogs among natural language texts are instructions about how to perform a particular task, often referred to as procedural instructions. Procedural instructions and programs also share the feature that the text can be "executed" to accomplish a goal.

One advantage to viewing programs as texts is that theories and methods in the study of text comprehension are relatively well developed so that widely accepted characterizations of text comprehension can serve as a starting point for thinking about the comprehension of programs. According to the dominant view, various knowledge structures (often referred to as schemas or frames) relevant to the text are activated in the course of comprehension of the text (e.g., Adams & Collins, 1979; Rumelhart, 1980). For example, if a person is reading a story about a trip to France, knowledge about how stories typically proceed (a story schema) as well as more specific content knowledge about vacation trips, the parts of France, etc., would be activated. Schemas that are verified (or persist) provide the perspective from which the text is understood, allow the reader to account for and interpret information explicitly mentioned in the text, and enable inferences to be made about information not mentioned. For example, the reader may determine after awhile that

the story involves a business trip rather than a vacation trip so that information initially interpreted in the context of a vacation may be reinterpreted in terms of knowledge about business trips. This process results in a mental representation of the text that is influenced by information and structure in the stimulus text as well as information and structure provided by activated knowledge. The memory representation of the text is assumed to have levels. One of the most widely cited theories distinguishes between a *microstructure* level consisting of propositions and their interrelations that correspond closely to the text and a *macrostructure* level consisting of a smaller number of propositions that characterize the text at a more abstract level (Kintsch & van Dijk, 1978). The theory implies that a key process in text comprehension involves chunking the text into segments that correspond to schema categories so that labels for segments will constitute the macrostructure for the text (Kintsch, 1977). In other words, the structure of activated knowledge is an organizing framework for the mental representation of the text at the macrostructure level. Thus mental representations of text and the related knowledge structures are linked in the comprehension process.

The purpose of the present research is to explore the role of two kinds of programming knowledge—text structure knowledge (Basili & Mills, 1982; Curtis, Forman, Brooks, Soloway & Ehrlich, 1984) and plan knowledge (Soloway & Ehrlich, 1984)—that might describe macrostructures in the construction of mental representations of program texts. These kinds of knowledge have analogs in other text comprehension domains (see Mandler, 1984, as well as Britton & Black, 1985, for many examples); they play a special role in understanding procedural instructions and programs, because complete comprehension of programs (and other texts) requires understanding multiple relations between parts of the text that are difficult to view simultaneously. Thus, the nature of the macrostructure will determine which aspects of the text will be relatively easier or more difficult to understand.

In the sections that follow, our analysis of program comprehension begins with analyses of the computer program stimulus structures. These analyses are *abstractions* of the *text* and they are intended to illustrate features of the text (not mental entities) that may or may not be detected during comprehension. We then describe two kinds of programming *knowledge structures* that are involved in computer program comprehension and propose two alternative hypotheses concerning the kind of knowledge that plays an organizing role in the mental representation of the text. Because there are correspondences between certain abstractions of the text and particular types of knowledge, the kind of knowledge that provides organizing structure in the resulting mental representation

of the text will have implications for which text features are explicitly included in the mental representation. These hypotheses and their implications are tested in two empirical studies.

MULTIPLE ABSTRACTIONS OF COMPUTER PROGRAM TEXT

For computer programs, as for other types of texts, there are different kinds of information implicit "in the text" that must be detected in order to fully understand the program (Green,1980; Green, Sime, & Fitter, 1980; Pennington, 1985). For example, the sequence of statements in the program and certain keywords provide information about the sequence in which program statements will be executed. This kind of information is called the control flow of the program and understanding a program requires understanding its control flow. Another kind of information contained in programs, called the data flow of the program, concerns the changes or constancies in the meaning or value associated with the names of program objects throughout the course of the program.

In the illustration that follows, a sample program text is analyzed in terms of four different kinds of information implicit in the text. Each of these analyses results in an abstraction of the text that highlights one set of relations between program parts but obscures others. The analyses are not intended to be claims about mental representations, rather these abstractions are based on formal analyses of programs developed by computer scientists. Analyses of natural language text in terms of underlying causal, referential, or logical relations are similar abstractions of text based on different kinds of information in the text that are relevant to its comprehension (Kintsch, 1974; Meyer, 1975; Trabasso, Secco, & van den Broeck, 1982).

The program text to be analyzed is written in COBOL, a programming language noted for its resemblance to English (see Fig. 1A). This program solves a toy problem in which a list of clients and their product orders for a month are processed and average order sizes for two subsets of clients are computed (see Fig. 1B).

The first abstraction of the program text is structured in terms of the goals of the program, that is, what the program is supposed to accomplish or produce (see Fig. 2). Shown as a goal hierarchy, it is a decomposition according to the major program functions or outputs (cf. Adelson, 1984). The higher level decompositions show that the program will produce three things: two averages and some printed output. At the lower levels, subgoals are specified for each higher level goal. For example, computing the average for the subset of "ordering" clients involves summing over orders, counting the relevant subset of clients and dividing. Notice that in this *function* abstraction there is little explicit information as to how these goals will be accomplished. For example, the total list of clients could be

A. COBOL PROGRAM SEGMENT

```
STATEMENT NUMBER
     1          MOVE ZERO TO COUNT-CLIENTS.
     2          MOVE ZERO TO TOTAL-ORDERS.
     3          MOVE ZERO TO INACTIVE-CLIENTS.
     4          READ ORDER-FILE INTO ORDER-REC.
     5          PERFORM SUM-ORDERS UNTIL ORDER-REC-ID = 999999.
     6          COMPUTE ACTIVE-CLIENTS = COUNT-CLIENTS - INACTIVE-CLIENTS.
     7          COMPUTE CLIENT-AVG = TOTAL-ORDERS/COUNT-CLIENTS.
     8          COMPUTE ACTIVE-AVG = TOTAL-ORDERS/ACTIVE-CLIENTS.
     9          DISPLAY MSG-1, ACTIVE-AVG UPON PRINTER.
    10          GO TO ORDER-EXIT.
    11     SUM-ORDERS.
    12          ADD 1 TO COUNT-CLIENTS.
    13          ADD ORDER-REC-QUANT TO TOTAL-ORDERS.
    14          IF ORDER-REC-QUANT = ZERO ADD 1 TO INACTIVE-CLIENTS.
    15          READ ORDER-FILE INTO ORDER-REC.
```

B. PROBLEM: GIVEN A LIST OF CLIENTS AND THEIR PRODUCT ORDERS FOR THIS MONTH, CALCULATE THE AVERAGE QUANTITY ORDERED FOR ALL CLIENTS AND THE AVERAGE FOR CLIENTS WHO ORDERED DURING THE MONTH. PRINT OUT THE AVERAGE ORDER SIZE FOR ORDERING CLIENTS.

FIG. 1. A sample program text (A) and the problem it solves (B).

searched once to count up the active clients and once again to add up the order quantities. Alternatively, a single pass through the list could classify the client as active or not and perform the appropriate count and sum operations when an active client is encountered. Of course, the implementation details are in the text but are lost in the abstraction focusing on functional relations between parts. Some inferences about the ordering of events can be made from this representation on the basis of everyday knowledge; for example, the orders must be summed before division can take place.

A second abstraction of the program text is structured in terms of program processes that transform the initial data objects into the outputs of the program (see Fig. 3). For example, Fig. 3 shows that the data object "file of client orders" is used by the process "count" to calculate the number of clients without orders this month. Because the flow of each data object can be traced through the series of transformations in which it participates, this is called a *data flow* abstraction. This abstraction is closely related to the function abstraction shown in Fig. 2. For example, the first-level decomposition of goals in the goal hierarchy is to compute averages and print an average. These correspond to the final data objects

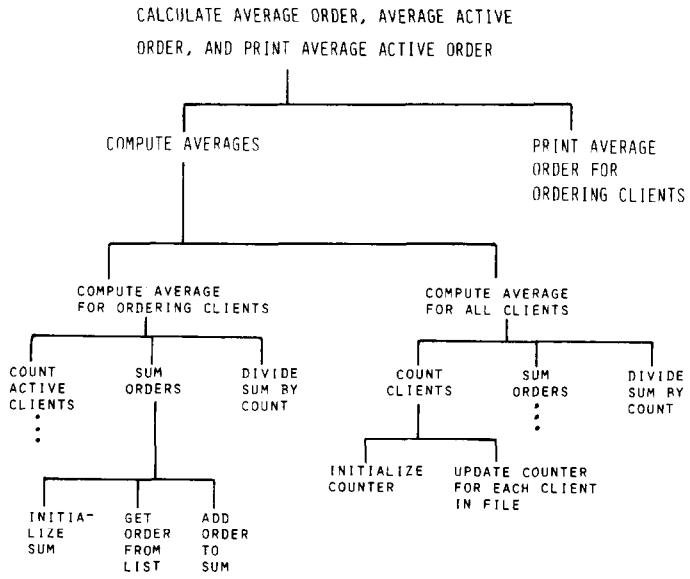

FIG. 2. Abstraction of function: The program accomplishes certain goals by producing outputs. Each level indicates a higher order goal that is decomposed into subgoals.

at the bottom of the data flow abstraction shown in Fig. 3, which are a printed average and a computed average. The goal hierarchy can be at least partly recovered from the data flow abstraction by working up from the bottom, although it requires the application of knowledge to infer the grouping of subgoals with their goals. However, in the data flow abstraction, everything that happens to a particular data object is readily available in a way that is not apparent from the goal hierarchy. In addition, the data flow abstraction allows more inferences to be made about the order in which certain operations will occur than does the goal hierarchy. If an action (marked by a box, e.g., "compute") has two data objects as inputs (marked by an oval, e.g., "sum of orders", "number of clients"), then the action cannot take place until the data objects are both available; thus the process that produce a data object (e.g., "sum orders") must execute prior to the process that consumes it ("compute average").

A third abstraction of the program text, called a *control flow* representation or flowchart, is structured in terms of the sequence in which program actions will occur (see Fig. 4). The links between program actions in this structure represent the passage of execution control instead of the passage of data as in the data flow abstraction. This form highlights sequencing information but conclusions about data flow must be inferred by looking for repeated data object names. For example, to find out in what events the "counter for clients" participates (easily determined in

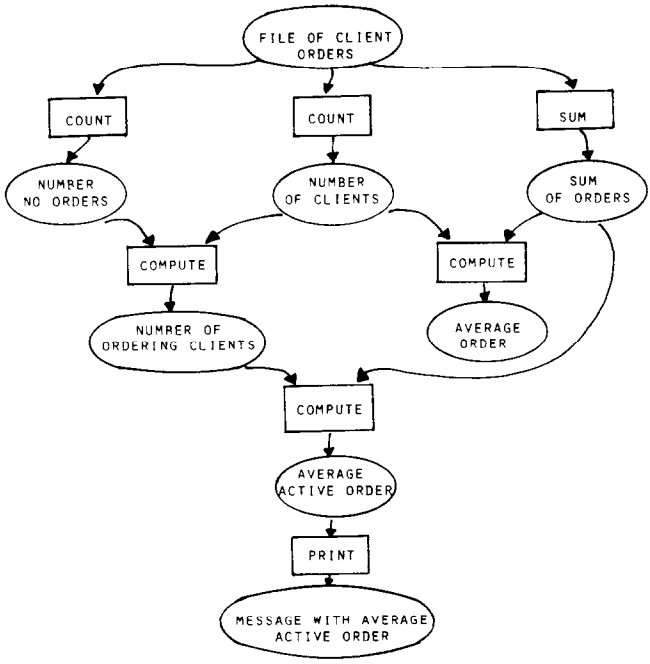

FIG. 3. Abstraction of data flow: Program actions transform initial data objects into final data objects. ◯ = data objects; ▭ = program actions.

the data flow abstraction, Fig. 3) it is necessary to track its use in the sequence of operations in the control flow abstraction (Fig. 4). It is also difficult to detect goal/subgoal relations quickly. For example, the higher order goal of computing an average over ordering clients is specified in the last procedural block of the control flow abstraction, but the subgoal operations of summing and counting are not explicitly linked to the higher order goal.

A fourth abstraction is structured in terms of the program actions that will result when a particular set of conditions is true (see Fig. 5). This abstraction is like a decision table in which each possible state of the world is associated with its consequences; it also resembles the production system condition-action pairs that are used to represent human procedural knowledge (e.g., Anderson, 1983; Newell & Simon, 1972). In this abstraction, the program is viewed as being in a particular state at each moment in time. The state triggers an action, execution of the action results in a new state, the new state triggers another action, and so on. It is therefore easy to find out what results if a given set of conditions occurs, and also relatively easy to find out what set(s) of conditions can lead to a given action. This kind of state information is much harder to deduce

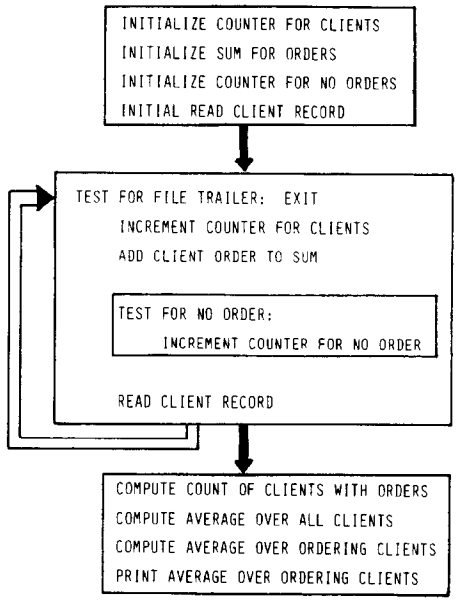

FIG. 4. Abstraction of control flow: Program actions occur in a particular sequence.

from the other abstractions. However, information about the sequence in which actions occur and information about higher level goals are difficult to extract in the *conditionalized action* abstraction.

This analysis of the multiple abstractions that characterize a computer program text also applies to English language instructions, such as training manuals, recipes, knitting instructions, and assembly instructions. In these texts, too, information is conveyed about what should be accomplished (goal hierarchy), how to do it (sequential procedure), the sets of conditions under which particular actions should be taken (conditionalized action), and the set of transformations that a particular object should go through (data flow). To be concrete, consider a hypothetical set of detailed instructions on how to cook spaghetti carbonara. An abstraction of the instructions in terms of function tells "what is to be accomplished." For example, the top level goal might be specified as "make spaghetti carbonara," with immediate subgoals of "make spaghetti, mix sauce, mix together." A procedural (control flow) abstraction specifies the order of execution and tells "how to do it." For example, "assemble the ingredients, heat the water, start the noodles, grate the cheese, check the noodles" might constitute a partial description of the sequence of steps. "What happens to particular objects" is portrayed by a data flow abstraction that traces the series of transformations applied to each object. For example, the cheese comes out of the refrigerator, is grated,

ACTIONS / CONDITIONS (STATE OF THE WORLD)	INITIALIZE COUNTERS	INITIALIZE SUM	READ CLIENT FILE	TEST FOR FILE TRAILER	INCREMENT COUNTER FOR CLIENTS	ADD TO SUM OF ORDERS	TEST FOR NO ORDER	INCREMENT COUNTER FOR NO ORDER	COMPUTE CLIENTS WITH ORDERS	COMPUTE AVERAGE OVER ALL CLIENTS	COMPUTE AVERAGE OVER ACTIVE CLIENTS	PRINT	STOP
START	X	X	X										
NEW RECORD				X									
HAVE TRAILER									X	X	X	X	X
NOT TRAILER					X	X	X						
& NO ORDER			X					X					
& ORDER			X										

FIG. 5. Abstraction of conditionalized action: A set of conditions results in the execution of some action(s). The execution of an action results in a new set of conditions.

separated in half, with one half going to the table and one-half going into an egg mixture. A condition-action abstraction specifies the conditions that could trigger certain actions such as when it is time to heat the bowl.

Comprehension requires the detection and representation of these multiple relations between parts of the text. From our illustrations it is clear that these relations are difficult to express simultaneously. The importance of this is that part of the difficulty of writing clear instructions, understanding instructions, or understanding programs is due to the trade-offs that inevitably occur in how much of each kind of information can be highlighted simultaneously. Uncertainty about the best way to write instructions or programs may be largely due to uncertainty about which structure should serve as the organizing principle for the instructions.

It is possible that one of the alternate text abstractions corresponds more closely than do the others to the structure of the programmer's mental representation of the program, due to features of the cognitive processing system and the organization of knowledge used in the comprehension process. For example, Adelson (1984) suggests that a mental representation in terms of program goals, reflecting text relations specified in Fig. 2, characterizes the natural cognitive representation of experienced programmers while a procedural representation, reflecting text re-

lations specified in Fig. 4, is most natural for novice programmers. It is also possible that the actual mental representation used by the programmer will reflect task and programming language influences in addition to the influences of cognitive capacities and knowledge structures. For example, different programming languages highlight different relations outlined in Figs. 2–5 (Green, 1980; Green et al., 1980). We assume that one mediating factor in correspondences between text structure and the structure of mental representations is the structure of programming knowledge activated during comprehension. We now examine how the structure of human programming knowledge is related to these text abstractions and to potential forms of mental representations of programs.

PROGRAMMING KNOWLEDGE

Various types of knowledge about programming will enable the programmer to detect and mentally represent the variety of relations that are implicit in the text. Some kinds of knowledge will be more important than others in constructing the macrostructure of the mental representation. In the present research we explore the role of two types of programming knowledge in program comprehension: knowledge of text structure and knowledge of program plans. These two kinds of knowledge do not exhaust the potential range of programming knowledge but they provide a useful starting point and they have been most frequently promoted as the cognitively natural bases for program design.

Text Structure Knowledge

Program text can be described in terms of a limited number of control flow constructs. Although there are differences about the exact number and description of these control flow constructs, three basic building blocks are typically included: *sequence* in which control passes from one action to the next; *iteration* in which an action is repeated until a specified condition exists (commonly referred to as looping); and *conditional* in which control passes to different actions depending on which of two or more conditions is met (sometimes referred to as if-then-else.)[1] These units could be called structured programming units because of an emphasis on disciplined control structuring according to these constructs by early structured programming advocates (e.g., Dahl, Dijkstra, & Hoare, 1972). These fundamental units have also been called prime programs referring to the idea that a program text can be decomposed into sequence, iteration, and conditional units in the way that a number can be

[1] Contention over the number of conceptual units concerns whether variations on looping structures should be recognized as separate constructs. These controversies do not affect the discussion.

decomposed into prime number factors. Prime programs at the lowest level of decomposition, represented as a single node, can be aggregated into higher level sequence, iteration, and conditional units (Linger, Mills, & Witt, 1979; Basili & Mills, 1982) so that the entire program text can be represented as a hierarchy of prime units. An analysis of the sample program text (Fig. 1A) in terms of these prime program text structure (TS) units is shown in Fig. 6A (cf. Curtis et al., 1984). For example, statements numbered 1–4 in Fig. 1A form a sequence unit as shown in Fig. 6A; Statements 5, 11, and the embedded sequence 12–15 form an iteration unit (loop); and statements numbered 6–10 form another sequence unit. A concatenation of the sequence, loop, and sequence units yields a higher level sequence unit that is the entire program text. Thus the text is structured as a hierarchy of prime program units.

The decomposition of a program into control primes and the diagramming of control flow according to structured programming units are analytic techniques that can be applied to programs. We refer to programmers' knowledge about these structured programming units as *text structure knowledge* (TS knowledge). Professional programmers could not easily escape exposure to this knowledge in the course of their programming education. One role that text structure knowledge could play in comprehension is that of organizing the memory representation macrostructure, and some researchers have claimed that knowledge of these structural components plays a central psychological role in program comprehension. For example, advocates of structured programming have hypothesized that programs organized according to a strict control construct discipline are easier to understand and modify because they correspond to the programmer's mental organization (Dahl et al., 1972); that the process of comprehending undocumented programs is similar to decomposing a program into prime programs (Linger et al., 1979).

One presentation of such a comprehension scheme proposes that the mental representation of a program has a macrostructure organized by control primes, that is, the sequence, iteration, and conditional text structure units (Atwood & Ramsey, 1978; Curtis et al., 1984; Basili & Mills, 1982). In this view, program comprehension proceeds by identifying sequence, iteration, and conditional units in the surface structure of the program and deriving their local purposes. These units then act as items that combine into higher order sequence, iteration, and conditional units, with higher level functions attached to units at this level. This process continues until the highest level is a single unit with an identifiable function. For example, the text structure (TS) decomposition of the sample program segment shown in Fig. 6A shows "initialization sequence," "read loop," and "computation sequence" as the first-level macrostructure control primes and higher levels of macrostructure are

created by their combination (see also Atwood & Jeffries, 1980; Davis, 1984; Mayer, 1977; Shneiderman, 1980). Thus, the text structure analysis represents a hypothesis about relations between program parts that organize the semantic representation of text in memory.

The decomposition of program text into text structure (TS) units (Fig. 6A) is most closely related to our earlier analysis of program text in terms of control flow relations (Fig. 4). However, such a decomposition will not necessarily correspond to the surface ordering of events in the program text as shown by the sequence of program text statement numbers in Fig. 6A.

There is some empirical support for the idea that program text structure knowledge plays an important role in comprehension and possibly an organizing role in memory. One line of support is provided by evidence that programs have a psychological "phrase structure" in which the phrases are syntactically marked by keywords of the programming language: WHILE . . . DO is a marker that initiates loops; BEGIN goes with END to mark a sequence, and so forth (McKeithen, Reitman, Reuter, & Hirtle, 1981; Norcio & Kerst, 1983). In one study testing free recall memory for program texts (McKeithen et al., 1981), statements recalled most frequently by experts but not by novices corresponded to statements marking the phrase structure. In another study (Norcio & Kerst, 1983) higher proportions of correct to incorrect recall transitions (and vice versa) occurred at the boundaries of the hypothesized text structure units, analogous to findings in sentence comprehension research that recall errors are greater over phrase structure boundary transitions (Fodor, Bever, & Garrett, 1974; Mitchell & Green, 1978; Tejirian, 1968).

Additional support is provided in a study of program comprehension in which programmers studied short programs composed of meaningful code, structured but meaningless code, or randomly arranged lines of code (Schmidt, 1983). More lines of both meaningful and structured but meaningless code were recalled compared to randomly arranged code. Further, longer study times occurred at control construct borders, in the same way that reading times are elevated at episode boundaries in story comprehension (Haberlandt, 1980; Mandler & Goodman, 1982).

Additional indirect support was obtained in a study (Adelson, 1981) of subjective organization in programmers' (multitrial) free recall of randomly presented lines of program code. The randomly presented lines contained statements that could be viewed as three routines (five lines each) or as five syntactic groupings (three lines each). Experts used program membership as an organizing principle and routines were grouped at a second level by procedural similarity, and not by the function of the routine.

In summary, the idea that text structure units play an organizing role in memory suggests three main features of program comprehension: (1) Comprehension proceeds by segmenting statements at the detail level into phraselike groupings that then combine into higher order groupings. (2) Syntactic markings provide surface clues to the boundaries of these segments. (3) The segmentation reflects the control structure of the program. Thus, in terms of the multiple abstractions of programs (Figs. 2–5), sequence information should be readily available; data flow connections that occur across unit boundaries should be relatively more difficult to infer; and function information should be least accessible, since it is most closely related to data flow and requires coordination across units.

Plan Knowledge

A second kind of knowledge, called program *plan knowledge* (PK knowledge), emphasizes programmers' understanding that patterns of program instructions "go together" to accomplish certain functions (Rich, 1981; Soloway, Ehrlich & Black, 1983; Soloway, Ehrlich & Bonar, 1982). Plans correspond to a vocabulary of intermediate level programming concepts such as searching, summing, hashing, counting, etc., and there are hundreds (maybe thousands) of these plans. Like other forms of engineering and design, "there is a craft discipline among programmers consisting of a repertoire of standard methods of achieving certain types of goals" (Rich, 1980).

A plan is a structure with roles for data objects, operations, tests, or other plans, and with constraints on what can fill the roles in a given instantiation as well as specifications as to data flow and control flow connecting segments within plans. Plans accomplish things and are hierarchically linked on the basis of function and role relations; one plan may be used to accomplish the goals of a higher order plan (Soloway et al., 1983). For example, a very simple plan is a counter plan that consists of an initialization part plus an update-by-one part. A plan to compute an average will include a counter plan as one of its parts. Higher level plans include such things as a find-first-value-search plan, a merge-two-files plan, or a bubble-sort plan.

The specification of plan knowledge in programming has been elaborated by Rich, Shrobe, and Waters (Rich, 1980, 1981; Rich & Shrobe, 1979; Rich & Waters, 1981; Shrobe, 1979; Waters, 1979), who have developed a large set of plans based on their intuitions about programming, and by Soloway and Ehrlich (Soloway & Ehrlich, 1984; Soloway et al., 1982, 1983) who have developed a more psychologically motivated theory of plan knowledge. A plan knowledge (PK) decomposition of the COBOL

program segment (Fig. 1A) in terms of underlying program plans is shown in Fig. 6B (this particular decomposition is based on work by the MIT Programmer's Apprentice project; Rich & Shrobe, 1979; see Soloway et al., 1983, for a similar analysis). As before, numbers in Fig. 6B refer to program statement numbers specified in Fig. 1A, and the hierarchical structure of the diagram in Fig. 6B shows that plans combine together to form higher order plans. For example, Statements 2 and 13 implement a counter plan; the segment as a whole consists of plans for reading through the inputs, counting, summing, and computing an average.

Plan knowledge units could also form the comprehension macrostructure, implying that understanding a program is finding a set of underlying plans such that parts of the program match the roles in the hypothesized plans. Comprehension of a program, under this view, would proceed by partial pattern matches activating candidate plans, causing programmers to search for further evidence to instantiate a plan. According to this concept of comprehension the program is mentally represented as a set of linked descriptions, like blueprints, rather than as a set of instructions to be executed. Thus, the plan knowledge analysis also represents a psychological hypothesis about relations between program parts that might organize the semantic representation of text in memory as depicted in Figure 6B.

Plan representations of a program are primarily based on data flow relations. This is because much of the control structure in a program that is not mandated by data flow requirements is arbitrary. Thus the plan knowledge (PK) analysis (Fig. 6B) is most closely related to our earlier analysis of program text in terms of data flow relations (Fig. 3) and func-

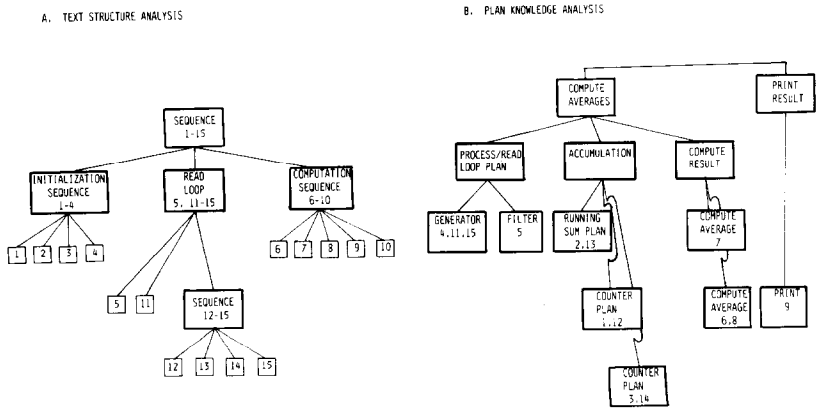

FIG. 6. Example analyses of the program segment shown in Fig. 1A. (Numbers refer to statement numbers in Fig. 1A.)

tion (Fig. 2). Such a decomposition also does not necessarily correspond to the text surface structure as shown by program text statement numbers in Fig. 6B.

There is also empirical evidence concerning the importance of plan knowledge in program comprehension. PK representations have been invoked to explain how expert programmers chunk program text in recall tasks (Greeno & Simon, 1984) by arguing that plan knowledge is used to code the functions of the presented program. Details of the program need not be encoded, because the programmer has only to expand a plan into one of its implementations to reconstruct the detail. This claim corresponds to claims made in research on natural language processing that encoding efficiency is achieved by activating scripts (Schank & Abelson, 1977) or other kinds of content schemas such as goal/plan knowledge about human actions (Schank & Abelson, 1977; Wilensky, 1983).

There is evidence that data flow relations are important in algorithm design (Kant & Newell, 1984) and program modification (Weiser, 1982), and evidence that experienced programmers are better than novices at inferring program function in a comprehension task (Adelson, 1984). These studies do not address questions about how function and data flow relations are inferred (i.e., by recognizing plans or in some other way), but they do indicate the central role of program function in experts' understanding.

Evidence more directly related to plans as critical elements in program comprehension is provided by Soloway and his colleagues (Soloway & Ehrlich, 1984; Soloway et al., 1982) using a cloze procedure to show that programmers will fill in a missing line of a program with a predicted plan element; that programmers have more difficulty comprehending a program in which the plan structure has been disrupted; and that experts but not novices can resolve dilemmas when conflicting cues about which plan to instantiate are provided. In addition, Brooks (1975) has simulated program composition by specifying a large set of program plans and processes that operate on them.

In summary, the idea that plan knowledge plays an organizing role in memory suggests the following features of program comprehension: (1) Comprehension proceeds by the recognition of patterns that implement known programming plans. (2) Plans are activated by partial pattern matches, and confirming details are either sought or assumed. (3) The resulting segmentation reflects the data flow structure of the program indexed by program function. Thus, in terms of the multiple abstractions of programs (Figs. 2–5), data flow and function information should be readily available; sequence and detail operations should be less accessible.

RESEARCH OVERVIEW

A summary of the correspondences we have proposed between textual relations (abstractions of program text), knowledge structures, and hypothesized mental representations is shown in Table 1. Features of the text activate different kinds of knowledge, some of which will provide an organizing structure for the mental representation of the text. The rows of Table 1 represent alternative hypotheses concerning the dominant form of the mental representation of programs. The structures illustrated in Figs. 6A and 6B show in detail the potential alternative meaning structures in memory corresponding to the TS and PK analyses of one text segment and we have outlined a view of comprehension that might lead to each.

There are several reasons to be interested in which of these views better characterizes computer program comprehension. First, well-known empirical results across a wide range of problem-solving domains, such as chess (e.g., Chase & Simon, 1973a, 1973b), GO (Reitman, 1976), bridge (Engle & Bukstel, 1978), music composition (Halpern & Bower, 1982; Sloboda, 1976), and computer programming (McKeithen et al., 1981; Shneiderman, 1976) show that experts quickly identify meaningful patterns in a problem array, that are stored in memory as chunks of information. These results suggest that for experts an abstract representation of a problem array is available quite quickly upon inspection. Much less is known about exactly which principles underlie experts' superior organization of problem information. The nature of mental representations of programs and the units that underlie their organization (e.g., Adelson, 1984; Curtis et al., 1984; Davis, 1984) are important for resolving arguments over how programs ought to be structured, understanding the psychological complexity of programs, and extending insight into skilled performance to an important complex task. Second, the two modes of comprehension have different consequences in terms of the kinds of information that are relatively easy or difficult to abstract from program text (Green, 1980). This in turn is important in determining standards for computer programming practices, tools, languages, and education.

More broadly, these two views of program comprehension mirror de-

TABLE 1
Correspondences between Text Abstractions, Knowledge Structures, and Mental Representations

Text relations	Knowledge structures	Mental representation
Control flow	Text structure	Procedural episodes
Function and data flow	Plan knowledge	Functional representation
Condition-action	Unknown	Unknown

bates in other areas of text comprehension and composition concerning the ways in which different kinds of knowledge contribute to text understanding. One kind of knowledge that has been proposed to influence comprehension is abstract knowledge of text structure. For example, content free abstract knowledge about the type and form of components usually included in a story (setting, episodes consisting of identifiable parts such as initiating events, goals, attempts, and consequences) is often hypothesized to provide organizing memory structures in story comprehension (e.g., Mandler, 1984; Rumelhart, 1975). A second kind of knowledge that has been proposed to influence comprehension is schematic content knowledge. For example, work on "scripts, goals, and plans" provides evidence that content specific knowledge about typical human action sequences in specific contexts and knowledge of typical plans that achieve certain goals provide organizing memory structures in story comprehension (Black & Bower, 1980; Bruce, 1980; Schank & Abelson, 1977; Thorndyke & Yekovich, 1980). Arguments about the priority of one or the other type of knowledge in story comprehension are difficult to resolve, since both story texts and the plans involved in human action sequences tend to have the same or similar structures (Black & Wilensky, 1979; van Dijk & Kintsch, 1983). In programming using more traditional languages, text structure knowledge corresponds to structured programming or prime program units; the units are few in number and abstract, a kind of "episode" for programs. Plan knowledge corresponds to schematic content knowledge and there are potentially thousands of such patterns.

The empirical evidence cited in the previous section concerning each view of computer program comprehension, in terms of text structure units (TS) or plan knowledge units (PK) is not definitive with respect to the role of the two kinds of knowledge in forming memory macrostructures. For example, superior recall of program statements introducing loops could reflect the priority of iteration control flow units or attention to key statements that active plan knowledge (McKeithen et al., 1981). Similarly, evidence that experienced programmers have tacit knowledge regarding awkward program constructions does not necessarily imply that this knowledge leads to plan based mental representations (Soloway et al., 1983). The research reported in the next sections was designed to operationally identify the form of mental representations of program texts, providing information about the kinds of relational information in programs that are most accessible and about the roles of text structure knowledge and plan knowledge in program comprehension.

In the first study programmers studied short program texts and responded to comprehension and memory questions. Short texts were used to obtain a high degree of experimental control. Although programming

studies have typically used texts of this length, it is desirable to examine experimental results in more realistic settings. In the second study programmers engaged in a more natural task in which they studied a program of moderate length, made a modification to it, and responded to comprehension questions. Thus the first study provides relatively direct information concerning the form of mental representations of program text. In the second study, comprehension data provide indirect evidence concerning the same questions for a different, more natural task.

STUDY 1

One effective technique for empirically investigating structures in memory is to index the relative distance between elements in a hypothesized structure by measuring priming effects in item recognition (McKoon & Ratcliff, 1980, 1984; Ratcliff & McKoon, 1978). In this method, subjects study one or more texts and are subsequently presented with a recognition test in which they must decide whether or not each item in the list was in the text they had just studied. A target item in the test list is preceded in one condition by another item hypothesized to be in the same cognitive unit as the target and thus close in the memory structure. In a second condition the target item is preceded by an item hypothesized to be in a different cognitive unit and thus further away in the memory structure. Under the assumption that activation of an item in the memory structure activates items close to it, especially those in the same cognitive unit, response time to the target preceded by an item in the same cognitive unit should be faster than response time to the same target preceded by an item not in the same cognitive unit (Anderson, 1983; McKoon & Ratcliff, 1980); that is, a priming effect should occur.

In the first experiment the priming technique was used to examine distances between program statements in program texts like the one shown in Fig. 1A. If the representations in memory of the meanings of the program segment correspond to the structures built by the TS (text structure) or PK (plan knowledge) decompositions, then the relative amounts of priming between the items should be predicted by the relative distances between the concepts in the diagrammed structures. For example, in Fig. 6A, the TS structure, Statement 2 (MOVE 0 TO TOTAL-ORDERS.) should prime Statement 4 (READ ORDER-FILE INTO ORDER-REC.) because they are in the same TS cognitive unit. However, Statement 2 (MOVE 0 TO TOTAL-ORDERS.) should not prime Statement 13 (ADD ORDER-REC-QUANT TO TOTAL-ORDERS.) as much because they are not in the same TS cognitive unit. The PK structure (Fig. 6B) makes the opposite prediction: statement 2 (MOVE 0 TO TOTAL-ORDERS.) should prime Statement 13 (ADD ORDER-REC-QUANT TO TOTAL-ORDERS.) because they are in the same PK cogni-

tive unit and Statement 2 (MOVE 0 TO TOTAL-ORDERS.) should not prime Statement 4 (READ ORDER-FILE INTO ORDER-REC.) as much because they are not in the same PK cognitive unit. Other covarying features such as argument repetition and surface distance will need to be controlled by balancing these attributes across the set of items used (McKoon & Ratcliff, 1984).

It is possible that neither of the theoretical decompositions shown in Figure 6 precisely describes the programmer's decomposition. In this case priming effects will not be obtained. However, failure to find priming effects is not informative as to what is wrong with the theoretical proposals and additional measures of program comprehension are needed. One additional measure is to ask programmers questions about their understanding of the program text in order to ascertain what aspects of meaning can be attained in limited time and to provide an assessment of learning relevant to interpretation of recognition memory test results. Earlier we suggested that there are at least four kinds of relations between program statements that contribute to a complete understanding of the program: major functional relation specifying the goal structure of the program (Fig. 2); data flow relations specifying the sets of events in the program in which particular variables participate (Fig. 3); control flow relations specifying the execution sequence of statements (Fig. 4); and state relations specifying the sets of conditions and resulting actions in the program (Fig. 5). A fifth kind of information in the program consists of the detailed operations themselves, the actions corresponding to a single statement or less.[2]

The two general approaches to program comprehension (TS, PK) differ in terms of the kinds of relations between program elements that are hypothesized to be central in mental representations. Therefore, the two approaches lead to differing predictions about the kind of information that will be directly available to the programmer from the representation, or easier to infer from the mental representation. The TS view suggests that when a programmer studies a program, the meaning is built up from the bottom in terms of the operations binding together into the control flow units that are assigned local purpose. Major function and goal information is available only after these relations have been built. Thus the TS view stresses detailed operations, control flow, and then function in the representational hierarchy, suggesting that questions about detailed operations and execution sequence will be answered more easily (faster and with fewer errors) than will questions about major function and data flow. The PK view suggests that when a programmer studies a program, func-

[2] It can be less, since the programmer may know that a single command in the programming language executes one or more actions.

tion is inferred immediately when a programmer identifies a familiar stereotypic unit and the operations and data objects will be bound to the role slots in the hypothesized plan. The PK view stresses data flow and functional dependencies as central in the representations, suggesting that function, data flow, and detailed operation information should be available in that order. Neither view predicts that state information will be easy to extract from program text, although some languages and applications (AI programming in LISP) emphasize these relations. Comprehension questions can be designed for a program text that asks specifically about these different kinds of relations in the program.

Methods

Subjects. Professional programmers with a minimum of 3 years of professional programming experience served as subjects in the research. Subjects were selected from a pool of over 400 programmers who volunteered to participate in response to mail solicitations to Data Manager Association members, radio and television announcements, Chicago newspaper stories, and approaches to several Chicago-area businesses and research institutions. Our choice of programming languages was constrained by the availability of experienced professionals for each language. Since 85% of the volunteers programmed primarily in COBOL, FORTRAN, and ASSEMBLER, subjects were drawn from the COBOL and FORTRAN programmers. This provides a basis for examining the generality of findings across the two languages most widely in use.

A total of 80 professional programmers participated in this study, 40 COBOL programmers and 40 FORTRAN programmers.[3] Differences between FORTRAN and COBOL programmers in educational level, college major, number of programming languages known, and number of years programming (but not number of years as a professional programmer) were statistically reliable ($p < .01$ level). The average FORTRAN programmer was 37 years old at the time of the study, male (95% of the sample), had majored in computer science or other science/engineering field, had completed some graduate level work beyond a bachelor's degree, knew six other programming languages, had taken four programming courses, had programmed for 14.5 years, had been a professional programmer for 10.8 years, and had spent an estimated 12,306 professional programming hours on program coding, debugging, and modification tasks. Forty-three percent had taught at least one programming course. The average COBOL programmer was 35 years old, male (77.5% of the sample), had a college degree, majored in social science or humanities, knew four other programming languages, had taken four programming courses, had programmed for 10.5 years, had been a professional programmer for 9.5 years, and had spent an estimated 11,196 professional programming hours on program coding, debugging, and modification tasks. Forty-five percent had taught at least one programming course.

Subjects were run over a period of 8 months from July 1983 to February 1984. Each programmer was paid $10 to cover transportation and parking costs.

Materials. The program segments developed for the research were drawn from four full-length programs currently in use in Chicago-area computer installations. The four programs spanned a range of program types: a batch file update program, an engineering application,

[3] Data from six additional subjects were discarded due to programmer difficulty with English (1), mechanical problems during the course of the experiment resulting in incomplete data (3), and motivational problems in completing the experiment (2).

an interactive program, and a computational program. Two of these programs were originally written in COBOL and two in FORTRAN. Eight program segments were taken from the programs and modified slightly so that they met the following criteria: (1) Each comprised exactly 15 lines; (2) each accomplished something sensible in isolation (i.e., was comprehensible as a fragment that did something concrete); (3) each contained TS and PK units that differed in content.

One of the program segments used in the experiment is shown in Fig. 1A. This particular segment is unusual because it is extremely simple. It was included in the research and is offered here as an example, because it has been used in many other studies of program comprehension and is frequently used as an example in published articles (e.g., Curtis et al., 1984; Soloway et al., 1983). Thus analyses in the present study can be compared directly to previous research. Figures 6A and 6B show the TS and PK theoretical analyses of the example program segment.

For each of the eight program segments, six comprehension questions were composed that varied according to the category of information about program relations to which each pertained. Examples of each kind of question for the program segment shown in Fig. 1A are: Will an average be computed? (function); Is the last record in ORDER-FILE counted in COUNT-CLIENTS? (sequence); Will the value of COUNT-CLIENTS affect the value of ACTIVE-AVG? (data flow); When ORDER-EXIT is reached, will ORDER-REC-ID have a particular known value? (state); Is TOTAL-ORDERS initialized to zero? (detailed operation).

Also for each of the eight segments, a recognition test list was constructed to examine priming effects between items. A critical target item was designated along with two primes (a TS prime and a PK prime) to form a triple to be used in test list construction. The essential feature of each triple was that the TS prime and the target were in the same cognitive unit according to the TS analysis of the segment but in different PK units and that the PK prime and the target were in the same cognitive unit according to the PK analysis of the segment but different TS units. For example, Statements 4, 13, and 2 (Fig. 1A) form a triple. Statement 4 is the TS prime because Statements 4 and 2 are in the same TS cognitive unit (Fig. 6A) but in different PK units (Fig. 6B). Statement 13 is the PK prime because Statements 13 and 2 are in the same PK cognitive unit (Figure 6B) but are in different TS units (Fig. 6A). Statement 2 is the target item since it appears in both pairs. For each program segment, four target items were identified along with their TS and PK primes, and the remaining three lines of code were designated as filler items. The targets were arbitrarily divided into two sets designated as A Materials and B Materials.

There are other bases besides roles in the TS and PK representations on which program statements might be associated in memory. For example, the surface distances between the prime and target statements differ for the example just given. In addition, some program statements have repeated arguments, others are very similar syntactically, and the direction (forward or backward) between the prime and target might differ within a triple. It was not possible to hold all of these other factors constant within any one triple or within any one program segment. However, the four potential influences—surface distance, argument repetition, syntactic similarity, and direction between prime and target—were balanced over all 32 (eight segments, four targets per segment) TS-target and PK-target pairs.

Prime and target items were embedded in a recognition test list consisting of 22 items, 7 false items and 15 true items. The 15 true items consisted of 3 "filler" true items and 6 prime-target pairs. The 6 targets consisted of 4 targets presented for the first time and 2 targets presented a second time. For the first-time targets, primes were paired with targets so that one group of subjects (within each language) saw PK primes immediately preceding A-targets in the recognition test list and TS primes immediately preceding B-targets. A second group of subjects (within each language) saw TS primes immediately preceding A-targets and PK primes immediately preceding B-targets. For the repeated targets, the prime

not seen before was paired with the target. False items used variable names that had occurred in the segment but were connected with an operation that had not connected them in the segment. False items did not consist of tricky misspellings or paraphrases of the program statements, since they were not intended to be lures. One of the 7 false items was a repeated item. Test lists were arranged in four different orders so that each prime-target pair appeared once in each quarter of the list, subject to the restrictions that a target item could not be placed in the first or second position of the test list and that primes had to immediately precede their targets. All program segments and test list items were prepared in two programming languages, FORTRAN and COBOL. COBOL subjects saw only COBOL segments and test items; FORTRAN subjects saw only FORTRAN segments and test items.

Procedure. Subjects participated in a single experimental session lasting approximately 2.5 h at the University of Chicago, Northwestern University, or their place of business. To begin the session an experimenter showed subjects the IBM Personal Computer to be used during the session and pointed out distinctive features of the keyboard. Thereafter, all instructions to the subjects were presented via the computer screen. The first part of the session consisted of general task instructions, detailed instructions concerning the use of editing features required during the task, and a practice trial.

Subjects were told that they would study a 15-line segment of code for a total of 4.5 min, divided into three 1.5-min intervals; that between the study intervals they would be asked to respond to comprehension questions and would be given a memory task. They were instructed that their primary task was to come to a complete understanding of the code so they could answer the comprehension questions accurately. We emphasized that their responses to the memory tasks should follow from their attempt to understand the code; that attempts should not be made to memorize the text or use special strategies for memorization. The exact task sequence was described and then demonstrated in the practice session.

The following task sequence was repeated three times for each program. Subjects studied a 15-line segment of code that appeared on the screen for exactly 1.5 min. Following instructions to prepare for comprehension questions, subjects responded to each question by pressing "yes" or "no." Response latencies and actual responses were recorded by the controlling program. The next screen announced the free recall section and subjects typed in as much of the 15-line segment as they could recall, in any order that it occurred to them. The third study-comprehension trial ended with a recognition memory test in place of the recall task. Recognition started with a screen reminding subjects to position their fingers correctly, to respond "yes" or "no" as quickly and accurately as possible, and not to pause during a list presentation. Subjects initiated the test with a keypress, with subsequent lines triggered by the previous response. The response and the response latency for each item were recorded.

The three study-test trials occurred for each of the eight program segments with a break between the fourth and fifth segments. At the conclusion of the session, subjects filled out a detailed background questionnaire and responded to questions posed by the experimenters about their reactions to the experiment and their own programming work.

These procedures were established by extensive pilot testing. For example, the total study time of 4.5 min was chosen to ensure high levels of recognition accuracy and moderately high level of segment comprehension. The comprehension questions were inserted before the recall and recognition tasks to focus subjects on the comprehension aspects of the task rather than on the memory requirements and to discourage inclinations to rehearse or retain a visual image of the text.

Design. The program segment and test list materials were used to form the basic research design: 2 (languages) × 4 (orders) × 2 (subject groups within language) × 2 (TS, PK prime types) × 2 (A, B sets of target items). Language, order, and subject group were between-subjects factors, and subject groups, prime type, and materials set formed a 2 × 2 × 2 repeated measures Latin square. In this design, comparisons between target response times

for different prime types is a within-subjects comparison but for different materials sets. A rearrangement of this design using first and second presentations of only those target items that were repeated in the test lists allows a within-subject comparison between identical target response times for different prime types. This comparison is of secondary interest because repetitions of true items in the test list are potentially confusing and may add variability to response times for these items.

This design provides tests of whether programmers' mental representations of program text reflect structural distances hypothesized by the TS analysis, the PK analysis, or neither analysis. Specifically, support for a TS macrostructure is obtained if response times to targets preceded by a TS prime are reliably faster than the same targets preceded by a PK prime. If this is the case, we can infer that the items specified by the TS analysis as forming a cognitive unit are in fact "closer" in memory than are the items specified by the PK analysis. Alternatively, support for a PK macrostructure is obtained if response times to targets preceded by a PK prime are reliably faster than the same targets preceded by a TS prime. Finally, if some response times to PK-primed targets are faster and other response times to TS-primed targets are faster, then no inferences may be drawn regarding which of the formulations more accurately portrays the nature of mental representations.

Response times and error rates for different kinds of comprehension questions provide an additional measure regarding relations that dominate in mental representations. Specifically, if support for a PK macrostructure is obtained with the recognition response times, then we expect to see fewer errors and faster response times for function and data flow comprehension questions. Alternatively, if support for a TS macrostructure is obtained with the recognition response times, then we expect to see fewer errors and faster response times for detailed operations and control flow comprehension questions.

Results

Recognition memory data. The question of primary interest is whether target response times are faster (1) when the target is immediately preceded by an item from an hypothesized TS cognitive unit or (2) when the target is immediately preceded by an item from an hypothesized PK cognitive unit or (3) there would be no difference between priming conditions. After removing extreme response times[4] and response times of incorrect items and items for which a prime error was made, a mean was computed for each subject for each of the two target sets (A, B). These means were analyzed in a $2 \times 2 \times 4 \times 2$ repeated measures analysis of variance with language (FORTRAN, COBOL), subject group (1, 2), and test list order (four orders) as between-subjects factors and type of prime

[4] A response time for a given subject and item was labeled extreme if it was more than 2.5 standard deviations from the subject's mean response time over correct responses, and if it was simultaneously greater than 2.5 standard deviations from the mean response time for that particular item computed over subjects in the same subject group. In addition, all response times greater than 10.0 s were considered extreme. About 1.9% of the response times were identified as extreme, and their removal lowered the average response time by about 150 ms and reduced variability. For example, the "cleaned" average response time for correct "yes" items was 2.512 s compared to a 2.670 uncleaned mean. All analyses were performed on cleaned and uncleaned data and in no case was the direction of differences between means altered by the removal of extreme response times.

(PK, TS) as a within-subject repeated measure. A second $2 \times 2 \times 2$ analysis of variance was performed on this data, treating materials as a random factor, with language (FORTRAN, COBOL) and materials set (A, B) as between-items factors, and prime type (PK, TS) as a within-item repeated measure. Examination of the cell means reveals that there are multiple influences on target response times (see Table 2).

As predicted by a text structure (TS) analysis of program comprehension, responses to TS-primed targets are on average 105 ms faster than responses to PK-primed targets, $F(1,64) = 4.51$, $p < .04$ (subjects analysis, see Table 2, Pt. A), $F(1,60) = 3.72$, $p < .06$ (items analysis). Considering only subjects whose comprehension scores were in the top quartile (since these subjects had a more complete understanding of the program segments), we see (Table 2, Pt. B) that the TS-primed speedup is larger, 237 ms, $F(1,15) = 8.35$, $p < .02$ (subjects analysis), $F(1,59) = 4360$, $p < .06$ (items analysis). Comparisons using the repeated target data show the same advantage for TS-primed targets, although the effect for repeated targets is statistically unreliable due to increased variance in repeat target times.

Target response times also differed for the (arbitrarily composed) A and B materials sets, for the two languages, and for the subject groups within each language. Responses to B materials took an average of 261 ms longer than responses to A materials, $F(1,64) = 28.18$, $p < .001$ (subjects analysis), $F(1,60) = 2.32$, $p < .14$ (items analysis), and this difference was larger for COBOL items than for FORTRAN items, $F(1,64) = 6.03$,

TABLE 2
Mean Response Times for Target Recognition Test Items as a Function of Prime Type

Subject group within language	FORTRAN subjects and materials		COBOL subjects and materials	
	PK prime	TS prime	PK prime	TS prime
A. All subjects, response time in seconds				
Subjects Group 1	2.691 (A materials)	2.695 (B materials)	2.526 (A materials)	2.834 (B materials)
Subjects Group 2	2.248 (B materials)	1.972 (A materials)	3.048 (B materials)	2.594 (A materials)
All subjects	2.470	2.333	2.787	2.714
B. Upper quartile (Q1) comprehension subjects, response time in seconds				
Q1 Subjects Group 1	2.560 (A materials)	2.463 (B materials)	2.667 (A materials)	2.948 (B materials)
Q1 Subjects Group 2	2.220 (B materials)	1.807 (A materials)	2.780 (B materials)	2.063 (A materials)
All Q1 subjects	2.391	2.135	2.724	2.505

$p < .02$ (subjects analysis, not significant for items analysis). COBOL subjects took longer to respond in general, $F(1,64) = 4.91$, $p < .04$ (subjects analysis), $F(1,60) = 4.88$, $p < .04$ (item analysis), and there was a subject group within-language difference, $F(1,64) = 5.29$, $p < .03$ (subjects analysis), $F(1,60) = 34.54$, $p < .001$ (items analysis). Subject group differences may not be attributed to experimental manipulations since response times for the 24 filler true items not involved in any experimental manipulation reveal identical differences between language and subject groups.

The array of effects can be seen more easily graphically (see Fig. 7). In Fig. 7A, response times are adjusted for the effect of subject groups within language, showing the TS priming effect for both languages, the effect for materials set, and the slight interaction between materials and language. In Fig. 7B, means are adjusted for the effect of materials sets, again showing the TS priming effect for both languages and the effect for subject group within language.

Overall recognition accuracy, measured by percentage correct, averaged 92.1%. FORTRAN subjects made fewer recognition errors (6%) compared to COBOL subjects (9.8%), $F(1,64) = 7.36$, $p < .01$, but recognition accuracy did not differ for subjects assigned to the two experimental conditions, nor were there any differences in recognition accuracy due to order of presentation of the program segments (Fs < 1). Accuracy for target recognition items across subjects and items averaged

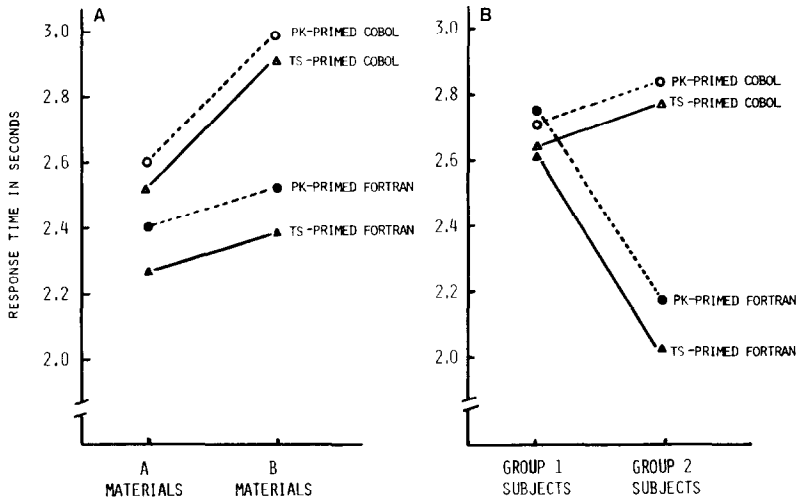

FIG. 7. Study 1 response times for recognition memory items comparing PK-primed item times to TS-primed item times for each set of materials within language adjusted for the effects of subject group (A) and for each subject group within language adjusted for the effects of materials set (B).

92.6% correct. Error rates differed by language (FORTRAN 5.8%, COBOL 9.1%; $F(1,64) = 4.19, p < .05$) but did not differ by experimental condition or by materials set (A-targets, B-targets) (Fs < 1). Correct responses averaged 2.670 s compared to incorrect responses, which averaged 3.667 s, $F(1,76) = 63.60, p < .001$, suggesting a difficulty relationship between speed and accuracy rather than a speed/accuracy trade-off. Thus, interpretation of the above results is not affected by error rates, and a constant error rate of 8–10% may be assumed.

It is not surprising that materials and subject differences account for a major portion of variability in time to recognize program statements. However, on top of these differences, program statements are consistently recognized faster when immediately preceded by a program statement in the same control flow unit (TS analysis). This result strongly supports the mental organization of program text proposed by a text structure analysis. On the basis of this result, we expect certain converging results in the programmers' responses to comprehension questions about these same program segments.

Comprehension data. Our main interest in comprehension accuracy is in differences that might occur between items in different information categories, that is, between questions asking about different kinds of information in program text. Referring to the earlier text analyses (Figs. 2–5), the 48 comprehension questions comprised 10 questions about detailed program operations (operations questions), 9 questions about program execution sequence (control flow questions), 9 questions about program data flow (data flow questions), 10 questions about program condition-action relations (state questions), and 10 questions about major program functions (function questions). We assume that higher error rates for questions in a particular information category imply that the information in that category is less easily accessed or computed from the memory representation. Under this assumption the text structure (TS) analysis (Fig. 6A) predicts that operations and control flow questions will be more easily answered and that data flow and major functions will be more difficult to infer. The plan knowledge (PK) analysis (Fig. 6B) predicts that data flow and major function information will be most accessible with operations and control flow less accessible.[5] State information is not predicted to be accessible under either formulation. The recognition memory results discussed above lead us to expect further support for

[5] To some extent these predictions are dependent on the time course of comprehension, so that more errors would be made earlier about less accessible information. Analyses of comprehension questions by presentation position were not informative due to the small number of items per cell at this level of analysis.

the TS formulation: Operations and control flow questions will be most easily answered.

Error rates were computed for each subject for items in each information category (percentage of items missed in the category) and for each item (percentage of subjects missing an item). The five scores per subject were submitted to a language (FORTRAN, COBOL) × information category (operations, control flow, data flow, state, function) repeated measures analysis of variance (with subjects as random factor) and the item scores were submitted to a language × information category analysis of variance (with items as random factor).

Information category of the comprehension question affected error rates, $F(4,312) = 26.75$, $p < .001$ (subjects analysis), $F(4,86) = 2.87$, $p < .03$ (items analysis). The ordering of difficulty of the information category questions was predicted by the text structure analysis: Questions about detailed operations and control flow relations were answered more accurately (15 and 21% errors, respectively), while more errors were made on data flow, state, and function items (28, 30, and 34% errors, respectively).

Overall comprehension levels for FORTRAN and COBOL programmers differed reliably when subjects were the unit of analysis, $F(1,78) = 6.01$, $p < .02$, although with items as the unit of analysis, variability among items swamped this difference, $F(1,86) = 1.10$. In addition the pattern of error rates for information categories differed for the two languages, $F(4,312) = 8.72$, $p < .001$ (subjects analysis), $F(4,86) = 0.827$ (not significant, items analysis). The comprehension pattern across information categories for FORTRAN subjects (see Fig. 8) was that questions about operations and control flow were answered most accurately, questions concerning major function next most accurately, followed by data flow and state questions. This yielded an inverted *U*-shaped pattern with operation, sequence, and function depressed, showing lower error rates (Fig. 8). The most noticeable differences between the FORTRAN and COBOL patterns was the elevated error rate on function questions for COBOL subjects, creating an increasing pattern across information categories. In addition to the elevation of major function question errors, COBOL data flow questions showed slightly higher accuracy.

The difference in patterns could be due to one or more of three factors. First, subjects from the two language groups were not equivalent in background characteristics. Second, the languages themselves could yield differences in ease of comprehension such that control flow and function information are easier to extract from FORTRAN than from COBOL text, and data flow information is easier to extract from COBOL than from FORTRAN. Third, the pattern of differences might be due to

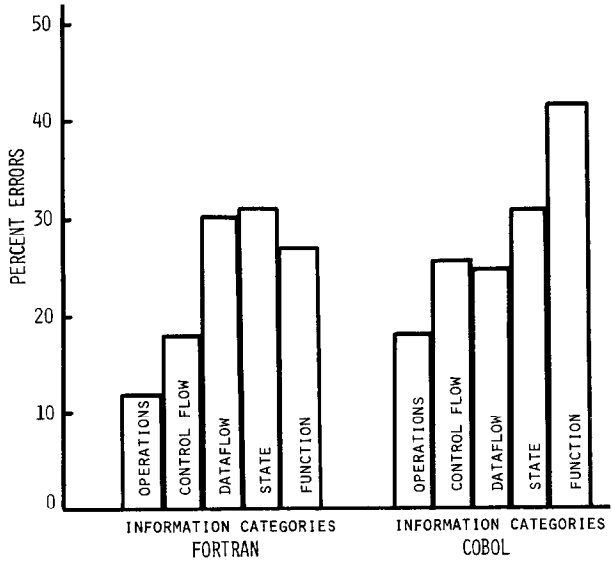

FIG. 8. Study 1 comprehension question error rates by information category for each language.

lower comprehension levels for the COBOL subjects and thus the COBOL pattern could reflect an earlier stage in the comprehension process and the FORTRAN pattern a later stage. Under the third interpretation, FORTRAN subjects, having understood more of the operation and control flow information, may have progressed further and could therefore either compute or retrieve function information from memory. COBOL subjects would not yet have reached this stage in the comprehension process after the allotted study time.

The first explanation, programmer background, is not supported because these variables are unrelated to comprehension performance on our task. For example, comprehension accuracy levels were equivalent for males and females, $F(1,78) = 1.32$; for different educational levels, $F(3,76) = 0.64$; and for different college majors, $F(3,71) = 1.16$. This is important because it suggests that any differences in performance between language groups will not be accounted for by sample differences in sex, education, and college major. There were also no differences in comprehension accuracy between programmers who had taught programming and those who had not, $F(1,78) = 0.96$.

In order to examine the other two explanations, subjects were divided into quartiles (within language) on the basis of their overall comprehension accuracy scores. If the overall FORTRAN pattern represents a later

stage of comprehension, then the upper quartile COBOL subjects should show a pattern more like the FORTRAN aggregate and the lower quartile FORTRAN subjects should show a pattern more like the aggregate COBOL subjects. Alternatively, if the differences in aggregate patterns reflect fundamental features of the language, then those differences will appear the same in the patterns for top and bottom quartile comprehension subjects. The comprehension question means across information categories for top and bottom quartile subjects are displayed graphically in Fig. 9.

Consistent with the stage of comprehension explanation, bottom quartile subject showed the COBOL aggregate pattern with elevated error rates for major function questions and top quartile subjects showed the inverted *U* pattern of the FORTRAN aggregate. A quartile × information category interaction, the statistical manifestation of this pattern, is only marginally reliable, $F(4,64) = 2.43$, $p < .06$ (subjects analysis), $F(4,86) = 2.02$, $p < .10$ (items analysis).

Some language specific features are also retained by the information category patterns, namely lower error rates for control flow questions for FORTRAN subjects (both quartile groups) and lower error rates for data

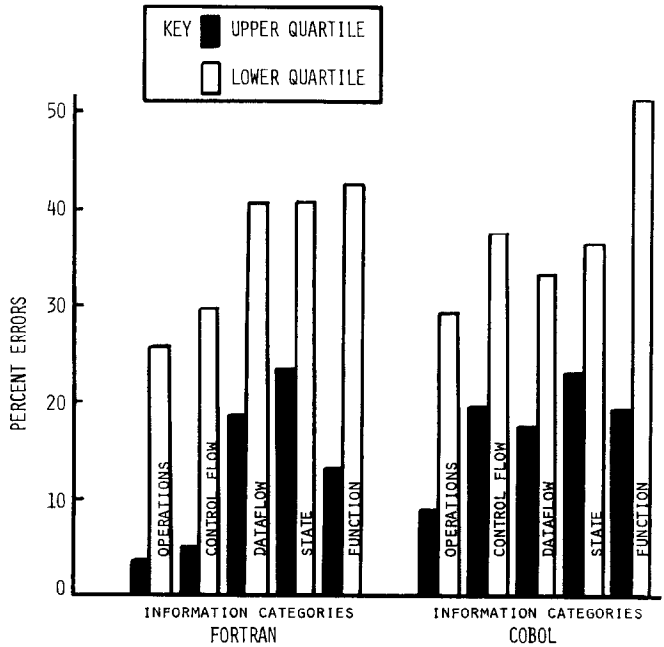

FIG. 9. Study 1 comprehension question error rates by information category for upper (dark bars) and lower (white bars) quartile subjects within each language.

flow questions for COBOL subjects (both quartile groups). This is consistent with an explanation attributing differences to language-specific features that affect the ease of extracting information from the text. However, as in the analysis of the complete subject sample, this interaction between language and information category shows statistical reliability only in the subjects analysis, $F(4,64) = 3.99$, $p < .006$, not in the items analysis, $F(4,86) = 0.619$.

In summary, after limited study time, experienced programmers' comprehension errors were strongly related to the kinds of inferences required to respond to the question, and to the language in which the programs were written (Fig. 8). Questions about program operations and control flow relations in the programs were answered correctly more often than questions about data flow relations and program states. Errors were made most frequently when inferences about program function were required. This general pattern supports the TS theoretical formulation (Fig. 6A). Differences between FORTRAN and COBOL programmers and between top and bottom quartile comprehenders suggest that inferences about program function (what it does) are the most difficult and appear late in the comprehension process, and that there are probably language-specific features that affect the ease of certain kinds of inferences (Fig. 9). FORTRAN programmers were consistently better on inferences about control flow while COBOL programmers were more accurate in responding to questions about program data flow relations.

In addition to data on comprehension errors, summarized above, we analyzed response times to comprehension questions, providing additional information about the relative difficulties of different kinds of inferences in comprehension. After correcting for reading time, we assume that a relatively faster response time to a question implies less processing has gone into constructing a response to the question. Faster responses could be due to direct retrieval of the information, to assessment of plausibility given retrieval of higher level information, or to fast computation of the response given retrieval of related information. Slower responses imply extensive search or burdensome computation from retrieved information (Reder, 1982; Glucksberg & McCloskey, 1981). For example, if major function information is stored directly and provides the macrostructure (Kintsch & van Dijk, 1978) of the text representation (Adelson, 1984; Atwood & Ramsey, 1978; Brooks, 1983; Shneiderman, 1980), then the fastest response times should occur for major function questions, with slower responses for more detailed comprehension questions like those in the operations category. However, if frequent errors correspond to the need to assemble responses at the time of questioning, then the error rate data above imply that responses to operations questions will be faster than responses to questions about program function.

Response times for comprehension questions about different information categories, standardized and adjusted for question length,[6] correspond to the pattern of comprehension *errors* for upper quartile subjects (Figure 9). Overall correct responses to true statements about the operations and control flow of the program were answered more quickly (mean residuals were $-.33$ and $-.16$, respectively) than questions about data flow and function ($-.05$, $-.11$), which were answered more quickly than questions about program states ($+.06$), $F(4,304) = 8.05$, $p < .001$ (subjects analysis), $F(4,86) = 2.85$, $p < .03$ (items analysis). Analyses of raw response times did not differ from analyses of residual response times due to relatively low correlations between question length and response time.

The response time data and the error data for the comprehension questions support the following conclusions: (1) Detailed operation information is stored directly, organized by control flow units (lowest percentage errors, fastest response times). (2) Control flow inferences are thus easy to retrieve or compute (low percentage errors, low moderate response times). (3) Some data flow and major function information is readily available (moderate response times) although when not stored, it is not easily computed (high percentage errors). (4) Program state inferences are difficult to compute (long response times, high percentage errors).

Discussion

The present results provide evidence that the dominant memory representation, formed during comprehension of short program texts in this experimental context, is organized by a small set of abstract program units related to the control structure of the program. More specifically, of the four program abstractions presented earlier (Figs. 2–5), relations captured by the procedural, control flow abstraction (Fig. 2) appear to be central in comprehension in our experimental task. Furthermore, the nature of the mental unitization of these relations corresponds to the basic program building blocks of sequence, iteration, and conditional identified by early advocates of structured programming.

Both recognition memory results and comprehension question results converge to support this conclusion. In the recognition memory test, recognition occurred faster when a statement was immediately preceded by

[6] Lengths of the comprehension questions varied (from 8 to 19 syllables, mean = 13.2 syllables) so response times were adjusted for reading time in order to compare response times between different information category question sets as follows: Response times were standardized for each subject; the response time predicted by a subject's syllables/response time correlation was subtracted out; the remainder is a standardized residual "due to thinking." No differences in results occur using other common methods of adjustment.

a statement in the same text structure unit than when it was immediately preceded by a statement that was not in the same text structure unit. This implies that statements in the same TS unit were closer together in programmers' memory structures. This priming effect cannot be accounted for by the text surface distance between the statements, by syntactic similarity between statements, or by argument repetition, since these features were controlled by counterbalancing test items.[7] Furthermore, responses to comprehension questions about control flow relations and program operations were answered faster and with fewer errors than were questions about data flow and function relations, supporting the idea that control flow and operation information is easier to access in memory.

Examination of the performance of programmers with the highest comprehension scores strengthens these conclusions and leads to a further speculation that segmentation on the basis of control flow relations occurs prior to comprehension of major program functions and data flow relations. First, the priming effect for TS cognitive units was strongest for top comprehenders. Second, top comprehenders' fast and error free responses to detail and control flow comprehension questions were accompanied by a disproportionate decrease in response errors to function questions. While one might argue on the basis of the error data alone that there is a priority for function information in top comprehenders' memory representations, the priming results and response time data undercut such a conclusion.

These empirical results fit a view of program comprehension in which the meaning of program text is developed largely from the bottom up. The text is first segmented according to simple control patterns segregating sequences, loops, and conditional patterns. At this level some specific inferences are made concerning the procedural roles of the segments. For example, a sequence at the beginning in which zeroes are assigned to variables may be designated "initialization of variables" (see Fig. 6A), without regard for the role of those variables in later computation. Another sequence may be designated as "something is calculated." A loop repeats whatever sequence is contained within it. A conditional pattern directs control to alternate sequences.

Data flow and function connections often require integration of operations across separate segments. For example, calculation of an average involves an initialization, a running sum, and final calculation. As in Fig.

[7] This does not preclude priming due to some other unknown form of relatedness that coincidently was confounded with TS unit membership. However, this "other" basis would also have to account for converging comprehension question results and for results obtained in the second study.

6A, these occur in three separate procedural units. The results suggest that these connections are made later in comprehension, and for programmers with the lowest comprehension scores they are not made correctly or at all within the time limits imposed by this study.

Several alternative views of program comprehension are not supported by the research results. For example, views based on strong analogies to chess players' perceptual pattern recognition processes are not supported (cf. Greeno & Simon, 1984). Patterns in program text that are recognized quickly appear to be general, few in number, and are discernible through syntactic markers of the language (programming keywords). Proposals that programs are understood initially through recognition of program plans, from an expert's mental library of hundreds or thousands of plans, that assign roles to configurations of program statements are not supported by this research (Rich & Shrobe, 1979; Soloway et al., 1983). While plan knowledge may well be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organizing principles for memory structures. Claims that data flow relations (Atwood & Ramsey, 1978; Kant & Newell, 1984; Weiser, 1982) or function hierarchies (Adelson, 1984) underlie the preferred or natural representation of programs are also not supported. Our results suggest that the natural representation of programs is procedural, at least for programs written in traditional programming languages. The best comprehenders in the present study were better at inferring function relations than were poorer comprehenders, but they also showed stronger effects for text structure memory organization.

A final result concerns the influence of programming language on comprehension (Green, 1980; Green et al., 1980). There is an indication in the data that programming language may affect the relations represented in initial phases of comprehension and the difficulty of extracting different kinds of information from the program. COBOL programmers were consistently better are responding to comprehension questions about data flow than were FORTRAN programmers. Furthermore, control flow relations were less easily inferred by COBOL programmers. This may be due to features of COBOL and FORTRAN that allow FORTRAN to be programmed in an order corresponding more closely to actual flow of control from statement to statement (at least in these short segments). In COBOL it is more usual to perform loops that are listed elsewhere in the code. Thus the surface structure of the text in COBOL corresponds less well to execution sequence than in FORTRAN. This can be seen by comparing Fig. 1A (program text) and Fig. 6A (TS analysis) in which the loop that sums a list of numbers is executed in Statement 5 but is specified in Statements 11–15. In the FORTRAN version of this segment, the loop

occurs in Statements 5–9 of the code and is executed when encountered there. Thus, there is some evidence that disruption of procedural units in the program text may affect comprehension patterns. Although the data are consistent with a hypothesis that greater difficulty in extracting procedural text units is related to greater difficulty in extracting program function, we must label this conclusion tentative because COBOL and FORTRAN languages and programmers differ in other ways as well.

It is important to question the extent to which the particular task used in the present research limits generalization of these results. For example, the conclusion that comprehension has a more bottom-up character and is organized in memory by procedural control constructs may be specific to understanding small program texts that do not have a larger context. Even if this were the case, it would not sharply dilute the importance of the present experiment. First, most research on programming skill has used similar short texts. Thus findings in this skill domain must be reconciled with the current results. For example developers of the plan knowledge theories have suggested that expert programmers' recall of texts like these is due to recognition of program plans (Greeno & Simon, 1984; Soloway et al., 1983). Of course, the present results suggest that this is not the case; that chunking in recall is explained by the grouping of statements into the sequence, loop, and conditional text units suggested by structured programming advocates. Second, the program segments used in the present research were all texts that were originally embedded within larger programs. To the extent that the larger context does not illuminate all program segments equally, the kind of processing reported in this experiment will certainly occur in actual programming tasks. However, we are also interested in an empirical analysis of the extent to which the first study's results are general across different programming tasks and for longer programming texts. This question is addressed in the second study.

STUDY 2

In the first study, programmers' comprehension strategies may have been influenced by several aspect of the experimental task: short undocumented program segments, the series of short study trials, and the demands of memory questions. In Study 2, a more natural programming environment was created in which programmers studied a program of moderate length (200 lines) and then made a modification to it. At two different points in time they were asked to summarize the program and respond to comprehension questions. Half of the programmers were asked to think aloud while they worked and the other half worked silently.

As in the previous study, comprehension questions were designed to ask about particular relations between program parts: control flow, data

flow, function, and condition-action relations. If the results of the previous study generalize to this task environment, then we expect to see good comprehension of control flow relations early in the comprehension process with comprehension of data flow and function catching up later in the process. Alternatively, data flow and function inferences may be made more readily at the outset due to the larger context in the Study 2 program text.

Methods

Subjects. Forty of the 80 professional programmers who participated in the previous study were invited to return for the second study. These 40 subjects included 20 COBOL and 20 FORTRAN programmers and were those programmers who had scored in the top and bottom quartiles in the comprehension task in the previous study. Subject were run over a period of 6 months from September 1984 to February 1985. Each programmer was paid a $50 fee for participation.

Materials. The stimulus program used for this research is a 200-line program currently in production use at a Chicago firm. The program was one of a series of programs that keeps track of and computes specifications for industrial plant designs. Originally written in COBOL, the program includes both file manipulation and computation. The text was easily translated into a believable FORTRAN program. The program contained a minimal amount of documentation as in the original production version of the program. The documentation included an introductory set of comments describing the program as one that keeps track of the space allocated for wiring (called cables below) and the wiring assigned to that space during the design of a building. No documentation was included within the COBOL text but the FORTRAN version contained one-line comments corresponding to COBOL paragraph headers. Thus the level of documentation in the two versions was judged to be equivalent with the naturally occurring exception that variable names were shorter in FORTRAN.

A modification task was devised that required altering the program to produce an additional output file and an exception list. As with most nontrivial modifications, this task required a relatively complete understanding of the goals of the original program (function), how different variables entered into computations and outputs (data flow), and where in the execution sequence certain transformations occurred (control flow).

A list of 40 comprehension questions was constructed that included 10 questions about control flow (e.g., is a point number grouping processed normally when a type code for a cable is not found?), 10 questions about data flow (e.g., does the value of TPR-WIDTH influence the value of DESIGN-INDEX for a particular point number?), 10 questions about function (e.g., is a report created with point numbers that exceed a DESIGN-INDEX?), and 10 questions about program states (e.g., when the end of the POINT-INDEX file is reached, can there be records in the TEMP-EXCEED file that have not yet been read?). Half of the questions were correctly answered with a "yes" response and half with a "no" response. The 40 questions were divided into two matching lists of 20 questions. For a question on the first list, a similar question was included on the second list so that the two lists contained comparable questions. The lists were arranged in a single random order.

Procedure. Subjects participated in one experimental session lasting approximately 2.5 h at the University of Chicago, Northwestern University, or their place of business. Subjects were familiar with the IBM personal computer used during the session since all subjects had participated in the previous study. All instructions were presented on the display monitor. The first part of the session consisted of general task instructions and detailed instructions concerning the method of displaying and altering the program text, including practice manipulating a program listing using these features.

Programmers were instructed that they were to make a modification to a program normally maintained by another programmer. However the "other programmer" was going on vacation and the modification was urgent. The subjects' task was then to become familiar with the program and to make the changes to it. Furthermore, the hypothetical other programmer had left the program with the subject to study and would return in 45 min to explain the modification task. Subjects accepted this scenario as realistic and meaningful. Thus in the study phase programmers studied the 200-line program for 45 min. Half of the subjects were instructed to think aloud (talk condition) while they studied and the remaining half were allowed to study silently (notalk condition).

The program text was presented on the computer display and subjects could scroll forward or backward, jump to another place in the program, split the screen into halves, and scroll either half. Subjects were also allowed to take notes or draw diagrams while studying the program. Most of the programmers were familiar with studying programs on a terminal but for those who were not, the split screen feature served the purpose of keeping a finger in a listing and jumping ahead in the listing. The program controlling the experiment kept track of the programmer's study sequence by recording which program line was in the center of the display screen.

After the 45-min study period, programmers were asked to type in a summary of the program and then to respond to the first list of 20 comprehension questions. In responding to the comprehension questions, programmers positioned their fingers on "yes" and "no" response keys. They had been instructed to respond yes or no as quickly and accurately as possible. On presentation of a question, subjects responded and then were given an opportunity to explain their responses. They then positioned their fingers to receive the next comprehension question.

After the 20 comprehension questions and a short break, the modification task was explained to the programmer and a time limit of 30 min was specified. Subjects were told that they should begin actual modifications at any point when they felt ready. If necessary the full 30 min could be spent continuing to learn about the program. However, all programmers had at least begun to make modifications by the end of the period and many had completed their changes. During the 30-min modification phase, the talk condition subjects were again asked to think aloud while they worked and the notalk subjects were permitted to work silently.

The session concluded with a second request to summarize the program and then to respond to the second list of 20 comprehension questions. The procedure for these tasks was the same as before. The controlling program recorded all responses, explanations, and times to respond.

Design. Comprehension question responses form the focus of the analyses for the current report in the following research design: 2 (COBOL, FORTRAN languages) × 2 (Q1, Q4 comprehension quartiles) × 2 (talk, notalk conditions) × 2 (comprehension test lists after study, after modification) × 4 (control flow, data flow, state, and function information category of comprehension questions). Language, comprehension quartile, and the talk/notalk condition were between-subjects factors; time of test and information category of the comprehension questions were repeated measures within subjects.

Results

Analyses of the proportion of errors in response to comprehension questions about different kinds of program relations reveal a pattern of errors that varies across information category according to time of comprehension test and talk-aloud condition of the programmer. Conditions at the first time of test, after the 45-min study phase, are most directly

comparable to the conditions of comprehension testing in Study 1. For this reason, results are presented separately below for comprehension after the study phase and after the modification phase. In addition, analyses of program summaries are available for summaries collected after the study phase. Summaries collected after modification could not be analyzed in the same terms because programmers tended to refer to their earlier summaries and then to concentrate on describing their modifications rather than giving complete program summaries as instructed.

Comprehension after study phase. After 45 min of study, the comprehension pattern for comprehension questions about control flow, data flow, program state, and function relations resembles the comprehension pattern observed for Study 1, with questions about control flow answered most accurately, followed closely by data flow. Errors on function questions and program state questions are relatively more frequent, $F(3,96) = 11.64$, $p < .001$ (see Fig. 10). This pattern across information categories did not differ reliably by language, quartile, or talk-aloud conditions (Fs less than 2). Upper and lower quartiles differed in overall level of comprehension, $F(1,32) = 15.75$, $p < .001$, with upper quartile subjects making approximately 40% errors and lower quartile subjects making

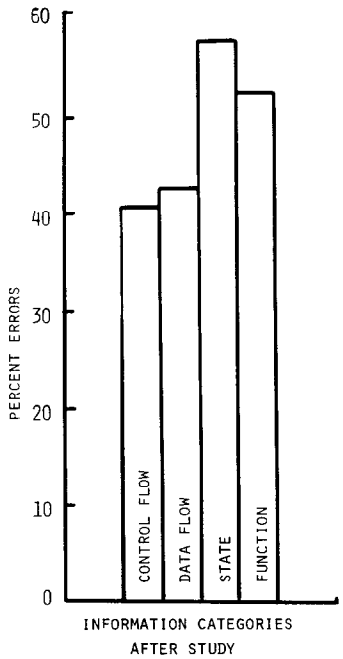

FIG. 10. Study 2 comprehension question error rates by information category, after study task.

60% errors. These error rates are high in part because they have been corrected for guessing by using the explanations provided by the programmer to determine comprehension. Uncorrected error rates averaged 25% for upper quartile subjects and 39% for lower quartile subjects. Analyses performed on uncorrected error rates yielded the same results as those performed on error rates corrected for guessing.

Program summaries were analyzed by classifying each summary statement according to the kind of program relation to which it referred and according to the level of detail specified in the statement. The first classification is referred to as the *type* of summary statement in terms of information categories; types included *procedural, data flow,* and *function* statements. These distinctions are best illustrated by the following excerpts from summaries. Procedural statements include statements of process, ordering, and conditional program actions. The summary of S109 consisted of mostly procedural statements:

> after this, the program will read in the cable file, comparing against the previous point of cable file, then on equal condition compares against the internal table . . . if found, will read the tray-area-point file for matching point-area. In this read if found, will create a type-point-index record. If not found, will read another cable record.

Data flow statements also include statements about data structures. S415 wrote a summary that contains references to many data flow relations:

> The tray-point file and the tray-area file are combined to create a tray-area-point file in Phase 1 of the program. Phase 2 tables information from the type-code file in working storage. The parameter file, cables file, and the tray-area-point file are then used to create a temporary-exceed-index file and a point-index file.

S057 wrote a summary that contains many function statements;

> the program is computing area for cable accesses throughout a building. The amount of area per hole is first determined and then a table for cables and diameters is loaded. Next a cable file is read to accumulate the sum of the cables' diameters going through each hole.

The examples above also differ in the *level of detail* contained in the summaries, the second dimension on which we classified summary statements. Four levels of detail were specified for coding: (a) *detailed* statements contained references to specific program operations and variables; (b) *program* level statements referred to a program's procedural blocks such as a search routine or to files as a whole; (c) *domain* level statements talked about real world objects such as cables and buildings; and (d) *vague* statements did not have specific referents. The excerpts presented above were also chosen because they differ in the predominant level of detail. S109's procedural summary is most detailed; S415's data

flow statements are at a program (file) level; and S057's function statements are at a domain level. An example of a vague statement is, "this program reads and writes a lot of files."

The foregoing examples were chosen for illustrative purposes because they contained a concentration of particular types of statements at a particular level of detail. Most summaries contained a mixture of statement types and levels but can be summarized in terms of general trends across subjects, and comparisons can be made between languages, comprehension quartiles, and talk-aloud conditions.

In terms of statement *type,* the majority (57%) of programmers' summary statements were classified as procedural, 30% were data flow/data structure statements, and 13% were function statements, $F(2,64) = 29.31, p < .001$. This pattern did not differ by quartile, by language, or by talk-aloud condition. In terms of the *level of detail,* classifying the same 100% of the summary statements in a second way, the predominant level was the program/file level accounting for 38% of the statements, 18% of the statements were detailed, 23% were specified at the domain level, and 21% were vague, $F(3,96) = 10.47, p < .001$. This pattern across level of detail differed for upper and lower quartile subjects, $F(3,96) = 4.65, p < .01$, with lower quartile comprehenders' summaries containing relatively more statements at a detailed level (20% Q4 versus 16% Q1) and more statements at a vague level (30% Q4 versus 14% Q1). A final observation concerns a relation observed between summary statement type and level. A majority of program summary statements about program function were expressed in the language of real world objects (cables, space, crowding, etc.) rather than in the language of programs. The majority of procedural summary statements were expressed in terms of program objects (files, computations, searching, etc.) rather than in the domain language.

Comprehension after modification phase. Looking again at comprehension errors for different information categories, the comprehension pattern shifts on the second comprehension test after the modification task, $F(3,96) = 9.77, p < .001$ (comprehension trial × information category interaction in an ANOVA treating comprehension trial as a repeated measure). The pattern of errors for this trial (see Fig. 11) shows the fewest errors for data flow and function questions with more errors on control flow questions, $F(3,96) = 14.85, p < .001$. Furthermore, this pattern is more exaggerated for the programmers who talked aloud while working, $F(3,96) = 5.93, p < .001$ (see Fig. 11). Although second-trial patterns are exaggerated for talk subjects, overall comprehension accuracy for talk and notalk subjects was roughly equivalent, $F(1,32) = 1.01$. Patterns of errors across information category did not differ by comprehension quartile or by language.

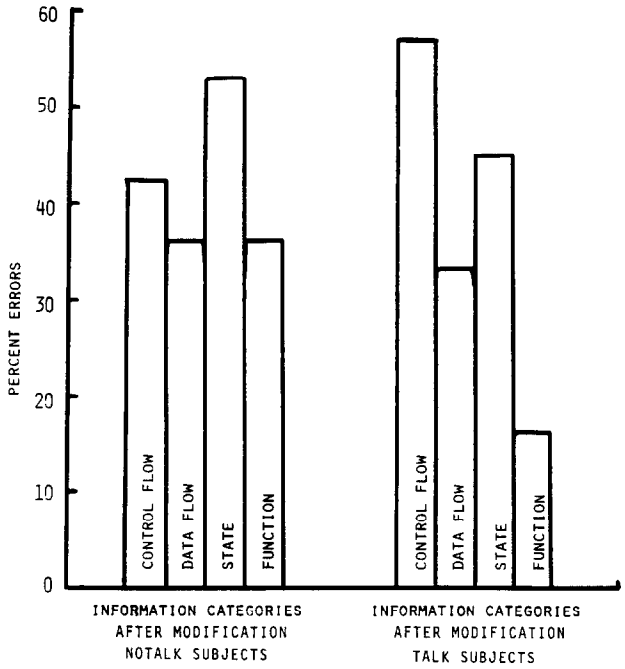

FIG. 11. Study 2 comprehension question error rates by information category, after modification task, for talk and notalk subjects.

Discussion

Comprehension results from Study 2 (Fig. 10) reinforce and extend the conclusions from Study 1 that the understanding of program control flow and procedures precedes understanding of program functions. This pattern of comprehension results appeared even in the context of a longer, partially documented program after a lengthy period. Analyses of program summaries also support this conclusion by showing a preponderance of procedural summary statements over data flow and function statements.

It is important to note that the story of program comprehension does not end with the establishment of a procedural representation. In Study 2 a different comprehension pattern emerged after a second exposure to the program during which programmers completed a program modification (Fig. 11). After the modification task, there was a marked shift toward increased comprehension of program function and data flow at the apparent expense of control flow information, and this shift was more extreme for programmers who were asked to think aloud while working. This suggests that either the additional time or the goal of modifying the program resulted in a *change* in the dominant memory representation.

The fact that talking aloud while working enhanced this shift suggests that task effects, rather than the extra time alone, are responsible.

One way to understand this shift in comprehension patterns is to go back to theories of text comprehension and speculate about a construct, introduced by van Dijk and Kintsch (1983), that they call a situation model. In this (1983) work, van Dijk and Kintsch suggest that two distinct but cross-referenced representations of a text are constructed during comprehension. The first representation, the *textbase,* includes the hierarchy of representations, described in the introduction to the present paper, consisting of a surface memory of the text, a microstructure of interrelations between text propositions, and a macrostructure that organizes the text representation. The second representation, the *situation model* is a mental model (e.g., Johnson-Laird, 1983) of what the text is about referentially. In our context, the program text in Study 2 is conceptually about searches, merges, computations, and so forth; referentially, it is about cables that take up space, finding out how big a particular cable is, computing the total size of the cables allocated to a particular space, comparing the cable allocation to the size of the space, etc. It is plausible that the functional relations between program procedures are more comprehensible in the terms of the real world objects. Thus, the textbase macrostructure may be dominated by procedural relations that largely reflect how programs in traditional languages are structured. The functional hierarchy can be developed with reference to a situation model expressed in terms of the real world objects. Data from our analysis of program summaries are consistent with this idea. Procedural summary statements were most often expressed in terms of program concepts and functional summary statements were most often expressed in terms of the real world object domain.

Van Dijk and Kintsch (1983) also suggest that the construction of the situation model depends on construction of the textbase in the sense that the textbase defines the actions and events that need explaining. This is consistent with our findings in both studies that procedural representations precede functional representations. In fact our results suggest that both time and incentive (talking aloud to an experimenter and having to do a modification) are involved in the successful construction of a functionally based situation model. If this analysis is correct, we could imagine conditions that might assist and speed up the extraction of program function and the construction of a functional representation. For example, documentation concerning the real world domain and the relation of program procedures to the domain might promote a simultaneous construction of both kinds of understanding.

One final aspect of the results of Study 2 deserves comment. Comprehension quartile as determined by comprehension scores in the experi-

mental setting of Study 1 predicted the comprehension scores in the more natural task of Study 2. However, the error rates on comprehension questions for *both* upper and lower quartile comprehenders were quite high in Study 2, even after 1.25 h of study and modification. Is this cause for practical concern, considering the fact that we are studying professional programmers with an average of 10 years of experience—people who are responsible for the programs that help design buildings, monitor space programs, keep track of bank balances, control defense systems and so on? The high error rates are not by themselves cause for concern because programmers were answering questions without reference to the program listing. It does not necessarily follow that the same errors would be made if subjects could have "looked up" the answers in the program. Greater concern would be warranted if we found that the high error rates were accompanied by great confidence in level of understanding, a measure we did not collect. But, our casual observations of subjects who talked while working suggest that this may have been the case for some of the programmers.

GENERAL DISCUSSION

At the outset we presented an analysis of computer program texts in terms of multiple abstractions of the text to illustrate relations between parts of programs. Specific abstractions expressing important relations in the design of computer programs include a goal hierarchy highlighting major functional achievements of a program (Fig. 2), a data flow abstraction highlighting the transformations that are applied to data objects (Fig. 3), a control flow abstraction highlighting the temporal sequence of execution of program actions (Fig. 4), and a conditionalized action representation specifying states of the program and the actions invoked (Fig. 5). Although our example analysis is specific to computer programming, analogies can be developed for closely related tasks that involve other kinds of texts such as instructional texts, and more distant analogies for design tasks in which other kinds of relations are more central.

The views of computer program comprehension contrasted throughout this report, based on analyses of plan knowledge (PK) and text structure knowledge (TS), represent claims about which kinds of relations outlined in the multiple abstractions analysis play a central organizing role in program comprehension. PK theory suggests that data flow and function relations will be dominant and TS theory suggests that control flow or procedural relations will be central. More generally, these two views represent positions about the role of different kinds of knowledge in comprehension. TS theory emphasizes the role of abstract knowledge of program text structures, while PK theory emphasizes the role of a large

collection of content-dependent knowledge that links specific program functions to plans that achieve them.

The present research results strongly support a view of program comprehension in which abstract knowledge of program text structures plays the *initial* organizing role in memory for programs, and that control flow or procedural relations dominate in the macrostructure memory representation. These results are consistent with conclusions reached by researchers in other text comprehension domains who suggest that knowledge of narrative and expository text structures guides comprehension processing and plays an important role above and beyond other content-schematic factors (e.g., Carpenter & Just, 1981; Cirilo & Foss, 1980; Haberlandt, Berian, & Sandson, 1980; van Dijk & Kintsch, 1983; Johnson & Mandler, 1980; Kieras, 1985; Mandler, 1978, 1984; Mandler & Johnson, 1977; Rumelhardt, 1975, 1980; Stein & Glenn, 1979; Thorndyke, 1977). These results are not consistent with conclusions suggesting such knowledge is not involved in comprehension in domains where extensive content knowledge may be available (e.g., Black & Bower, 1980; Black & Wilensky, 1979; Bruce, 1980; Schank & Abelson, 1977; Thorndyke & Yekovich, 1980). Thus as there is good evidence that "episodes" function as psychological units in story comprehension, there is also good evidence that structured programming building blocks function as psychological units in program comprehension.

In terms of the multiple abstractions analysis, programmers' mental representations in this research were closest to the procedural representation (Fig. 4) based on control flow relationships. Should we then conclude that a procedural form is the "natural" mental representation? In the current research, we originally expected that mental representations would show function and data flow relations to be primary. If that had occurred, then there would be ample ground to claim that these relations reflected a "natural" or preferred cognitive organization because text and language structure as well as the programmers' training combine to highlight procedural relations. However, given the current results, we are not sure whether the mental organization reflects language/text structure and training, or cognitive "naturalness," or both. There is some evidence from research on the comprehension of procedural instructions that the memory structure reflects procedural relations rather than functional relations whether or not the text from which the procedure is learned has a procedural form (Smith & Spoehr, 1984). On the other hand, the language differences found in the present research suggest that language structure will matter, that the form in which it is convenient to mentally represent a design will be a form that is closely related to the structure of the stimulus. This is consistent with an emphasis that was popular in earlier problem-solving research: Stimulus structures are a major influence on

the form of mental representations, even for logically isomorphic problems (Hayes & Simon, 1977).

We also found evidence that in *later* stages of program comprehension, under appropriate task conditions, a second representation is available that reflects the functional structure of the program and is expressed in the language of the real world domain to which the program is applied. Our explanations for this later, task-related shift in comprehension are speculative and draw on the concept of a *situation model* representation of the program that is distinct from the macrostructure organization of the textbase (van Dijk & Kintsch, 1983). What is clear from our research is that this second, functional representation is not constructed quickly or automatically. Programmers required extensive involvement with the program before being able to use this structure to respond to questions about the program. Further research is needed to explore the viability of the situation model explanation and the extent to which changes in stimulus structures will alter the time course of its emergence.

REFERENCES

Adams, M., & Collins, A. (1979). A schema-theoretic view of reading. In R. Freedle (Ed.), *New directions in discourse processing* (Vol. 2). Norwood, NJ: Ablex.

Adelson, B. (1981). Problem solving and the development of abstract categories in programming language. *Memory & Cognition*, **9**, 422–433.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, **10**, 484–495.

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge MA: Harvard Univ. Press.

Atwood, M. E., & Jeffries, R. (1980). *Studies in plan construction. I: Analysis of an extended protocol* (Tech. Rep. No. SAI-80-028-DEN). Englewood, CO: Science Publications, Inc.

Atwood, M. E., & Ramsey, H. R. (1978). *Cognitive structures in the comprehension and memory of computer programs: An investigation of computer program debugging* (Tech. Rep. No. SAI-78-054-DEN). Englewood, CO: Science Applications, Inc.

Basili, V. R., & Mills, H. D. (1982). Understanding and documenting programs. *IEEE Transactions on Software Engineering*, **SE-8**, 270–283.

Black, J. B., & Bower, G. H. (1980). Story understanding as problem-solving. *Poetics*, **9**, 223–250.

Black, J. B., & Wilensky, R. (1979). An evaluation of story grammars. *Cognitive Science*, **3**, 213–230.

Bisanz, G. L., & Voss, J. F. (1981). Sources of knowledge in reading comprehension: Cognitive development and expertise in a content domain. In A. M. Lesgold & C. A. Perfetti (Eds.), *Interactive processes in reading*. Hillsdale, NJ: Erlbaum.

Britton, B. K., & Black, J. B. (Eds.). (1985). *Understanding expository text*. Hillsdale, NJ: Erlbaum.

Brooks, R. E. (1975). *A model of human cognitive behavior in writing code for computer programs*. Unpublished doctoral dissertation, Carnegie-Mellon University, Pittsburgh.

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, **18**, 543–554.

Bruce, B. (1980). Analysis of interacting plans as a guide to the understanding of memory structure. *Poetics*, **9**, 295–312.

Carpenter, P. A., & Just, M. A. (1981). Cognitive processes in reading: Models based on readers' eye fixations. In A. M. Lesgold & C. A. Perfetti (Eds.), *Interactive processes in reading*. Hillsdale, NJ: Erlbaum.

Chase, W. G., & Simon, H. A. (1973a). The mind's eye in chess. In W. G. Chase (Ed.), *Visual information processing*. New York: Academic Press.

Chase, W. G., & Simon, H. A. (1973b). Perception in chess. *Cognitive Psychology*, **4**, 55–81.

Chi, M. T. H., Glaser, R., & Rees, E. (1982). Expertise in problem solving. In R. J. Sternberg (Ed.), *Advances in the psychology of intelligence*. Hillsdale, NJ: Erlbaum.

Cirilo, R. K., & Foss, D. J. (1980). Text structure and reading time for sentences. *Journal of Verbal Learning and Verbal Behavior*, **19**, 96–109.

Curtis, B., Forman, I., Brooks, R. E., Soloway, E., & Ehrlich, K. (1984). Psychological perspectives for software science. *Information Processing and Management*, **20**, 81–96.

Dahl, O. J., Dijkstra, E., & Hoare, C. A. R. (1972). *Structured programming*. Orlando/London: Academic Press.

Davis, J. S. (1984). Chunks: A basis for complexity measurement. *Information Processing and Management*, **20**, 119–127.

van Dijk, T. A., & Kintsch, W. (1983). *Strategies of discourse comprehension*. New York: Academic Press.

Engle, R. W., & Bukstel, L. (1978). Memory processes among bridge players of differing expertise. *American Journal of Psychology*, **91**, 673–689.

Fodor, J. A., Bever, T. G., & Garrett, M. F. (1974). *The psychology of language*. New York: McGraw-Hill.

Glucksberg, S., & McCloskey, M. (1981). Decisions about ignorance: Knowing that you don't know. *Journal of Experimental Psychology: Human Learning and Memory*, **7**, 311–325.

Green, T. R. G. (1980). Programming as a cognitive activity. In H. T. Smith & R. R. G. Green (Eds.), *Human interaction with computers*. New York: Academic Press.

Green, T. R. G., Sime, M. E., & Fitter, M. J. (1980). The problems the programmer faces. *Ergonomics*, **23**, 893–907.

Greeno, J. G., & Simon, H. A. (1984). *Problem solving and reasoning* (Tech. Rep. No. UPITT/LRDC/ONR/APS-14). Pittsburgh, PA: University of Pittsburgh.

Haberlandt, K. (1980). Story grammar and reading time of story constituents. *Poetics*, **9**, 99–118.

Haberlandt, K., Berian, C., & Sandson, J. (1980). The episode schema in story processing. *Journal of Verbal Learning and Verbal Behavior*, **19**, 635–650.

Halpern, A. R., & Bower, G. H. (1982). Musical expertise and melodic structure in memory for musical notation. *American Journal of Psychology*, **95**, 31–50.

Hayes, J. R., & Simon, H. A. (1977). Psychological differences among problem isomorphs. In N. J. Castellan, D. B. Pisoni, & G. R. Potts (Eds.), *Cognitive theory* (Vol. 2). Hillsdale, NJ: Erlbaum.

Johnson, N. S., & Mandler, J. M. (1980). A tale of two structures: Underlying and surface forms in stories. *Poetics*, **9**, 51–86.

Johnson-Laird, P. N. (1980). *Mental models*. Cambridge, MA: Harvard Univ. Press.

Kant, E., & Newell, A. (1984). Problem solving techniques for the design of algorithms. *Information Processing and Management*, **20**, 97–118.

Kieras, D. (1985). Thematic processes in the comprehension of technical prose. In B. K. Britton & J. B. Black (Eds.), *Understanding expository text*. Hillsdale, NJ: Erlbaum.

Kintsch, W. (1974). *The representation of meaning in memory*. Hillsdale, NJ: Erlbaum.

Kintsch, W. (1977). On comprehending stories. In M. A. Just & P. A. Carpenter (Eds.), *Cognitive processes in comprehension*. Hillsdale, NJ: Erlbaum.

Kintsch, W., & van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review*, **85**, 363–394.

Linger, R. C., Mills, H. D., & Witt, B. I. (1979). *Structured programming: Theory and practice*. Reading, MA: Addison-Wesley.

Mandler, J. M. (1978). A code in the node: The use of a story schema in retrieval. *Discourse Processes*, **1**, 14–35.

Mandler, J. M. (1984). *Stories, scripts and scenes: Aspects of schema theory*. Hillsdale, NJ: Erlbaum.

Mandler, J. M., & Goodman, M. S. (1982). On the psychological validity of story structure. *Journal of Verbal Learning and Verbal Behavior*, **21**, 507–523.

Mandler, J. M., & Johnson, N. S. (1977). Remembrance of things parsed: Story structure and recall. *Cognitive Psychology*, **9**, 111–151.

Mayer, R. E. (1977). A psychology of learning BASIC. *Communications of the ACM*, **22**, 589–593.

McKeithen, K. B., Reitman, J. S., Reuter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, **13**, 307–325.

McKoon, G., & Ratcliff, R. (1980). Priming in item recognition: The organization of propositions in memory for text. *Journal of Verbal Learning and Verbal Behavior*, **19**, 369–386.

McKoon, G., & Ratcliff, R. (1984). Priming and on-line text comprehension. In D. E. Kieras & M. A. Just (Eds.), *New methods in reading comprehension research*. Hillsdale, NJ: Erlbaum.

Meyer, B. J. F. (1975). *The organization of prose and its effect upon memory*. Amsterdam: North-Holland.

Miller, J. R. (1985). A knowledge-based model of prose comprehension: Applications to expository texts. In B. K. Britton & J. B. Black (Eds.), *Understanding expository text*. Hillsdale, NJ: Erlbaum.

Mitchell, D. C., & Green, D. W. (1978). The effects of context and content on immediate processing in reading. *Quarterly Journal of Experimental Psychology*, **30**, 609–636.

Newell, A., & Simon, H. A. (1972). *Human problem solving*. New York: Prentice-Hall.

Norcio, A. F., & Kerst, S. M. (1983). Human memory organization for computer programs. *Journal of the American Society for Information Science*, **34**, 109–114.

Pennington, N. (1985). Cognitive components of expertise in computer programming: A review of the literature. *Psychological Documents*, **15**, No. 2702.

Ratcliff, R., & McKoon, G. (1978). Priming in item recognition: Evidence for the propositional structure of sentences. *Journal of Verbal Learning and Verbal Behavior*, **17**, 403–417.

Reder, L. M. (1982). Plausibility judgments versus fact retrieval: Alternative strategies for sentence verification. *Psychological Review*, **89**, 250–280.

Reitman, J. S. (1976). Skilled perception in GO: Deducing memory structures from inter-response times. *Cognitive Psychology*, **8**, 336–356.

Rich, C. (1980). *Inspection methods in programming* (Tech. Rep. No. 604). Cambridge, MA: MIT Artificial Intelligence Laboratory.

Rich, C. (1981). A formal representation for plans in the programmer's apprentice. *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Vancouver, Canada.

Rich, C., & Shrobe, H. E. (1979). Design of a programmer's apprentice. In *Artificial intelligence: An MIT perspective*. Cambridge, MA: MIT Press.

Rich, C., & Waters, R. C. (1981). *Abstraction, inspection and debugging in programming* (Memo No. 634). Cambridge, MA: MIT Artificial Intelligence Laboratory.

Rumelhardt, D. E. (1975). Notes on a schema for stories. In D. G. Bobrow & A. Collins (Eds.), *Representation and understanding: Studies in cognitive science*. New York: Academic Press.

Rumelhardt, D. E. (1980). A reply to Black and Wilensky. *Cognitive Science*, **4**, 313–316.

Schank, R. C., & Abelson, R. B. (1977). *Scripts, plans, goals, and understanding*. Hillsdale, NJ: Erlbaum.

Schmidt, A. L. (1983). *Comprehension of computer programs by expert and novice programmers*. Unpublished doctoral dissertation, Southern Illinois University, Carbondale, IL.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer and Information Science*, **5**, 123–143.

Shneiderman, B. (1980). *Software psychology*. Cambridge, MA: Winthrop.

Shrobe, H. E. (1979). *Dependency directed reasoning for complex program understanding* (Tech. Rep. No. 503). Cambridge, MA: MIT Artificial Intelligence Laboratory.

Sloboda, J. A. (1976). Visual perception of musical notation: Registering pitch symbols in memory. *Quarterly Journal of Experimental Psychology*, **28**, 1–16.

Smith, E. E., & Spoehr, K. (1984, June). *Comprehension of instructions for operating devices*. Paper presented at ONR Conference, Tucson, AZ.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **SE-10**, 595–609.

Soloway, E., Ehrlich, K., & Black, J. B. (1983). Beyond numbers: Don't ask "how many" . . . ask "why." *Proceedings of the Conference on Human Factors in Computer Systems*, Boston, MA.

Soloway, E., Ehrlich, K., & Bonar, J. (1982). Tapping into tacit programming knowledge. *Proceedings of the Conference on Human Factors in Computer Systems*, Gaithersburg, MD.

Stein, N. L., & Glenn, C. G. (1979). An analysis of story comprehension in elementary school children. In R. Freedle (Ed.), *New directions in discourse processing* (Vol. 2). Norwood, NJ: Ablex.

Tejirian, E. (1968). Syntactic and semantic structure in the recall of orders of approximation to English. *Journal of Verbal Learning and Verbal Behavior*, **7**, 1010–1015.

Thorndyke, P. W. (1977). Cognitive structures in comprehension and memory of narrative discourse. *Cognitive Psychology*, **9**, 77–110.

Thorndyke, P. W., & Yekovich, F. R. (1980). A critique of schemata as a theory of human story memory. *Poetics*, **9**, 23–50.

Trabasso, T., Secco, T., & van den Broeck, P. (1982). Causal cohesion and story coherence. In H. Mandl, N. L. Stein, & T. Trabasso (Eds.), *Learning and comprehension of text*. Hillsdale, NJ: Erlbaum.

Waters, R. C. (1979). A method for analyzing loop programs. *IEEE Transactions on Software Engineering*, **SE-5**, 237–247.

Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM*, **25**, 446–452.

Wilensky, R. (1983). *Planning and understanding*. Reading, MA: Addison–Wesley.

(Accepted November 11, 1986)