END

DATE
FILMED
6-87

DTIC

①

AD-A179 392

DTIC

Stimulus Structures and Mental Representations
in Expert Comprehension of Computer Programs

Nancy Pennington
Center for Decision Research
Graduate School of Business
University of Chicago

September 1986

DTIC
ELECTE
APR 2 0 1987
S E D

In Press, Cognitive Psychology

87 4 16 123

8

ABSTRACT

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 eighty 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 forty 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.

INTRODUCTION

Computer programming is a complex cognitive task composed of a variety of subtasks and involving several kinds of specialized knowledge (Pennington & Grabowski, 1986). 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 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.

1

REPORT DOCUMENTATION PAGE

AD-A179 591

Form Approved
OMB No 0704 0188

1a REPORT SECURITY CLASSIFICATION	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution unlimited.

4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
Technical Report No. 3-ONR	

6a NAME OF PERFORMING ORGANIZATION	6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION
Graduate School of Business University of Chicago		Personnel and Training Research Programs Office of Naval Research (Code 1142PT)

6c ADDRESS (City, State, and ZIP Code)	7b ADDRESS (City, State, and ZIP Code)
1101 E. 58th Street Chicago, IL 60637	800 North Quincy Street Arlington, VA 22217-5000

8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
		N00014-82-K-0759

8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
	61153N	RR04206	RR04206-0A	NR667-503

11 TITLE (Include Security Classification)

Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs

12 PERSONAL AUTHOR(S)

Pennington, Nancy

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical Report	FROM 2/85 TO 8/85	September 1, 1986	50

16 SUPPLEMENTARY NOTATION

in press, Cognitive Psychology

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	computer programming, expertise, text comprehension cognitive skill, problem solving, software psychology
05	09		

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This report is a complete revision and extension of a previous technical report (Technical Report No. 2-ONR, January, 1985) with the same title; additional data are reported in this version.

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 eighty professional programmers were tested on comprehension and recognition of short computer program texts. The results suggest

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL
Dr. Michael Shafto	202-696-4596	ONR 1142PT

DD Form 1473, JUN 86   Previous editions are obsolete.   SECURITY CLASSIFICATION OF THIS PAGE

19. (cont.)

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 forty 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.

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

2

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, 1982). 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 it resemblance to English (see Figure 1.A). 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 Figure 1.B).

****
Insert Figure 1 about here
****

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 Figure 2). It is labeled a goal hierarchy but could also be described as 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 abstraction there is little explicit information as to how these goals will be accomplished. For example, the total list of clients could be 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

3

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-CLIE
    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.

figure 1

everyday knowledge; for example, the orders must be summed before division can take place.

```
****************************
Insert Figure 2 about here
****************************
```

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 Figure 3). For example, Figure 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 goal hierarchy shown in Figure 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 at the bottom of the data flow abstraction shown in Figure 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 produces a data object (e.g., "sum orders") must execute prior to the process that consumes it ("compute average").

```
****************************
Insert Figure 3 about here
****************************
```

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 Figure 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 the data flow abstraction, Figure 3) it is necessary to track its use in the sequence of operations in the control flow abstraction (Figure 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.

```
****************************
Insert Figure 4 about here
****************************
```

A fourth abstraction is structured in terms of the program actions that

4

GOAL HIERARCHY: THE PROGRAM ACCOMPLISHES CERTAIN GOALS BY
PRODUCING OUTPUTS. EACH LEVEL INDICATES A
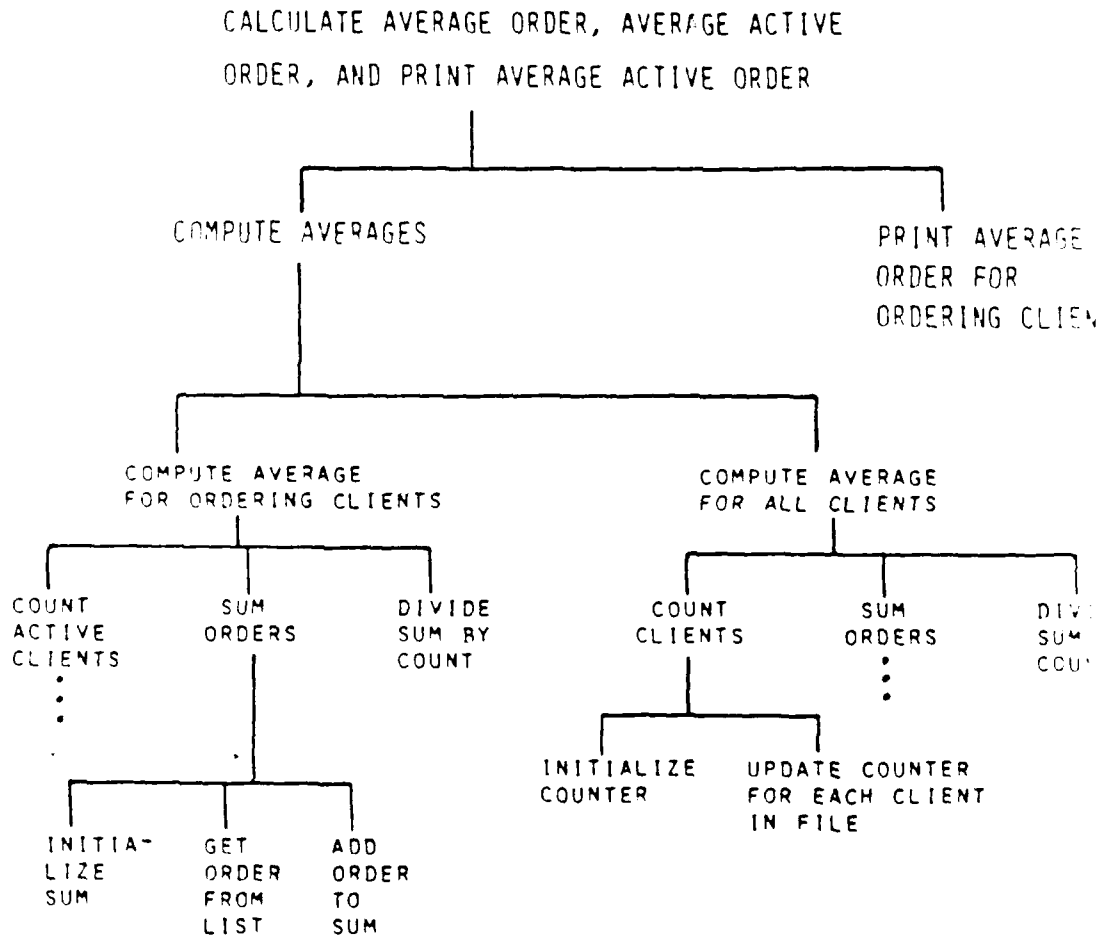HIGHER ORDER GOAL IS DECOMPOSED INTO SUBGOALS.

CALCULATE AVERAGE ORDER, AVERAGE ACTIVE
ORDER, AND PRINT AVERAGE ACTIVE ORDER

COMPUTE AVERAGES

PRINT AVERAGE
ORDER FOR
ORDERING CLIEN

COMPUTE AVERAGE
FOR ORDERING CLIENTS

COMPUTE AVERAGE
FOR ALL CLIENTS

COUNT
ACTIVE
CLIENTS

SUM
ORDERS

DIVIDE
SUM BY
COUNT

COUNT
CLIENTS

SUM
ORDERS

DIV
SUM
CCU

INITIA-
LIZE
SUM

GET
ORDER
FROM
LIST

ADD
ORDER
TO
SUM

INITIALIZE
COUNTER

UPDATE COUNTER
FOR EACH CLIENT
IN FILE

Figure 2

DATA FLOW: PROGRAM ACTIONS TRANSFORM INITIAL DATA OBJECTS
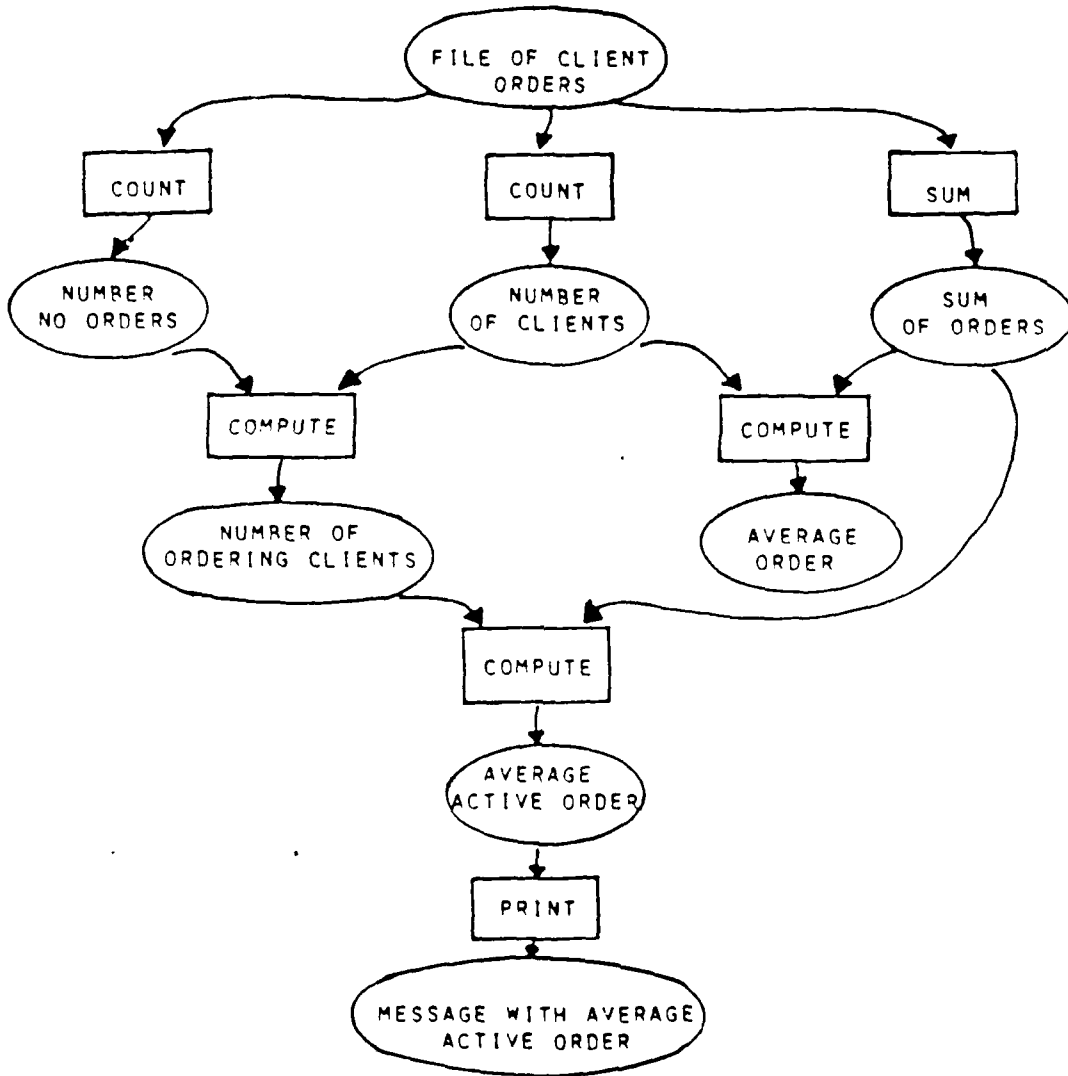INTO FINAL DATA OBJECTS. ⬭ INDICATE DATA
OBJECTS. ▭ INDICATE PROGRAM ACTIONS.

FILE OF CLIENT ORDERS

COUNT | COUNT | SUM

NUMBER NO ORDERS | NUMBER OF CLIENTS | SUM OF ORDERS

COMPUTE | COMPUTE

NUMBER OF ORDERING CLIENTS | AVERAGE ORDER

COMPUTE

AVERAGE ACTIVE ORDER

PRINT

MESSAGE WITH AVERAGE ACTIVE ORDER

Figure 3

CONTROL FLOW: PROGRAM ACTIONS OCCUR IN A SPECIAL SEQUENCE.

INITIALIZE COUNTER FOR CLIENTS
INITIALIZE SUM FOR ORDERS
INITIALIZE COUNTER FOR NO ORDERS
INITIAL READ CLIENT RECORD

TEST FOR FILE TRAILER: EXIT
        INCREMENT COUNTER FOR CLIENTS
        ADD CLIENT ORDER TO SUM

        TEST FOR NO ORDER:
                INCREMENT COUNTER FOR NO ORDER

        READ CLIENT RECORD

COMPUTE COUNT OF CLIENTS WITH ORDERS
COMPUTE AVERAGE OVER ALL CLIENTS
COMPUTE AVERAGE OVER ORDERING CLIENTS
PRINT AVERAGE OVER ORDERING CLIENTS

Figure 4

will result when a particular set of conditions is true (see Figure 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 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.

****
Insert Figure 5 about here
****

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 functions 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, make 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, 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 should 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 tradeoffs 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 Figure 2, characterizes

5

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.

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										

Figure 5

the natural cognitive representation of experienced programmers while a procedural representation, reflecting text relations specified in Figure 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 Figures 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 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 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 (Figure 1.A) in terms of these prime program text structure (TS) units is shown in Figure 6.A (cf., Curtis, et al., 1984). For example statements numbered 1 through 4 in Figure 1.A form a sequence unit as shown in Figure 6.A; statements 5, 11, and the embedded sequence 12 through 15 form an iteration unit (loop); and statements numbered 6 through 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.

6

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 Figure 6.A 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 (Figure 6.A) is most closely related to our earlier analysis of program text in terms of control flow relations (Figure 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 Figure 6.A.

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
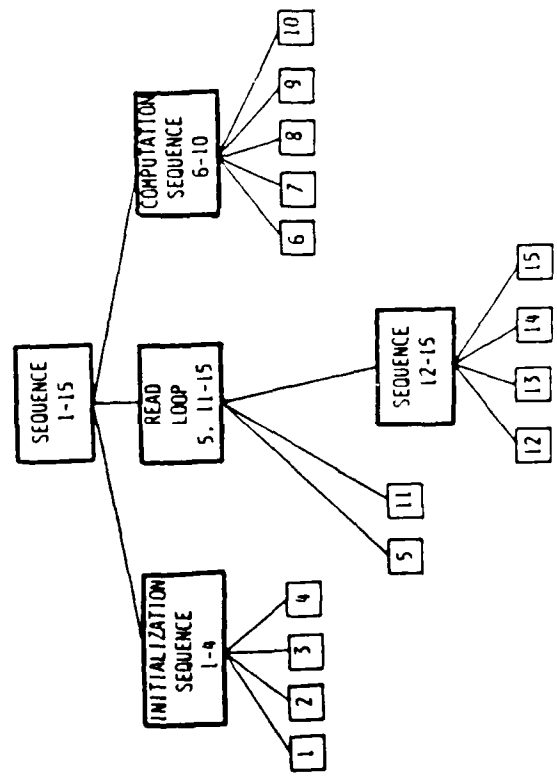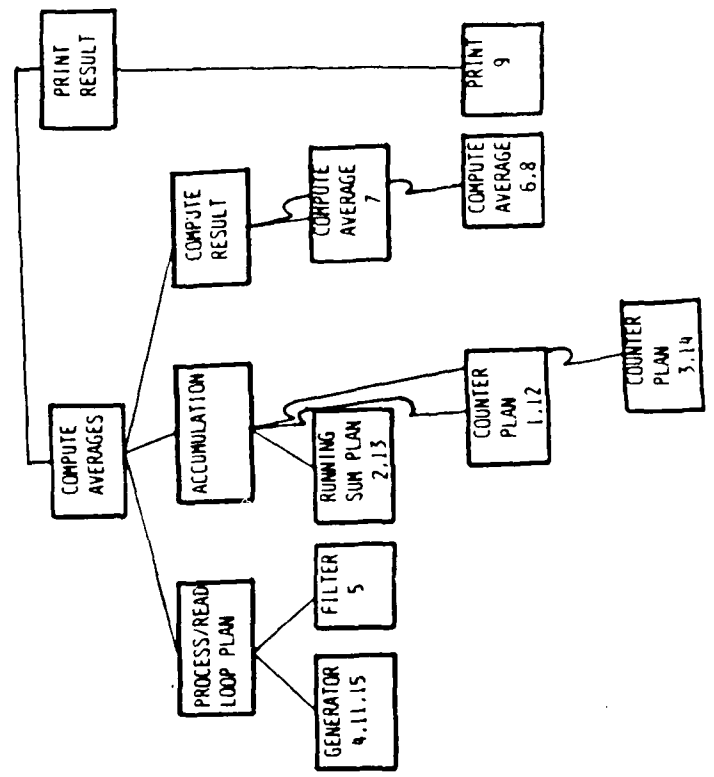
7

A. TEXT STRUCTURE ANALYSIS

B. PLAN KNOWLEDGE ANALYSIS

Figure 6

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 (5 lines each) or as five syntactic groupings (3 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 phrase-like 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 (Figures 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, & 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

8

(Soloway & Ehrlich, 1984; Soloway, et al., 1983; Soloway, et al., 1982) who have developed a more psychologically motivated theory of plan knowledge. A plan knowledge (PK) decomposition of the COBOL program segment (Figure 1.A) in terms of underlying program plans is shown in Figure 6.B (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 Figure 6.B refer to program statement numbers specified in Figure 1.A and the hierarchical structure of the diagram in Figure 6.B 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 6.B.

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 (Figure 6.B) is most closely related to our earlier analysis of program text in terms of data flow relations (Figure 3) and function (Figure 2). Such a decomposition also does not necessarily correspond to the text surface structure as shown by program text statement numbers in Figure 6.B.

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

9

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
(Figures 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 Figures 6.A
and 6.B 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.
****
Insert Table 1 about here
****

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 debates 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

10

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 activate 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 very 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 ONE

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

11

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 farther 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 Figure 1.A. 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 Figure 6.A, 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 (Figure 6.B) 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 cognitive 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 co-varying 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 (Figure 2); data flow relations specifying the sets of events in the program in which particular variables participate (Figure 3); control flow relations specifying the execution sequence of statements (Figure 4); and state relations specifying the sets of conditions and resulting actions in the program (Figure 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, function 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 ask specifically about these different kinds of relations in the program.

Methods

Subjects. Professional programmers with a minimum of three 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 6 other programming languages, had taken 4 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 4 other programming languages, had taken 4 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 eight months from July, 1983 to February, 1984. Each programmer was paid $10.00 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

13

file update program, an engineering application, 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 Figure 1.A. 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 6.A and 6.B show the TS and PK theoretical analyses of the example program segment.

For each of the 8 program segments, 6 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 Figure 1.A 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 8 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 (Figure 1.A) form a triple. Statement 4 is the TS prime because statements 4 and 2 are in the same TS cognitive unit (Figure 6.A) but in different PK units (Figure 6.B). Statement 13 is the Pk prime because statements 13 and 2 are in the same PK cognitive unit (Figure 6.B) but are in different TS units (Figure 6.A). Statement 2 is the target item since it appears in both pairs. For each program segment, 4 target items were identified along with their TS and PK primes and the remaining 3 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 (8 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

14

consisted of 3 "filler" true items and 6 prime-target pairs. The 6 targets
consisted 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
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 hours 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 minutes, divided into three 1.5 minute 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 minutes. 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
the 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

15

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 minutes was chosen to insure 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) x 4 (orders) x 2 (subject groups within language) x 2 (TS, PK prime types) x 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 2x2x2 repeated measures latin square. In this design, comparisons between target response times for different prime types will be 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

16

extreme response times 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 2x2x4x2 repeated measures analysis of variance with language (FORTRAN, COBOL), subject group (1, 2), and test list order (4 orders) as between subjects factors and type of prime (PK, TS) as a within subject repeated measure. A second 2x2x2 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).

<div align="center">

****

Insert Table 2 about here

****

</div>

As predicted by a text structure (TS) analysis of program comprehension, responses to TS-primed targets are on average 105 milliseconds faster than responses to PK primed targets, $F(1,64) = 4.51$, $p < .04$ (subjects analysis, see Table 2, Part 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, Part B) that the TS-primed speedup is larger, 237 milliseconds, $F(1,15) = 8.35$, $p < .02$ (subjects analysis), $F(1,59) = 3.60$, $p < .06$ (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 language, and for the subject groups within each language. Responses to B materials took an average of 26 milliseconds longer than responses to A materials, $F(1,64) = 28.18$, $p < .001$ (subjects analysis), $F(1,60) = 2.32$, $p < 0.14$ (items analysis), and this difference was larger for COBOL items than for FORTRAN items, $F(1,64) = 6.03$, $p < .02$ (subjects analysis, not significant for items analysis). COBOL subjects took longer to respond in general, $F(1,64) = 4.91$, $p < .04$ (subjects analysis), $F(1,60) = 4.88$, $p < .04$ (items 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 Figure 7). In Figure 7.A, response times are adjusted for the effect of subject groups within lanuage, showing the TS priming effect for both languages, the effect for materials set and the slight interaction between materials and language. In Figure 7.B, 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.

<div align="center">

****

Insert Figure 7 about here

****

</div>

Overall recognition accuracy, measured by percent 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
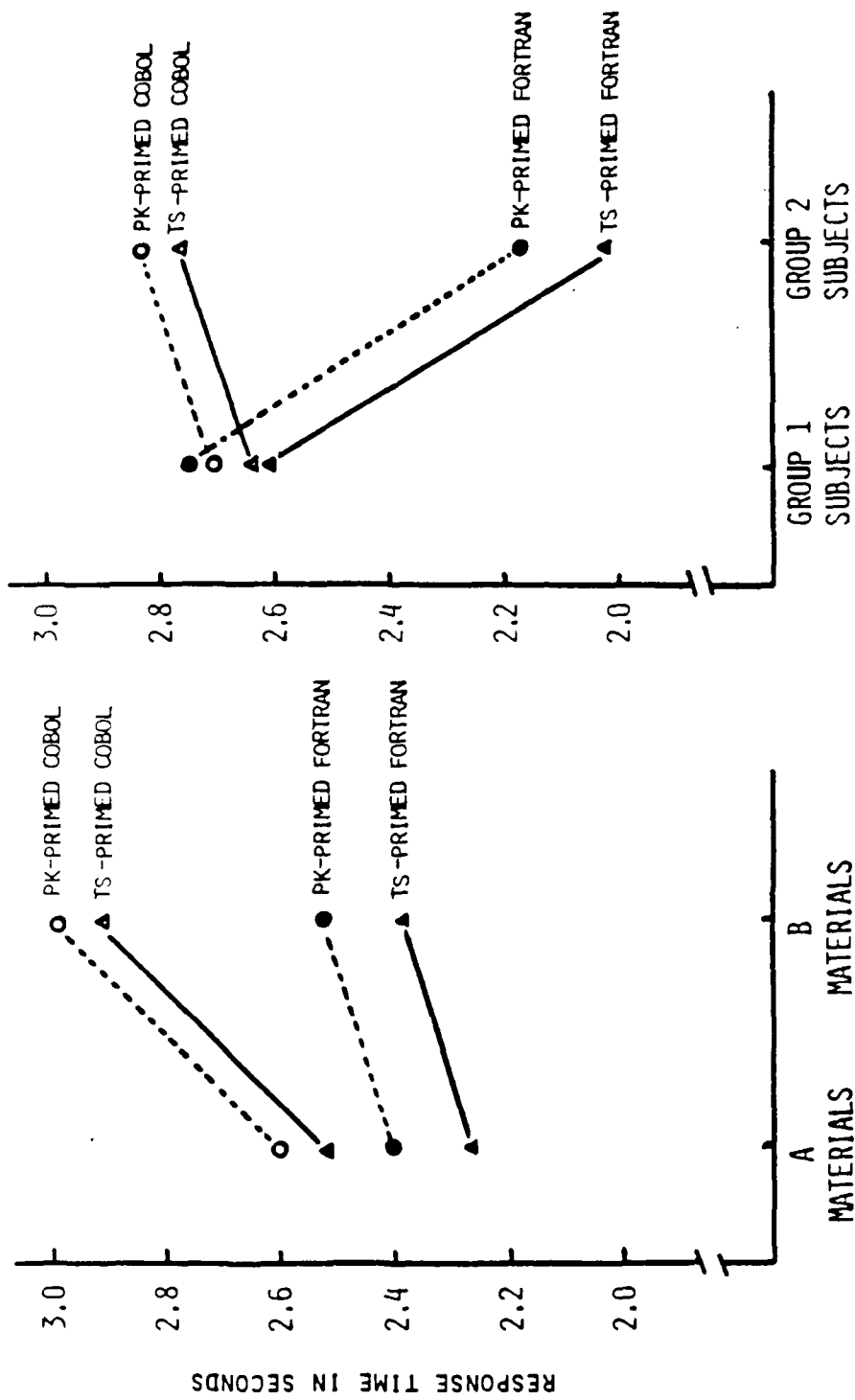
17

RESPONSE TIME IN SECONDS

A. RESPONSE TIMES ADJUSTED FOR
EFFECTS OF SUBJECT GROUP

B. RESPONSE TIMES ADJUSTED FOR
EFFECTS OF MATERIALS SETS

Figure 7

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 less than 1). Accuracy for target recognition items across subjects and items averaged 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 less than 1). Correct responses averaged 2.670 seconds compared to incorrect responses, which averaged 3.667 seconds, $F(1,76) = 63.60$, $p < .001$, suggesting a difficulty relationship between speed and accuracy rather than a speed/accuracy tradeoff. 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 (Figures 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 (Figure 6.A) 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 (Figure 6.B) 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 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 (percent of items missed in the category) and for each item (percent of subjects missing an item). The five scores per subject were submitted to a language (FORTRAN, COBOL) by 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 by 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 most accurately (15%, 21% errors respectively) while more errors were made on data flow, state, and function items (28%, 30%, 34% errors respectively).

18

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), $F(4,86) - .827$ (not significant, items analysis). The comprehension pattern across information categories for FORTRAN subjects (see Figure 8) was: 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 (Figure 8). The most noticeable difference 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.

****
Insert Figure 8 about here
****

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 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, $F(3,76) - .64$; 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, $F(1,78) - .96$.

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 Figure 9.
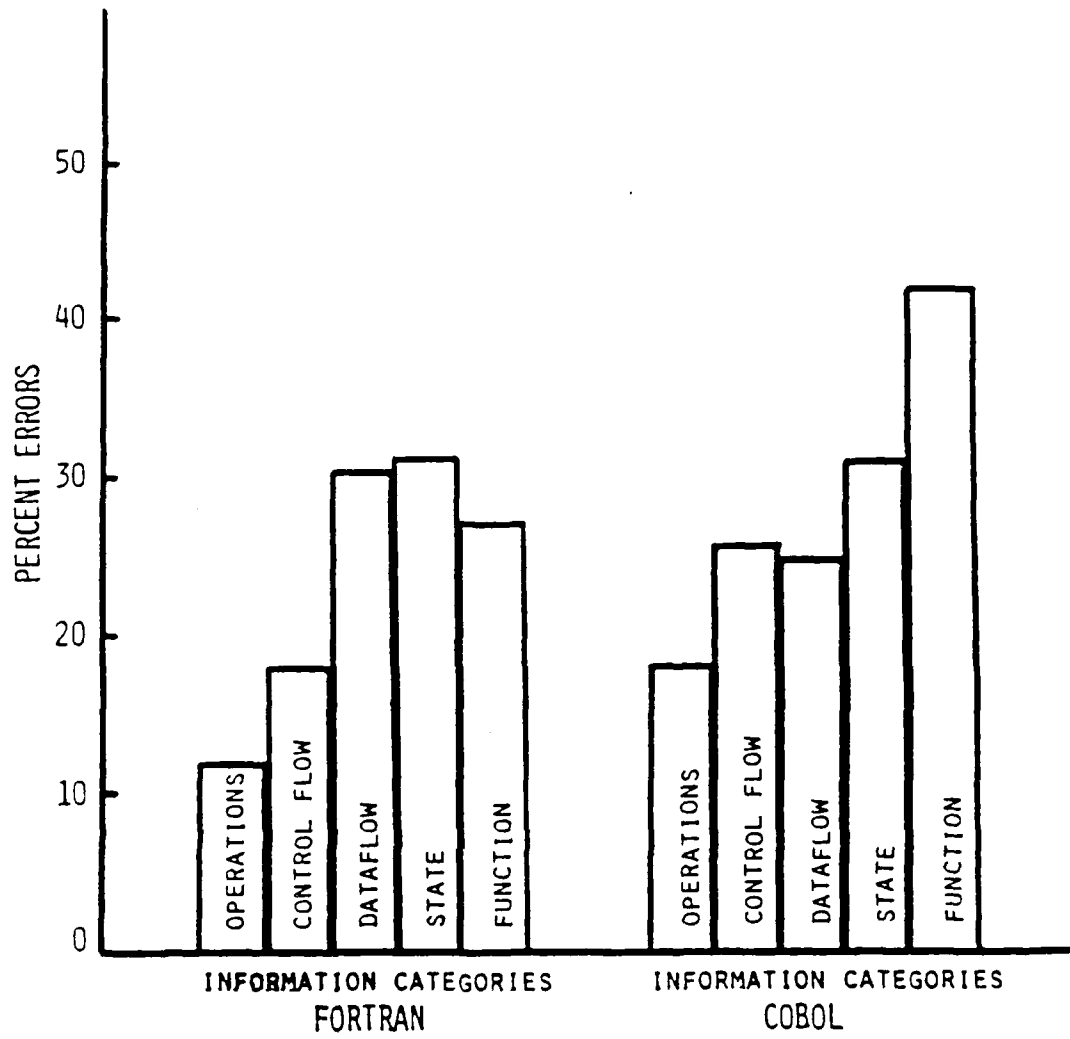
19

Figure 8

Consistent with the stage of comprehension explanation, bottom quartile subjects 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 by 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 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, $F(4,86) = .619$.

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 (Figure 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 (Figure 6.A). 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 (Figure 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.
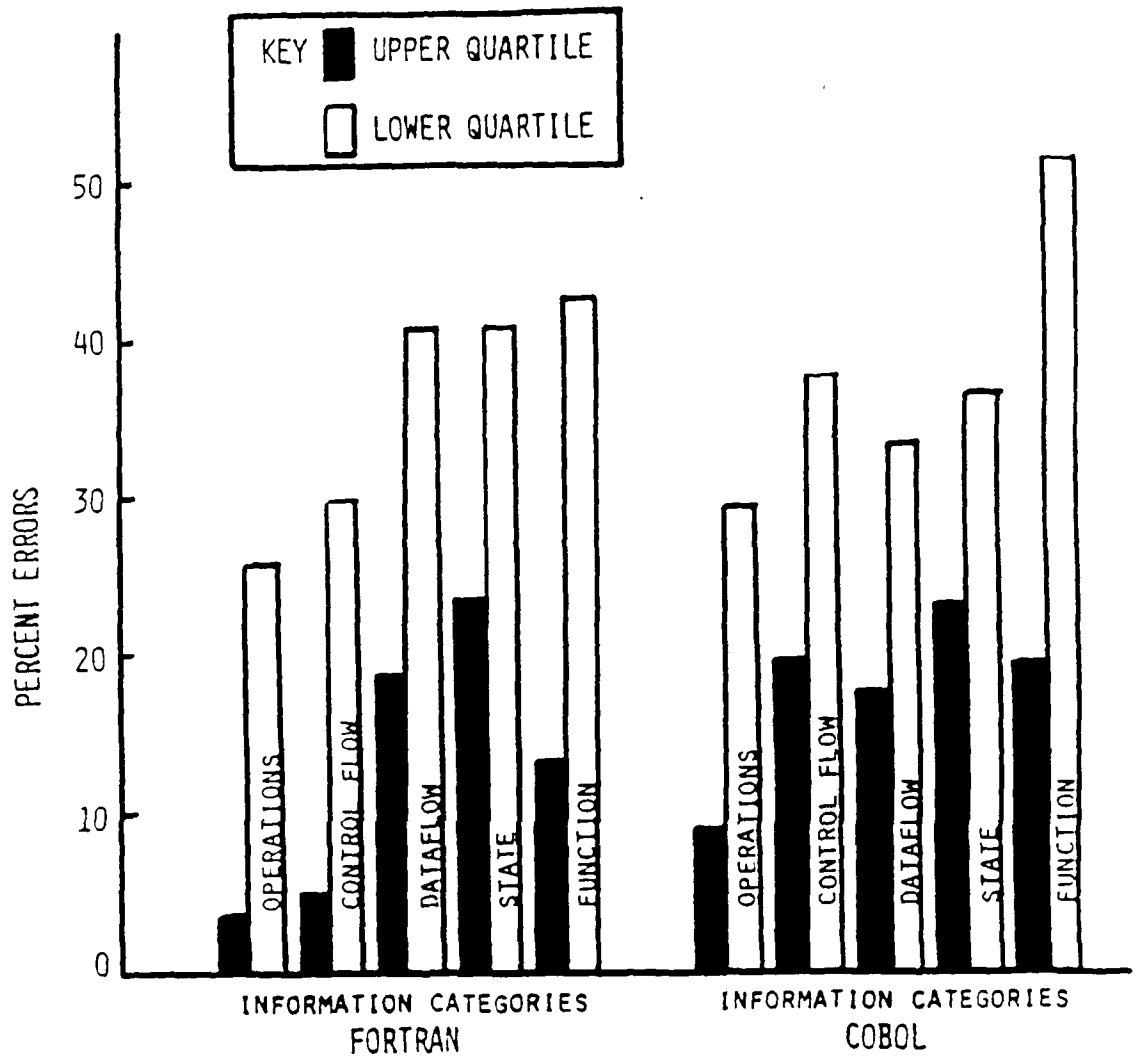
20

Figure 9

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 percent errors, fastest response times). (2) Control flow inferences are thus easy to retrieve or compute (low percent 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 percent errors). (4) Program state inferences are difficult to compute (long response times, high percent 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 (Figures 2 through 5), relations captured by the procedural, control flow abstraction (Figure 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 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 the text surface distance between the statements, by syntactic similarity between statements, or by argument repetition since these features were controlled by counter-balancing 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

21

might argue on the basis of the error data alone that there is a priority f function information in top comprehenders' memory representations, the prim 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 zeros are assigned to variables may be designated "initialization of variables" (see Figure 6.A), 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 Figure 6.A, 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 at 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 test in COBOL corresponds less well to execution sequence than in FORTRAN. This can be seen

22

by comparing Figure 1.A (program text) and Figure 5 A (T. analysis) in which
the loop that sums a list of numbers is executed in statement 9 but is
specified in statements 11 through 15. In the FORTRAN version of this
segment, the loop occurs in statements 5 through 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 TWO

In the first study, programmers' comprehension strategies may have been
influenced by several aspects of the experimental task: short undocumented
program segments, the series of short study trials, and the demands of memory
questions. In Study Two, 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 Two program text.

23

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. Subjects were run over a period of six months from September 1984 to February 1985. Each programmer was paid a $50.00 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 non-trivial 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 forty questions were divided into two matching lists of twenty 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 hours 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

24

subjects' task then was 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 minutes 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 minutes. 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 minute 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 minutes was specified. Subjects were told that they should begin actual modifications at any point when they felt ready. If necessary the full 30 minutes 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 minute 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) x 2 (Q1, Q4 comprehension quartiles) x 2 (Talk, Notalk Conditions) x 2 (comprehension test lists after study, after modification) x 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 minute study phase, are most directly comparable to the conditions of comprehension testing in Study One. For this

25

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 minutes 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 One, 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 Figure 10). This pattern across information categories did not differ reliably by language, quartile or talk-aloud condition (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 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.

****
Insert Figure 10 about here
****

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 one of the program. Phase two 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
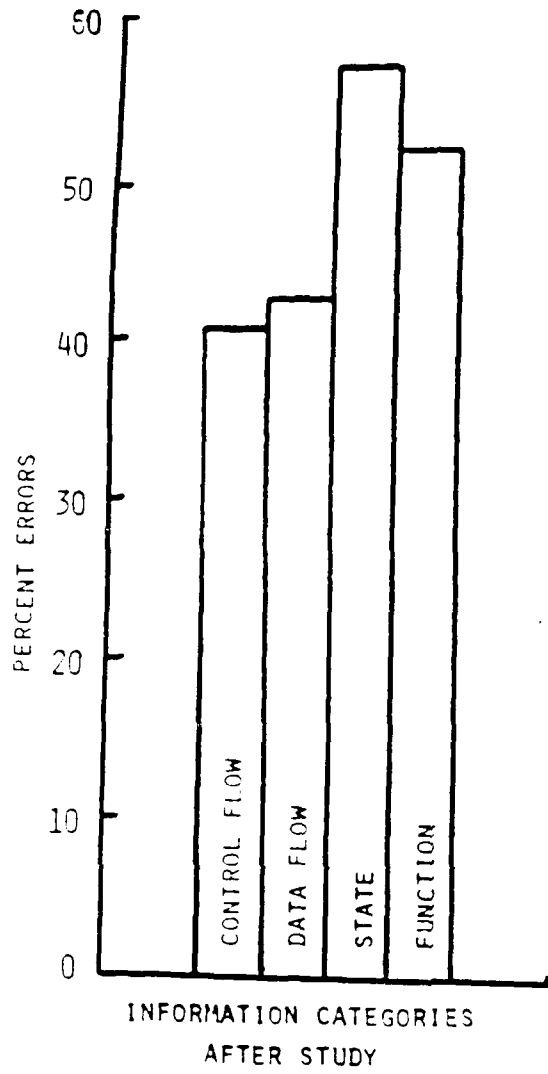
26

Figure 10

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, $F_{(3,96)} = 9.??$,
$p < .001$ (comprehension trial by information category interaction in an ANOVA
treating comprehension trial as a repeated measure). The pattern of errors
for this trial (see Figure 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
Figure 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.

```
****************************
Insert Figure 11 about here
****************************
```

Discussion

Comprehension results from Study Two (Figure 10) reinforce and extend the conclusions from Study One 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 study 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 Two a different comprehension pattern emerged after a second exposure to the program during which programmers completed a program modification (Figure 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 Two 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
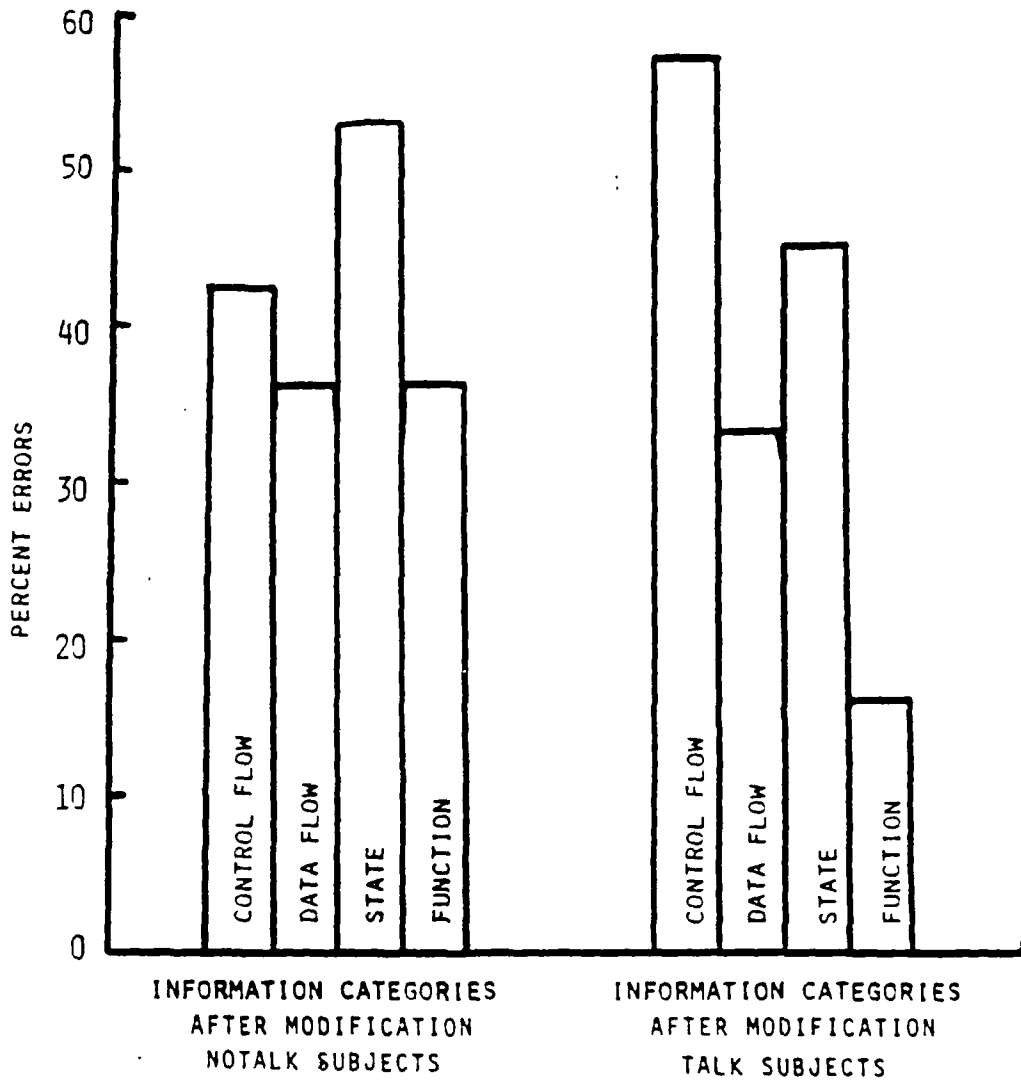
28

Figure 11

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 Two deserves comment. Comprehension quartile as determined by comprehension scores in the experimental setting of Study One predicted the comprehension scores in the more natural task of Study Two. However, the error rates on comprehension questions for _both_ upper and lower quartile comprehenders were quite high in Study Two, even after 1.25 hours 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 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 different 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 (Figure 2), a data flow abstraction highlighting the transformations that are applied to data objects (Figure 3), a control flow abstraction highlighting the temporal sequence of execution of program actions (Figure 4), and a conditionalized action representation specifying states of the program and the actions invoked (Figure 5). Although this specific 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 task 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

29

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; Rumelhart, 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 (Figure 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.

30

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 and 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 University 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 Applications, 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 and 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. 'ternberg (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.

31

Dahl, O. J., Dijkstra, E., & Hoare, C. A. R. (1972). Structured programming 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., & Garret, 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 and T. 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: 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, and 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 University 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 and 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. (1982). Cognitive components of expertise in computer programming: A review of the literature. _Psychological Documents_, Ms. No. 4170.

Pennington, N. & Grabowski, B. (1986). _Cognitive components of expertise in computer programming: A conceptual framework_. Unpublished manuscript, University of Chicago, Chicago, IL.

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: The MIT Press.

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

Rumelhart, D. E. (1975). Notes on a schema for stories. In D. G. Bobrow and

33

A. Collins (Eds.), *Representation and understanding: Studies in cognitive science*. New York: Academic Press.

Rumelhart, 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 Publishers, Inc.

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, Arizona.

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, Maryland.

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.

Footnotes

[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.

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

[3]Data from 6 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).

[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 seconds 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 milliseconds and reduced variability. For example the "cleaned"
average response time for correct "yes" items was 2.512 seconds 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.

[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.

[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.

35

^{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.

List of Figures

36

Table 1
Correspondences Between Text Abstractions,
Knowledge Structures, and Mental Representations

TEXT RELATIONS	KNOWLEDGE STRUCTURES	MENTAL REPRESENTATION
Control Flow	Text Structure	Procedural Episodes
Function Data Flow	Plan Knowledge	Functional Representation
Condition-Action	Unknown	Unknown

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

A. All subjects, response time in seconds

Subject Group Within Language	FORTRAN SUBJECTS AND MATERIALS		COBOL SUBJECTS AND MATERIALS	
	PK Prime	TS Prime	PK Prime	TS Prime
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

Subject Group Within Language	FORTRAN SUBJECTS AND MATERIALS		COBOL SUBJECTS AND MATERIALS	
	PK Prime	TS Prime	PK Prime	TS Prime
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

38

Dr. Beth Adelson
Department of Computer Science
Tufts University
Medford, MA 02155

Dr. Robert Ahlers
Code N711
Human Factors Laboratory
Naval Training Systems Center
Orlando, FL 32813

Dr. Ed Aiken
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. John Black
Teachers College
Columbia University
525 West 121st Street
New York, NY 10027

Dr. Deborah A. Boehm-Davis
Department of Psychology
George Mason University
4400 University Drive
Fairfax, VA 22030

Dr. Jeff Bonar
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Maj. Hugh Burns
AFHRL/IDE
Lowry AFB, CO 80230-5000

Dr. John M. Carroll
IBM Watson Research Center
User Interface Institute
P.O. Box 218
Yorktown Heights, NY 10598

Dr. Fred Chang
Navy Personnel R&D Center
Code 51
San Diego, CA 92152-6800

Dr. Davida Charney
English Department
Penn State University
University Park, PA 16802

Dr. L. J. Chmura
Computer Science and
  Systems Branch
Naval Research Laboratory
Washington, DC 20375-5000

Defense Technical
  Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC
12 Copies

ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda, MD 20014

Mr. Wallace Feurzeig
Educational Technology
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

Dr. Michael Friendly
Psychology Department
York University
Toronto ONT
CANADA   M3J 1P3

Dr. Sherrie Gott
AFHRL/MODJ
Brooks AFB, TX 78235

Dr. James G. Greeno
University of California
Berkeley, CA 94720

Dr. Henry M. Halff
Halff Resources, Inc.
4918 33rd Road, North
Arlington, VA 22207

Dr. Bruce Hamill
The Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20707

Dr. Jim Hollan
Intelligent Systems Group
Institute for
 Cognitive Science C-015
 [UCSD]
La Jolla, CA 92093

Dr. Ed Hutchins
Intelligent Systems Group
Institute for
 Cognitive Science C-015
 [UCSD]
La Jolla, CA 92093

Dr. Peter Pirolli
Hewlett-Packard Laboratories
P.O. Box 10490
Palo Alto, CA 94303-0971

Dr. Wendy Kellogg
IBM T. J. Watson Research Ctr.
P.O. Box 218
Yorktown Heights, NY 10598

Dr. David Kieras
University of Michigan
Technical Communication
College of Engineering
1223 E. Engineering Building
Ann Arbor, MI 48109

Dr. Clayton Lewis
University of Colorado
Department of Computer Science
Campus Box 430
Boulder, CO 80309

Dr. Marcia C. Linn
Lawrence Hall of Science
University of California
Berkeley, CA 94720

Dr. Stuart Macmillan
FMC Corporation
Central Engineering Labs
1185 Coleman Avenue, Box 580
Santa Clara, CA 95052

Dr. James McMichael
Assistant for MPT Research,
 Development, and Studies
 OP 01B7
Washington, DC 20370

Dr. James R. Miller
MCC
9430 Research Blvd
Echelon Building #1, Suite 231
Austin, TX 78759

Dr. Lance Miller
IBM-FSD Headquarters
6600 Rockledge Drive
Bethesda, MD 20817

Dr. William Montague
NPRDC Code 13
San Diego, CA 92152-6800

Dr. Allen Newell
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. A. F. Norcio
Computer Science and
 Systems Branch
Naval Research Laboratory
Washington, DC 20375-5000

Dr. Donald A. Norman
Institute for Cognitive Science
University of California
La Jolla, CA 92093

Library, NPRDC
Code P201L
San Diego, CA 92152-6800

Dr. Judith Reitman Olson
School of Business
 Administration
University of Michigan
Ann Arbor, MI 48109

Dr. Peter Polson
University of Colorado
Department of Psychology
Boulder, CO 80309

Dr. Mary S. Riley
Program in Cognitive Science
Center for Human Information
    Processing
University of California
La Jolla, CA 92093

Dr. Marc Sebrechts
Department of Psychology
Wesleyan University
Middletown, CT 06475

Dr. Sylvia A. S. Shafto
Department of
    Computer Science
Towson State University
Towson, MD   21204

Dr. Elliot Soloway
Yale University
Computer Science Department
P.O. Box 2158
New Haven, CT 06520

Chair, Department of
    Computer Science
University of Maryland,
    College Park
College Park, MD   20742

Dr. Ralph Wachter
JHU-APL
Johns Hopkins Road
Laurel, MD   20707

Dr. Keith T. Wescourt
FMC Corporation
Central Engineering Labs
1185 Coleman Ave., Box 580
Santa Clara, CA 95052

Dr. Barbara White
Bolt Beranek & Newman, Inc.
10 Moulton Street
Cambridge, MA 02238

Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Joe Yasatuke
AFHRL/LRT
Lowry AFB, CO 80230

Dr. Joseph L. Young
Memory & Cognitive
    Processes
National Science Foundation
Washington, DC 20550

ATE
MED
28