Empirical Studies of Programming Knowledge

ELLIOT SOLOWAY AND KATE EHRLICH

Abstract—**We suggest that expert programmers have and use two types of programming knowledge: 1) *programming plans*, which are generic program fragments that represent stereotypic action sequences in programming, and 2) *rules of programming discourse*, which capture the conventions in programming and govern the composition of the plans into programs. We report here on two empirical studies that attempt to evaluate the above hypothesis. Results from these studies do in fact support our claim.**

Index Terms—**Cognitive models of programming, novice/expert differences, program conprehension, software psychology.**

I. Introduction: Motivation and Goals

WHAT is it that expert programmers know that novice programmers don't? We would suggest that the former have *at least* two types of knowledge that the latter typically do not.

- *Programming Plans:* Program fragments that represent stereotypic action sequences in programming, e.g., a RUNNING TOTAL LOOP PLAN, an ITEM SEARCH LOOP PLAN [16].
- *Rules of Programming Discourse:* Rules that specify the conventions in programming, e.g., the name of a variable should usually agree with its function; these rules set up expectations in the minds of the programmers about what should be in the program. These rules are analogous to discourse rules in conversation.

In our view, programs are composed from programming plans that have been modified to fit the needs of the specific problem. The composition of those plans are governed by rules of programming discourse. Thus, a program can be correct from the perspective of the problem, but be difficult to write and/or read because it doesn't follow the rules of discourse, i.e., the plans in the program are composed in ways that violate some discourse rule(s).

Our approach to programming borrows directly from at least two, converging sources: the research in text processing in artificial intelligence and psychology, and the research in problem solving with experts and novices. First, we base our claim that text comprehension research is appropriate to the task of understanding program comprehension on the following observation: programs have a dual nature—they can be *executed* for effect, and they can be *read* as communicative entities. Viewed in this light, we felt that the notion of *schemas*, one of the most influential notions to have emerged from recent research on text comprehension (e.g., [2], [3], [10], [16]) should be applicable to program comprehension.

"Schemas are generic knowledge structures that guide the comprehender's interpretations, inferences, expectations, and attention when passages are comprehended" [10].

Our notion of programming plan corresponds directly to this notion of schema.

Second, research with experts and novices in various technical domains (chess [4], [7], physics [14], electronic circuitry [8]) have shown that the former seem to develop *chunks* that represent functional units in their respective domains, while the latter do not. Similar results have been obtained in the programming domain [1], [16], [18]. The work reported in this paper builds on and extends the above research in the programming domain by examining whether or not programmers have and use specific programming plans and rules of programming discourse in the process of comprehending computer programs. Moreover, the work reported here extends our earlier studies on programming plans [9], [19] by presenting a broader, more systematic empirical examination of these concepts. Note too that this work is another example of our efforts to explore the cognitive underpinnings of programming: while in Soloway *et al.* [20] we examined the cognitive fit of a particular programming language construct (Pascal's WHILE loop) to people's natural cognitive strategies, here we examine the role that various types of programming knowledge play in the comprehension of programs.

In this paper we describe two empirical studies we conducted with programmers of varying levels of expertise. The goal of these studies was to evaluate the above claim: do expert programmers possess programming plans and discourse rules? Programs that do not conform to the plans and discourse rules should violate the programmers' expectations: for example, if they see a variable initialized to zero (N := 0) at the top of a program, they should be surprised to see it being changed via a read statement (READ(N)) later in the program. While this type of situation will not create an unrunnable program, it certainly violates the accepted conventions of programming: 1) variables are usually updated in the same fashion as they are initialized; thus we would expect N to be updated via an assignment statement, and 2) programmers do not like to include statements that have no effect: a READ statement destroys whatever is in the variable initially, and thus the initial setting of N to zero is superfluous. We claim that these violations in expectations—the surprises due to violations of conventions—can make such programs much more difficult to comprehend.

Manuscript received August 1, 1983; revised May 14, 1984. This work was supported in part by the National Science Foundation, under NSF Grants MCS-8302382 and IST-8310659, and by a contribution from the Burroughs Corporation. However, the views expressed in this paper do not necessarily reflect those of the Burroughs Corporation. Portions of this paper appear in the *Proceedings of the Conference on the Nature of Expertise*, Carnegie-Mellon University, Pittsburgh, PA, October 1983.

E. Soloway is with the Department of Computer Science, Yale University, New Haven, CT 06520, and Compu-Tech, Inc., New Haven, CT.

K. Ehrlich is with Honeywell Information Systems, Inc. Waltham, MA 02154.

Thus, if advanced programmers have knowledge about plans and discourse rules then programs that do not conform to the rules of discourse (*unplan-like programs*) should be harder for them to understand than programs that do conform to these rules (*plan-like programs*). In contrast, we would not expect novice programmers to have acquired as many of the plans and conventions in programming: by definition a novice programmer has less knowledge than an advanced programmer. Thus, we would not expect novice programmers to be as sensitive to violations of conventions—since they don't know what the conventions are in the first place. Therefore, in a task that requires understanding a program, we expect 1) advanced programmers to do much better than the novice programmers on the programs that do conform to the plans and rules, while we expect 2) advanced programmers to perform at the level of novices when the programs violate the plans and the discourse rules.[1]

The organization of this paper is as follows. First, we present a brief description of our experimental studies; this section provides the needed motivation for why our "stimulus materials"—the computer programs used in our experiments—were constructed in the manner that they were. Next, we present a detailed description of how and why unplan-like programs can be generated from plan-like ones. In the following two major sections, we present detailed descriptions of each of our empirical studies, along with a discussion of the results from the studies. We close with implications from these studies for the practice of programming.

II. Brief Description of Both Experimental Techniques

The first stage in both experimental procedures is as follows. First, we construct a plan-like program, i.e., one that uses only typical programming plans and whose plans are composed so as to be consistent with rules of programming discourse. Next, we construct an unplan-like version of that program by violating one (or possibly two) of the discourse rules. We will refer to the plan-like version of the program as the Alpha version, while the unplan-like version will be referred to as the Beta version.[2] An example of an Alpha version with a Beta version for a programming problem is given in Fig. 1. (In Section III we describe in detail how these programs were constructed, and why we would consider the Beta version to be unplan-like.)

A. Brief Description of Study I: Fill-in-the-Blank

Our first study uses a "fill-in-the-blank technique": here we take out one line of code from the program and replace that line with a blank. The task we ask of our experimental subjects, who are novice and advanced student programmers, is to fill the blank line in with a piece of code that, in their opinion, best completes the program. An example of the programs with blank lines is given in Fig. 1. Note carefully that we do *not* tell the subjects what problem the program is intended to solve. However, since there is only one blank line per program, a great deal of context is still left. If advanced programmers do have and use programming plans for stereotypic programming situations, then they should be able to recognize the program fragment in the plan-like versions as an example of programming plan *X*, and they should all fill in the blank line with the same piece of code. However, in the case of the unplan-like programs, advanced programmers should be more unsure of what plan is being indicated; thus, they should be less likely to complete the program in the correct fashion. On the other hand, novice programmers should not be as surprised by the unplan-like programs since they have not as yet acquired the programming conventions. Thus, we expect that the advanced programmers will be more affected by the unplan-like programs than will the novices.

Notice that both the Alpha version and the Beta version are runnable programs that in almost all cases compute the *same* values.[3] Moreover, to an untrained eye their differences may even not be apparent: they always only differ by a very few textual elements. Thus, our experimental materials are not random programs, as were used in previous studies [1], [16], [18]. While those studies demonstrated the basic effect—that advanced programmers have strategies for encoding and remembering programs better than do novice programmers—we see our work as focusing on the *detailed knowledge* that programmers have and use.

B. Brief Description of Study II: Recall

In our second study, we used essentially the same stimulus materials as in Study I. This time, however, the task was a recall one and all subjects were expert professional programmers. Subjects were presented with a complete program which they were asked to recall *verbatim*. Half the programs were plan-like and half were unplan-like. Each program was presented three times. On the first trial, subjects were asked to recall as much of the program as possible. On the second and third trials, they were asked to either add to their original recall or to change any part of their recall that they felt was in error. We tracked the progression of their recall by asking them to use a different color pencil on each of the three trials. This technique, of repeated presentation of the same program, was developed by Kahney [12] for research specifically on the comprehension of computer programs. If programming plans help programmers to encode a program more efficiently we should find that experts recall more of the program earlier. However, given sufficient time, they should be able to recall as much of the unplan-like programs as the plan-like ones. Again, while others have shown this basic effect our motivation is to identify specific knowledge units and to demonstrate the *significant* influence that planliness has on program comprehension: a change in just a few characters can result in significant differences in performance!

[1]In the second study we only used expert professional programmers as subjects, and thus we can not look for this type of interaction. Rather, we simply want to 1) evaluate our hypothesis with professional programmers, and 2) observe whether there is a difference in performance within the experts on programs that vary along the *plan-like* dimension.

[2]Clearly, the Beta versions are not totally unplan-like; in fact, they have many plans in common with the Alpha versions. The term "unplan-like" is thus meant for emphasis only.

[3]In only one program type, MAX (e.g., Fig. 1), do the Alpha and Beta versions compute different values.

Version Alpha

```
PROGRAM Magenta(input, output),
VAR Max, I, Num   INTEGER,
BEGIN
    Max  = 0,
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Num),
          If Num  >  Max THEN Max  = Num
        END,
    WRITELN(Max),
END
```

```
PROGRAM Magenta(input, output),
VAR Max, I, Num   INTEGER,
BEGIN
    Max  = 0,
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Num),
                    -------
          If Num    |     |  Max THEN Max  = Num
                    |     |
                    -------
        END,
    WRITELN(Max),
END
```

Version Beta

```
PROGRAM Purple(input, output),
VAR Max, I, Num   INTEGER,
BEGIN
    Max  = 999999,
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Num),
          If Num  <   Max THEN Max  = Num
        END,
    WRITELN(Max),
END
```

```
PROGRAM Purple(input, output),
VAR Max, I, Num   INTEGER,
BEGIN
    Max  = 999999,
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Num),
                    -------
          If Num  |      | Max THEN Max  = Num
                  |      |
                    -------
        END,
    WRITELN(Max),
END
```

Program type 1

Basic plan	search plan (max, min)
Discourse rule	A variable's name should reflect its function (1)
How construct Beta version	violate discourse rule (1)
Alpha case	variable name agrees with search function
Beta case	variable name does NOT agree with search function

Fig. 1. Example: Program type 1.

III. Generating Plan-Like and Unplan-Like Programs

What makes a program plan-like rather than unplan-like is the way in which plans are *composed* in a program. The composition is governed by *rules of programming discourse*, which are analogous to discourse rules in ordinary conversation or discourse rules that govern the structure of stories. In Fig. 2 we depict a set of programming discourse rules that we have identified. Individually, they look innocuous enough, and one could hardly disagree with them. While these rules typically are not written down nor taught explicitly, we claim that programmers have and use these rules in the construction and comprehension of programs. If programmers do use these rules

(1) Variable names should reflect function.

(2) Don't include code that won't be used.

(2a) If there is a test for a condition, then the condition must have the potential of being true.

(3) A variable that is initialized via an assignment statement should be updated via an assignment statement).

(4) Don't do double duty with code in a non-obvious way.

(5) An IF should be used when a statement body is guaranteed to be executed only once, and a WHILE used when a statement body may need to be repeatedly executed.

Fig. 2. Rules of programming discourse.

and expect other programmers to also use these rules, then we would predict that programs that violate these rules should be harder to understand than programs that do not.

One key point to notice in the following sections is that the unplan-like version (the Beta version) is *only slightly* different than the plan-like version (the Alpha version). That is, the idea is to take a plan-like program and modify it ever so slightly, by violating a discourse rule, so as to create an unplan-like version. Both versions are executable programs that usually compute the same function. Moreover, both versions have about the same surface characteristics: about the same number of lines of code, about the same number of operands and operations, etc. Thus, while more traditional methods of calculating program complexity (e.g., lines of code, or Halstead metrics [11]) would predict no difference in the difficulty of understanding for the two programs (the Alpha version and the Beta version), we are looking for differences in actual performance on an understanding task.

In the following sections we will describe how and why we constructed the plan-like and unplan-like programs for use in our empirical studies. In each of the next four sections we will describe a different pair of programs, where the Beta version of the pair is generated by violating one (or possibly two) of the discourse rules given in Fig. 2. We will refer to a pair of programs as exemplifying a program type; thus, we will describe four different program types:[4] 1) MAX, 2) SQRT, 3) AVERAGE, and 4) IF/WHILE.

A. Program Type 1: MAX

In Fig. 1, version Alpha is the plan-like version of a program that finds the maximum of some numbers. In our plan jargon, it uses the MAXIMUM SEARCH LOOP PLAN which in turns uses a RESULT VARIABLE PLAN. Notice that the RESULT VARIABLE is appropriately named Max, i.e., the name of the variable is consistent with the plan's function. In contrast version Beta is unplan-like since it uses a MINIMUM SEARCH LOOP PLAN in which the RESULT VARIABLE is inconsistent with the plan's function: the program computes the minimum of some numbers using a variable name Max. To create the Beta version, we violated the first rule of programming discourse in Fig. 2: *variable names should reflect function.* (See also, Weissman [22], who did exploratory empirical studies on the role of variable names.)

The fill-in-the-blank versions of both these programs are also given in Fig. 1. Our hypothesis is that programmers will see the variable name Max and thus "see" the program as a MAXIMUM SEARCH LOOP PLAN. In other words, the name of the variable will color how they understand the rest of the program. Therefore, in the Beta version, where the function of the procedure is inconsistent with variable Max, we predict that programmers will fill in the blank with a >, rather than a <–indicating that they see the program as computing the maximum of a set of integers, instead of the minimum.

B. Program Type 2: SQRT

The Alpha and Beta programs in Fig. 3 are both intended to produce the square root of N. Since N is in a loop which will repeat 10 times, 10 values will be printed out. The question is: How should N be set? In version Alpha the DATA GUARD PLAN constrains what should be filled into the blank line. That is, the Sqrt function must be protected from trying to take the Sqrt of a negative number; thus, the immediately preceding IF test checks to see if the number is negative, and makes it positive if necessary. Besides protecting the Sqrt function, the DATA GUARD PLAN exerts influence on what could reasonably be filled into the blank. The very presence of the DATA GUARD PLAN implies that the numbers might be negative and thus the manner in which N is set *must allow for it to be negative.* A typical way of realizing this constraint is via a Read(N); the user decides what values should be entered. In contrast, setting N via an assignment statement, e.g., N := I, would *never* result in a negative number–thus making the DATA GUARD PLAN totally superfluous. The influence of the DATA GUARD PLAN over the blank line stems from a rule of programming discourse: *if there is a test for a condition, then the condition must have the potential of being true.* Thus, the blank line must be filled in with something that does not make the DATA GUARD PLAN superfluous, e.g., Read(N).

In version Beta, however, we have added an additional constraint on the blank line: the VARIABLE PLAN for N starts off with an assignment type of initialization (N := 0) and sets up the expectation that N will also be updated via an assignment statement, e.g., N := N + I, or N := N + 1. However, this expectation conflicts with the expectation set up by the DATA GUARD PLAN [namely, Read(N)]. Moreover, there is an additional level of conflict: the expectation of the DATA GUARD PLAN is now in conflict with the initialization of N to 0. This latter conflict is due to a violation of the following rule of programming discourse: *a variable that is initialized via an assignment statement should be updated via an assignment statement.*

C. Program Type 3: AVERAGE

The programs in Fig. 4 calculate the average of some numbers that are read in; the stopping condition is the reading of the sentinel value, 99999. Version Alpha accomplishes the task in a typical fashion: variables are initialized to 0, a read-a-value/process-a-value loop [20] is used to accumulate the running total, and the average is calculated after the sentinel has been read. Version Beta was generated from version Alpha by violating another rule of programming discourse: *do not do double duty in a nonobvious manner.* That is, in version Beta, unlike in Alpha, the initialization actions of the COUNTER VARIABLE (Count) and RUNNING TOTAL VARIABLE PLANs (Sum) in Beta serve two purposes:

- Sum and Count are given initial values.
- The initial values are chosen so as to compensate for the fact that the loop is poorly constructed and will result in an off-by-one bug: the final sentinel value (99999) will be incorrectly added into the RUNNING TOTAL VARIABLE, Sum, and the COUNTER VARIABLE, Count, will also be incorrectly updated.

We felt that using Sum and Count in this way was most nonobvious, and would prove very hard for advanced programmers to comprehend.

D. Program Type 4: IF/WHILE

The difference between an IF statement and a WHILE statement in Pascal is that the latter executes a body of statements

[4]The names given to each of the four types carry no deep significance: they are meant only to aid the reader.

Version Alpha

```
PROGRAM Beige(input, output),
  VAR Num   REAL,
      I   INTEGER,
  BEGIN
     FOR I  = 1 TO 10 DO
        BEGIN
          Read (Num),
          IF Num < 0 THEN Num  = -Num,
          Writeln ( Num, Sqrt(Num) ),
             (* Sqrt is a built-in
               function which returns the
               square root of its argument*)
        END,
  END
```

```
PROGRAM Beige(input, output),
  VAR Num   REAL,
      I   INTEGER,
  BEGIN
     FOR I  = 1 TO 10 DO
        BEGIN
          -------------------------------
          |                             |
          |                             |
          -------------------------------
          IF Num < 0 THEN Num  = -Num,
          Writeln ( Num, Sqrt(Num) ),
             (* Sqrt is a built-in
               function which returns the
               square root of its argument*)
        END,
  END
```

Version Beta

```
PROGRAM Violet(input, output),
  VAR Num   REAL,
      I   INTEGER,
  BEGIN
     Num  = 0,
     FOR I  = 1 TO 10 DO
        BEGIN
          Read (Num),
          IF Num < 0 THEN Num  = -Num,
          Writeln ( Num, Sqrt(Num) ),
             (* Sqrt is a built-in
               function which returns the
               square root of its argument*)
        END,
  END
```

```
PROGRAM Violet(input, output),
  VAR Num   REAL,
      I   INTEGER,
  BEGIN
     Num  = 0,
     FOR I  = 1 TO 10 DO
        BEGIN
          -------------------------------
          |                             |
          |                             |
          -------------------------------
          IF Num < 0 THEN Num  = -Num,
          Writeln ( Num, Sqrt(Num) ),
             (* Sqrt is a built-in
               function which returns the
               square root of its argument*)
        END,
  END
```

Program type 2

Basic plan	guard plan, variable plan
Discourse rule	Don't include code that won't be used (2) If there is a test for a condition, then the condition must have the potential of being true (2a) A variable that is initialized via an assignment statement should be updated via an assignment statement (3)
How construct Beta version	include two incompatible discourse rules (2) and (3)
Alpha case	guard plan predicts read initialization
Beta case	guard plan predicts read update, but initialization plan predicts assignment update

Fig. 3. Example: Program type 2.

repeatedly, while the former only executes the body once; note both have a testing component. In looking at programs written by novice programmers, we found that novices sometimes used a WHILE statement when the body would only be executed once: it was as if novices have a rule such as *when a body needs to be executed only once*, then *either a* WHILE *or an* IF *could be used.* We felt that advanced programmers would be horrified by such a rule, and, moreover, would be confused in seeing a WHILE in a situation that "clearly" called for an IF.

The programs in Fig. 5 were developed to test the above hypothesis. Both these programs test to see if some variable contains a number that is greater than a maximum, and if so, the variable is reset to the maximum. The Alpha version uses an IF test; the Beta version uses a WHILE statement. The Beta

Version Alpha

```
PROGRAM Grey(input, output),
VAR Sum, Count, Num  INTEGER,
    Average  REAL,
BEGIN
    Sum  = 0,
    Count  = 0,
    REPEAT
        READLN(Num),
        IF Num <> 99999 THEN
                        BEGIN
                           Sum  = Sum + Num,
                           Count  = Count + 1,
                        END,
      UNTIL Num = 99999,
    Average  = Sum/Count,
    WRITELN(Average),
END
```

```
PROGRAM Grey(input, output),
VAR Sum, Count, Num  INTEGER,
    Average  REAL,
BEGIN
    Sum  = 0,
    -------------------------------
    |                             |
    |                             |
    -------------------------------
    REPEAT
        READLN(Num),
        IF Num <> 99999 THEN
                        BEGIN
                           Sum  = Sum + Num,
                           Count  = Count + 1,
                        END,
      UNTIL Num = 99999,
    Average  = Sum/Count,
    WRITELN(Average),
END
```

Version Beta

```
PROGRAM Orange(input, output),
VAR Sum, Count, Num  INTEGER,
    Average  REAL,
BEGIN
    Sum  = -99999,
    Count  = -1,
    REPEAT
        READLN(Num),
        Sum  = Sum + Num,
        Count  = Count + 1,
      UNTIL Num = 99999,
    Average  = Sum/Count,
    WRITELN(Average),
END
```

```
PROGRAM Orange(input, output),
VAR Sum, Count, Num  INTEGER,
    Average  REAL,
BEGIN
    Sum  = -99999,
    -------------------------------
    |                             |
    |                             |
    -------------------------------
    REPEAT
        READLN(Num),
        Sum  = Sum + Num,
        Count  = Count + 1,
      UNTIL Num = 99999,
    Average  = Sum/Count,
    WRITELN(Average),
END
```

Program type 3

Basic plan	read/process, running total loop plan
Discourse rule	don't do double duty in a non-obvious way (4)
How construct Beta version	violate discourse rule (4)
Alpha case	initialize to standard values
Beta case	initialize to non-standard values to compensate for poorly formed loop

Fig. 4. Example: Program type 3.

version was generated from the Alpha version by violating the following discourse rule: *An* IF *should be used when a statement body is guaranteed to be executed only once, and a* WHILE *used when a statement body may need to be repeatedly executed.* If the advanced programmers do have this rule, then we predict that they would not recognize the RESET PLAN in the Beta version nearly as often as they would in the Alpha version.

IV. Detailed Description of Study I: Fill-in-the-Blank

A. Subjects

A total of 139 students participated in the experiment. These students were recruited from programming classes and were paid $5 for participating in the experiment. There were 94 novice level programmers and 45 advanced level programmers.

Version Alpha

```
PROGRAM Gold(input,output);
  CONST
    MaxSentence=99;
    NumOfConvicts=5;
  VAR
    ConvictID, I, Sentence : INTEGER;

  BEGIN
    FOR I :=1 TO NumOfConvicts DO
      BEGIN
        READLN(ConvictID, Sentence);
        IF Sentence > MaxSentence
          THEN Sentence := MaxSentence;
        WRITELN(ConvictID, Sentence);
      END;
  END.
```

```
PROGRAM Gold(input,output);
  CONST
    MaxSentence=99;
    NumOfConvicts=5;
  VAR
    ConvictID, I, Sentence : INTEGER;

  BEGIN
    FOR I :=1 TO NumOfConvicts DO
      BEGIN
        READLN(ConvictID, Sentence);
        IF Sentence > MaxSentence
           ------------------------------------
          THEN |                                    |
           ------------------------------------
        WRITELN(ConvictID, Sentence);
      END;
  END.
```

Version Beta

```
PROGRAM Silver(input,output);
  CONST
    MaxSentence=99;
    NumOfConvicts=5;
  VAR
    ConvictID, I, Sentence : INTEGER;

  BEGIN
    FOR I :=1 TO NumOfConvicts DO
      BEGIN
        READLN(ConvictID, Sentence);
        WHILE Sentence > MaxSentence
          DO Sentence := MaxSentence;
        WRITELN(ConvictID, Sentence);
      END;
  END.
```

```
PROGRAM Silver(input,output);
  CONST
    MaxSentence=99;
    NumOfConvicts=5;
  VAR
    ConvictID, I, Sentence : INTEGER;

  BEGIN
    FOR I :=1 TO NumOfConvicts DO
      BEGIN
        READLN(ConvictID, Sentence);
        WHILE Sentence > MaxSentence
           ------------------------------------
          DO |                                    |
           ------------------------------------
        WRITELN(ConvictID, Sentence);
      END;
  END.
```

```
Program type 4

Basic plan        reset to boundary condition

Discourse rule    An IF should be used when a statement body
                  is guaranteed to be executed only once,
                  and a WHILE used when a statement body may
                  need to be repeatedly executed (5)

How construct
Beta version:     violate discourse rule (5)

Alpha case:       use IF for testing and one time execution

Beta case:        use WHILE for testing and one time execution
```

Fig. 5. Example: Program type 4.

Novice programmers were students at the end of a first course in Pascal programming. The advanced level programmers had completed at least 3 programming courses, and most were either computer science majors or first year graduate students in computer science; all had extensive experience with Pascal.

B. Materials

We created two pairs of programs (an Alpha version and a Beta version comprise one pair) for each of the 4 program types described in Section III; thus there were eight pairs of programs, two pairs of programs for each program type. One instance (an Alpha-Beta pair) of each of the four program types was presented in the preceding sections. Both instances of a program type were similar. For example, in Fig. 6 the second instance of the program type MAX is given; while the first instance of this type searched for the maximum (minimum) integer input (Fig. 1), the second instance searched from the maximum (minimum) character input.

Version Alpha

```
PROGRAM Green(input, output),
VAR I   INTEGER,
    Letter, LeastLetter   Char,
BEGIN
    LeastLetter  =  'z',
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Letter),
          If Letter <  LeastLetter
             THEN LeastLetter  = Letter,
        END,
    Writeln(LeastLetter);
END
```

```
PROGRAM Green(input, output),
VAR I   INTEGER,
    Letter, LeastLetter   Char,
BEGIN
    LeastLetter  =  'z',
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Letter),
                    -------
          If Letter |     | LeastLetter
                    |     |
                    -------
             THEN LeastLetter  = Letter,
        END,
    Writeln(LeastLetter);
END
```

Version Beta

```
PROGRAM Yellow(input, output),
VAR I   INTEGER,
    Letter, LeastLetter   Char,
BEGIN
    LeastLetter  =  'a',
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Letter),
          If Letter >  LeastLetter
             THEN LeastLetter  = Letter,
        END,
    Writeln(LeastLetter),
END
```

```
PROGRAM Yellow(input, output),
VAR I   INTEGER,
    Letter, LeastLetter   Char,
BEGIN
    LeastLetter  =  'a',
    FOR I  = 1 TO 10 DO
        BEGIN
          READLN(Letter),
                    -------
          If Letter |     | LeastLetter
                    |     |
                    -------
             THEN LeastLetter  = Letter;
        END,
    Writeln(LeastLetter),
END
```

```
      Program type 1 -- Instance 2


Basic plan          search plan (max, min)


Discourse rule      A variable's name should reflect its function (1)

How construct
Beta version        violate discourse rule (1)


Alpha case          variable name agrees with
                    search function

Beta case           variable name does NOT agree
                    with search function
```

Fig. 6. Example: Program type 1–instance 2.

C. Design: Independent and Dependent Variables

The three independent variables in this study were:

1) version–Alpha (plan-like), Beta (unplan-like)
2) program type–1 MAX, 2 SQRT, 3 AVERAGE, 4 IF/WHILE
3) level of expertise–novice or advanced.

There were two dependent variables:

1) accuracy of the response; a correct response was one that completed the intended plan,[5] and
2) time to complete a problem.

[5]Strictly speaking filling in the blank line with an answer that differs from the plan-like one would not necessarily be *incorrect*. For example, filling in the blank line in Beta of Fig. 1 with a > would still result in a running program. However, it would be a strange program. Thus, by *correct* we actually mean the line of code that in our judgment best fulfills the overall intent of the program.

D. Procedure

Each subject was given eight programs. In four of the problems, the subject received the Alpha version of the program while in the other four problems, the subject received the Beta version of the program. We also counterbalanced versions within each of the four program types such that if a subject received an Alpha version for one program of a type then the subject would receive the Beta version for the other program of the same type. The test programs were presented as a booklet in which the order of the programs was randomized for each subject. Subjects were instructed to work through the booklet in the given order. As we mentioned earlier, each program was presented with one blank; subjects were not told what problems the programs were intended to solve. Subjects were given the following instruction: *fill in the blank line with a line of Pascal code which in your opinion best completes the program.* They were given as much time to do the test as they wanted; almost all finished within an hour.

E. Results and Discussion

The main results in this study were:

- the experts performed better than the novices (61 percent versus 48 percent, $F_{1,137} = 17.27, p < 0.001$),
- subjects answered the Alpha versions correctly more often than they did the Beta versions (88 percent versus 31 percent, $F_{1,137} = 375.22, p < 0.001$).
- the interaction between program version and expertise was significant ($F_{1,137} = 6.78, p < 0.01$).

Moreover, using a Newman-Keuls test the difference in performance between the novice and the advanced subjects for the Alpha versions was significant at the 0.05 level, while there was no significant difference between the two groups of subjects on the Beta versions. Thus, the statistical analyses support the visual effect of the graph in Fig. 7: the performance of the advanced students was reduced to that of the novices by the Beta versions!

The magnitude of the change in performance by the advanced programmers is impressive (Fig. 7): the advanced programmers performed about 50 percent worse on the Beta versions that they did on the Alpha versions. (This difference was significant at the 0.01 level using a Newman-Keuls test.) Given that the only difference between the two versions was a violation of one (or possibly two) rule of programming discourse, we are impressed with the enormous size of this difference. Clearly, discourse rules in programming have a major effect on programmers' abilities to comprehend programs.

A breakdown by version and program type is given in Table I. Here we see the percentage of subjects that correctly answered each program.

- There was a significant difference in accuracy for the four program types ($F_{3,411} = 26.81, p < 0.001$).
- Also, the differences between the Alpha and Beta programs was not constant over the four program types. This interaction between program type and version was significant ($F_{3,411} = 68.39, p < 0.001$).

While we had attempted to keep all the program types at about the same level of difficulty, apparently we were not successful in this regard.

There was also a significant three-way interaction between program type, version, and expertise ($F_{3,411} = 3.12, p < 0.05$). An intuition for this interaction can be gleaned from Table I: performance on the Beta version of the SQRT program type differed greatly from the performance on the Beta versions of the other program types. This difference was statistically significant at the 0.01 level using a Newman-Keuls test. Why was the performance on the Beta versions of this one program type so high? The most plausible explanation is based on a practice effect: since in every other program that the subjects saw, data were input via a READ statement, subjects simply did not even see the conflict and immediately filled in the blank line with a READ.

In Table II we display a breakdown of the number and type of errors that subjects made. There were of course, more errors made on the Beta versions (390) than on the Alpha versions (140) ($p < 0.001$ by a sign test).[6] More interesting, however, were the type of errors made on the Beta versions. Our theory makes a strong prediction about the type of incorrect response that subjects will make on the Beta versions: if subjects do not recognize that the Beta versions are unplan-like, and if subjects are simply using plans and discourse rules to guide their responses, then we expect them to perceive the Beta version as just being an Alpha version, *and provide the plan-like response for the Alpha version.* For example, as discussed earlier (Section III-A), Program Purple in Fig. 1 actually computes the minimum of a set of inputs; however, it appears, because of the key variable name MAX to compute the maximum of some input values. The correct fill-in-the-blank answer for the Program Purple was '<'. However, we predicted that those subjects who fill in the blank incorrectly would do so by saying '>'—which *is* the correct answer for the Alpha version.

The data do bear out the above prediction: the difference between the plan-like incorrect responses and the unplan-like incorrect responses on the Beta versions was significant ($p < .01$ by a sign test):[7] 66 percent (257/390) of the incorrect responses on the Beta versions were one specific response—the response that would have been appropriate for the corresponding Alpha version of the program.

Another view of the effect of the unplan-like Beta versions on our subjects' performance can be seen by examining the amount of time it took subjects to provide a *correct* response to the Alpha and the Beta versions. Fig. 8 depicts this relationship. The difference in response time for the correct answers between the Alpha and Beta versions was significant ($F_{1,288} = 35.1, p < 0.001$); it took approximately 50 percent more time to respond correctly to the Beta versions than it did to respond correctly to the Alpha versions. The difference between novice and advanced programmers was also significant ($F_{1,288} = 8.6,$

[6]The p value of 0.001 reduces the likelihood that we are affirming a chance result from having partitioned the data.

[7]The p value of 0.01 reduces the likelihood that we are affirming a chance result from having partitioned the data.

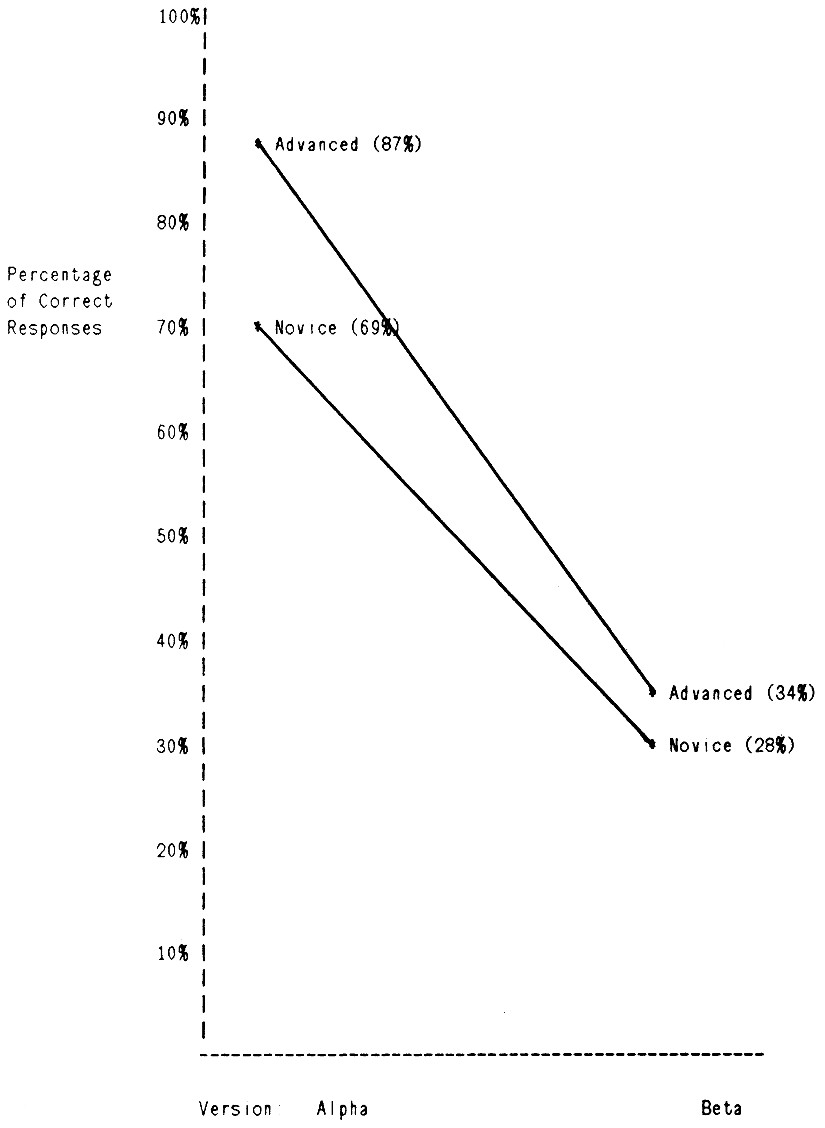


Fig. 7. Interaction: Expertise and program type.

TABLE I
PERCENTAGE OF CORRECTNESS BY PROGRAM TYPE

Program Type	Alpha	Beta
NOVICES (N = 94)		
1 MAX	78%	12%
2 SQRT	69%	61%
3 AVERAGE	80%	01%
4 IF/WHILE	48%	38%
ADVANCED (N = 45)		
1 MAX	93%	13%
2 SQRT	87%	84%
3 AVERAGE	96%	06%
4 IF/WHILE	73%	31%

TABLE II
ERROR DATA: FILL-IN-THE-BLANK STUDY

ERRORS on Alpha and Beta Versions:	
Alpha Versions:	
Total number of errors by Novice and Advanced Subjects:	140
Beta Versions:	
Total number of errors by Novice and Advanced Subjects:	390
ERRORS on only Beta Versions:	
Plan-like Errors:	
Total number on Beta versions by Novice and Advanced Subjects:	257
Unplan-like Errors:	
Total number on Beta versions by Novice and Advanced Subjects:	133

	390

$p < 0.01$); however, we did not find an interaction between expertise and program version in this situation ($F < 1$).

Our interpretation of these data is as follows: we conjecture that a correct response to the Alpha versions required only that programmers use their knowledge of programming plans and rules of programming discourse in a straightforward manner. However, in order to arrive at a correct answer to the Beta versions, subjects needed to employ additional processing techniques, e.g., trying to run the program on some sample numbers. This additional mental processing time corresponds to the increase in response time. Thus, not only do unplan-like pro-

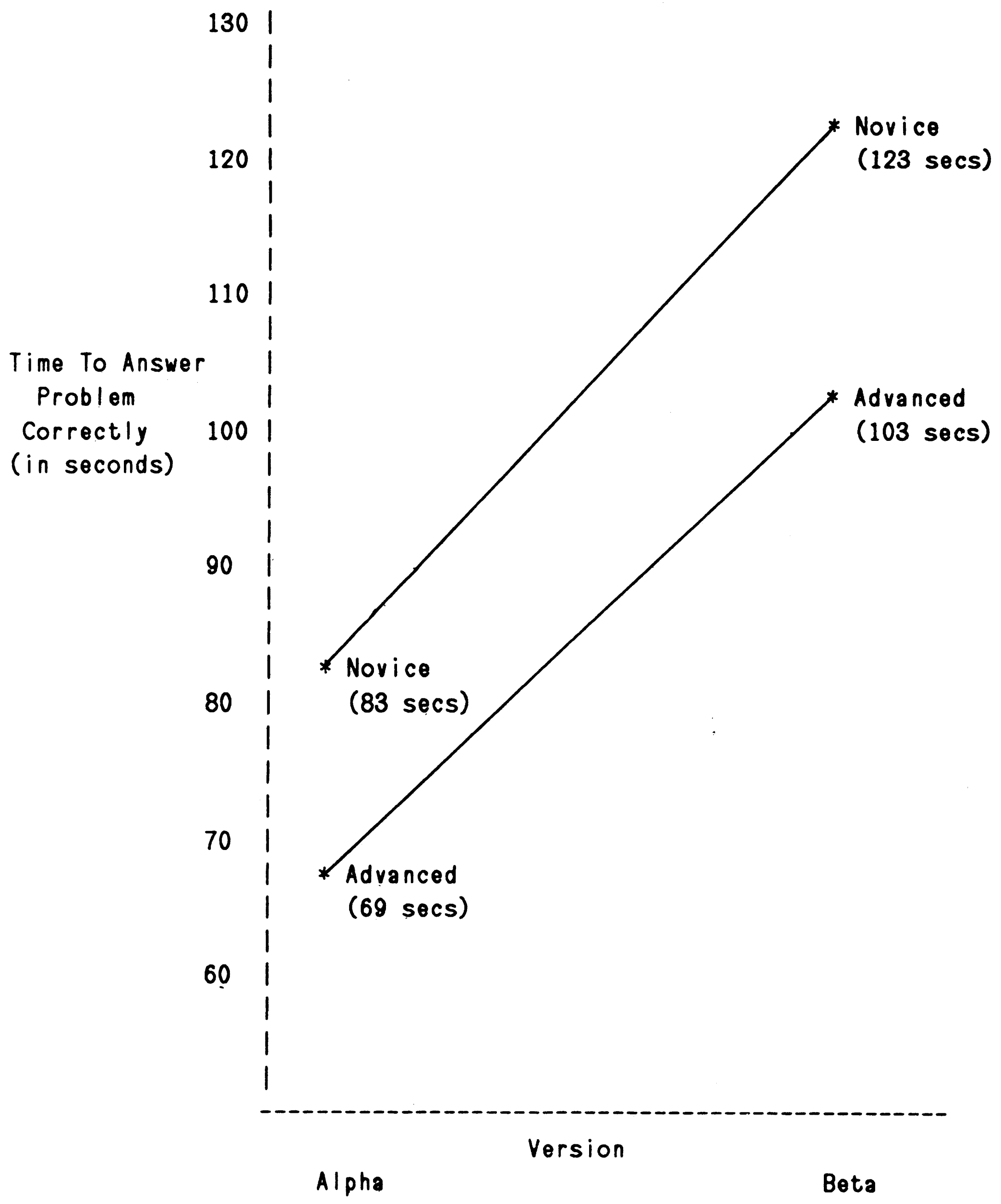


Fig. 8. Time to respond correctly: Alpha version versus beta version.

grams reduce the accuracy of programmers, their time to respond correctly to the unplan-like ones goes up dramatically since they need to bring in additional reasoning strategies in order to compensate for the unplanliness of the Beta versions.

V. Detailed Description of Study II: Recall

A. Subjects

A total of 41 professional programmers participated in this study. The mean number of years experience was 7.5, with a standard deviation of 4.8. The minimum number of years of experience was 2 and the maximum was 20. The company for which these programmers worked gave them time off during the workday to participate in this study. Thus we did not have to pay subjects for their participation.

B. Materials

The programs we used in this study were essentially the same as those used in the study described above; the main differences were:

- the programs in this study were translated from Pascal into Algol, the language used by these subjects;
- program type SQRT was eliminated; in the Beta versions, these programs simply have an extra line of code, the initialization of N, which we felt was too mild a difference from the Alpha versions.

As described in Section III, each Alpha-Beta pair of programs was essentially identical[8] except for two critical lines (e.g., see lines 5 and 9 in the programs in Fig. 9.) We have called these lines critical because they carry the information that makes the programs plan-like or not. In the following analysis, we will focus on the two critical lines in assessing whether expert

[8]Programs of type AVERAGE were slightly different (see Fig. 4): the Alpha versions used a process/read loop structure, while the Beta conditions used a process/read structure [19]. However, the Alpha programs contain fewer lines than the Beta versions; thus, by more standard measures of program complexity (e.g., Halstead [11] or "lines of code") the Alpha programs should be harder to comprehend than the Beta ones.

```
Program Type  MAX   Version  Alpha

% PROGRAM MAGENTA,
01 BEGIN
02    FILE REM (KIND = REMOTE, UNITS = CHARACTERS,
03             MAXRECSIZE = 1920, MYUSE = IO);
04    INTEGER MAX,I,NUM,
05    MAX := 0,
06    FOR I := 1 STEP 1 UNTIL 10 DO
07       BEGIN
08          READ (REM,*/,NUM),
09          IF NUM > MAX THEN MAX := NUM,
10       END;
11    WRITE(REM,*/,MAX);
12 END
```

```
Program Type  MAX   Version  Beta

% PROGRAM PURPLE,
   BEGIN
      FILE REM (KIND = REMOTE, UNITS = CHARACTERS,
               MAXRECSIZE = 1920, MYUSE = IO);
      INTEGER MAX,I,NUM,
      MAX := 1000000,
      FOR I := 1 STEP 1 UNTIL 10 DO
         BEGIN
            READ (REM,*/,NUM),
            IF NUM < MAX THEN MAX := NUM;
         END,
      WRITE(REM,*/,MAX),
   END
```

Fig. 9. Example: Critical lines in Algol programs. The critical lines in these programs—the lines that are different—are lines 05 and 09.

programmers recall plan-like programs better than unplan-like ones: we predict that the programmers should be able to recall the critical lines from the plan-like programs earlier than the critical lines from the unplan-like ones. The basis for this prediction is as follows: programmers will use their plans and discourse rules to encode a program when it is presented. In a plan-like program, the critical lines are the key representatives of the program's plans, and thus they should be recalled very early on. The fact that representatives of a category are recalled first is a recognized psychological principle [5]. However, in an unplan-like program, the critical lines do not represent the program's plans and as such should act no differently than the other lines in the program; thus, they should not be recalled first.

C. *Design: Independent and Dependent Variables*

In this study there were three independent variables:

- version–Alpha (plan-like), Beta (unplan-like)
- program type–MAX, AVERAGE, IF/WHILE
- trial–first, second, third presentation

As explained below, the dependent variable was correctness of recall of the critical lines.

D. *Procedure*

Subjects were presented with a complete program which they were asked to recall *verbatim*. The program was presented three times, each time for 20 s. On the first trial, subjects were asked to recall as much of the program as possible. On the second and third trials, they were asked to either add to their original recall or to change any part of their recall that they felt was in error. We tracked the progression of their recall by asking them to use a different color pencil on each of the three trials.

Just as in the previous study, there was two Alpha-Beta program pairs for each of three types of programs (MAX, AVERAGE, IF/WHILE). Each subject was shown a total of six programs: three Alpha and three Beta. We also counterbalanced version within each of the 3 program types such that if a subject received an Alpha version for one program of a type then the subject would receive the Beta version for the other program of the same type.

A critical line was scored as correct if and only if the line was recalled exactly as presented. If a subject recalled part of a critical line on the first trial and the rest of the line on the third trial, then the line would be scored as being recalled on the third trial. Similarly, if a subject recalled a whole line on the first trial but the recall was wrong, and if the subject corrected the line of the third trial, then again, this line would be scored as being correct on the third trial.

E. *Results and Discussion*

In Fig. 10 we present a summary of the results from the recall study. This figure shows the performance of programmers on the critical lines for all the programs. Shown are the cumulative percentages of recall for the critical lines for each of the three trials (presentations).[9] After the first trial, for example, 42 percent (101/240) of the critical lines on the Alpha versions were recalled correctly, while only 24 percent (58/240) of the critical lines on the Beta versions were recalled correctly. The effect of version was significant ($F_{1,40} = 9.05, p < 0.01$): more Alpha critical lines were recalled than Beta critical lines. The interaction of version and trial was also significant ($F_{2,80} = 4.72, p < 0.011$). The fact that the difference between the recall of the critical lines for the Alpha and the Beta versions changes over trials supports our hypothesis that the critical lines in the Alpha versions were recalled sooner than those in the Beta versions. Thus, just as in the study described previously (Section IV), we again see the significant, detrimental effect that unplan-like programs have on programmer performance.

In Table III we breakdown the errors and changes made by our three subjects. Of particular interest are the number of changes: programmers made almost three times as many changes on the Beta programs as they did on the Alpha programs.[10] Moreover, the changes made on the Beta programs were consistent with our theoretical predictions: subjects typi-

[9]The basis for this calculation is as follows: each subject was shown three Alpha programs and three Beta programs, there were two lines per program, and there were 40 subjects. Thus, there was a possible 240 critical lines in the Alpha programs, and 240 critical lines in the Beta programs.

[10]All changes on the Beta programs were from incorrect to correct; one subject changed from correct to incorrect on an Alpha program.

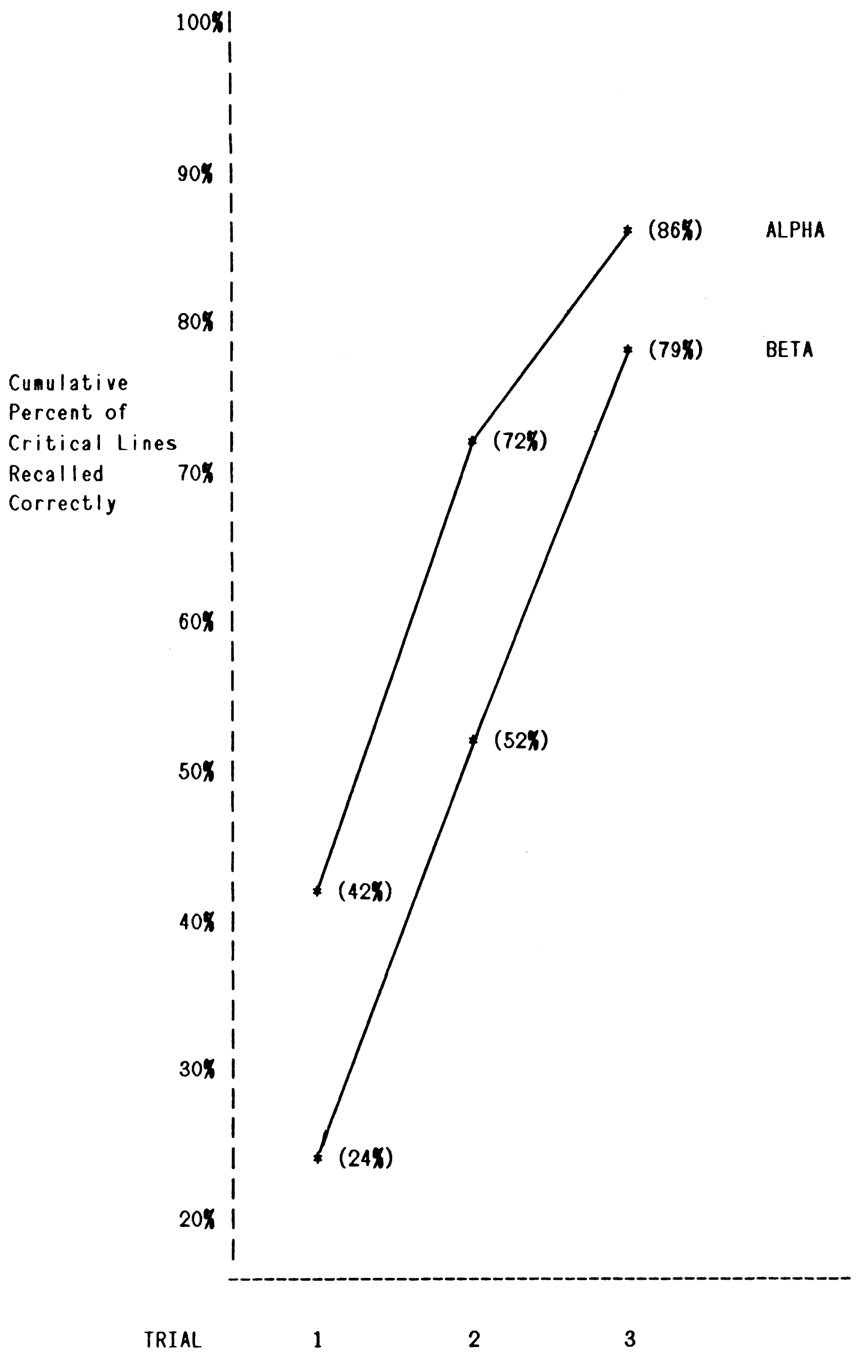


Fig. 10. Summary statistics of recall study.

cally incorrectly "recalled" the plan-like answer, and then changed their answer later to match what was actually being shown in the program. In particular, 22 out of 32 changes (69 percent) were from plan-like, but incorrect answers, to the correct answer. (This difference was significant $p < 0.025$ by a sign test.[11]) For example, on program PURPLE, which is a Beta version of type MAX (Fig. 9), of the eight subjects who made changes to the If line (line 9), seven of them initially wrote Num > Max—the response that would be correct if the program were actually finding the maximum value (see the Alpha version in Fig. 9)—and then changed their response on

[11]The p value of 0.025 reduces the likelihood that we are affirming a chance result from having partitioned the data.

TABLE III
BREAKDOWN OF THE ERRORS AND CHANGES THAT WERE MADE

	Correctly Recalled	Errors	Unrecalled	Changes
ALPHA (Out of 240)	206	17	23	12
BETA (Out of 240)	189	24	33	32

For BETA Programs Only

Changes Plan-like to Correct	Changes Otherwise
22	10

later trials to the correct Num < Max. Notice that these subjects were initially "recalling" something that *was not in the program.* Thus, just as in the fill-in-the-blank study, an analysis of incorrect responses is particularly telling: programmers expected to see plan-like programs and consistently behaved as if the programs at hand were plan-like.

VI. Concluding Remarks

The objective of these studies was to evaluate the claim that advanced programmers have and use 1) programming plans and 2) rules of programming discourse. The experiments discussed above were expressly designed to examine this question. The results of the experiments, as described above, support our initial claim.

- In Study I, when test programs were plan-like, advanced programmers did perform significantly better than novice programmers; however, when the test programs were *not* plan-like (i.e., the plan composition violated some rule of programming discourse), then the performance of the advanced programmers was reduced to essentially that of the novice programmers.
- In Study II, the performance of the expert programmers was significantly different on the plan-like programs as compared to the unplan-like ones: the critical lines in the plan-like programs were recalled earlier than those in the unplan-like ones.

On the one hand, the results point to the fragility of programming expertise: advanced programmers have *strong* expectations about what programs should look like, and when those expectations are violated—in seemingly innocuous ways—their performance drops drastically. On the other, the results support our claim that the plan knowledge and the discourse rule knowledge, upon which the expectations are built, do play a powerful role in program comprehension.

We hasten to point out that measures of surface characteristics, such as lines of code or Halstead metrics, would not predict the differences in performance we obtained. The Beta versions typically has either the same number of lines of code or slightly fewer lines of codes than did the comparable Alpha versions. We certainly do not dispute the results of earlier studies that show that such surface measures do correlate with program complexity (e.g., [6]). However, as our study vividly shows, surface feature measures do not necessarily predict complexity.

More importantly, our approach is to provide *explanations* for why a program may be complex and thus hard to comprehend. Towards this end, we have attempted to articulate the programming knowledge that programmers have and use. Thus, our intent is to move beyond *correlations* between programmer performance and surface complexity as measured by Halstead metrics, lines of code, etc., to a more principled, cognitive explanation (see also, [20]).

A potential criticism of this work is that the programs we used in the experiments were unrealistic: while our experimental programs were short, the programs produced by experts are typically much longer. One major rationale for the use of short programs was experimental control: we wanted to keep as much constant as possible and only vary one (or possibly two) discourse rule. Given the range of results we obtained for the different program types (see Table I) we feel justified in our concern. Nonetheless, we are sensitive to the above criticism: while our intuition is that the effects we observed will in fact be more pronounced in longer programs, clearly, our studies need to be replicated with longer programs. While not discounting the value of this criticism, we feel that the magnitude of effects that we observed is too pronounced to simply be ignored.

In closing, then, our studies support the claim that knowledge of programming plans and rules of programming discourse can have a significant impact on program comprehension. In their book called *Elements of Style*, Kernighan and Plauger [13] also identify what we would call discourse rules. Our empirical results put teeth into these rules: it is not merely a matter of aesthetics that programs should be written in a particular style. Rather there is a psychological basis for writing programs in a conventional manner: programmers have strong *expectations* that other programmers will follow these discourse rules. If the rules are violated, then the utility afforded by the expectations that programmers have built up over time is effectively nullified. The results from the experiments with novice and advanced student programmers and with professional programmers described in this paper provide clear support for these claims.

Acknowledgments

The authors would like to thank a number of colleagues who provided invaluable aid and comments on this research: B. Adelson, C. Seifert, D. Littman, E. Gold. We would also like to thank the reviewers for their helpful comments.

References

[1] B. Adelson, "Problem solving and the development of abstract categories in programming languages," *Memory and Cognition*, vol. 9, pp. 422-433, 1981.
[2] F. C. Bartlett, *Remembering*. Cambridge, MA: Univ. Press, 1932.
[3] G. H. Bower, J. B. Black, and T. Turner, "Scripts in memory for text," *Cognitive Psychol.*, vol. 11, pp. 177-220, 1979.
[4] W. C. Chase, and H. Simon, "Perception in chess," *Cognitive Psychol.*, vol. 4, pp. 55-81, 1973.
[5] R. G. Crowder, "Principles of learning and memory," Lawrence Erlbaum Associates, Hillsdale, NJ, 1976.
[6] B. Curtis, S. Sheppard, and P. Milliman, "Third time charm: Stronger prediction of programmer performance by software complexity metrics," in *Proc. 4th Int. Conf. Software Eng.*, IEEE Comput. Soc., N. Denton, TX, 1979.
[7] A. D. deGroot, *Thought and Choice in Chess*. Paris, France: Mouton, 1965.
[8] D. Egan and B. Schwartz, "Chunking in recall of symbolic drawings," *Memory and Cognition*, vol. 7, pp. 149-158, 1979.
[9] K. Ehrlich and E. Soloway, "An empirical investigation of the tacit plan knowledge in programming," in *Human Factors in Computer Systems*, J. Thomas and M. L. Schneider, Eds. Norwood, NJ: Ablex Inc., 1984.
[10] A. C. Graesser, *Prose Comprehension Beyond the Word*. New York: Springer-Verlag, 1981.
[11] M. M. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
[12] J. H. Kaheny, "Problem solving by novice programmers," in *The Psychology of Computer Use: A European Perspective*. London, England: Academic, 1983.

[13] B. Kernighan and P. Plauger, *The Elements of Style*. New York: McGraw-Hill, 1978.

[14] J. Larkin, J. McDermott, D. Simon, and H. Simon, "Expert and novice performance in solving physics problems," *Science*, vol. 208, pp. 140-158, 1980.

[15] K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle, "Knowledge organization and skill differences in computer programmers," *Cognitive Psychol.*, vol. 13, pp. 307-325, 1981.

[16] C. Rich, "Inspection methods in programming," M.I.T. Artificial Intell. Lab., Cambridge, MA, Tech. Rep. TR-604, 1981.

[17] R. C. Schank and R. Abelson, "Scripts, plans, goals, and understanding," Lawrence Erlbaum Associates, Hillsdale, NJ, 1977.

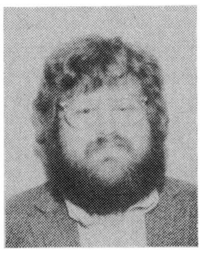
[18] B. Shneiderman, "Exploratory experiments in programmer behavior," *Int. J. Comput. Inform. Sci.*, vol. 5, no. 2, pp. 123-143, 1976.

[19] E. Soloway, K. Ehrlich, and J. Bonar, "Tapping into tacit programming knowledge," in *Proc. Conf. Human Factors in Comput. Syst.*, Nat. Bureau Standards, Gaithersburg, MD, 1982.

[20] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs: An empirical study," *Commun. ACM*, vol. 26, pp. 853-861, 1983.

[21] E. Soloway, K. Ehrlich, and J. Black, "Beyond numbers: Don't ask "how many" ...ask "why", " in *Proc. SIGCHI Conf. Human Factors in Comput. Syst.*, SIGCHI, Boston, MA, 1983.

[22] L. Weissman, "Psychological complexity of computer programs: An experimental methodology," *SIGPLAN Notices*, vol. 9, June 1974.



Elliot Soloway received the B.A. degree in philosophy from Ohio State University, Columbus, in 1969, and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts, Amherst, in 1976 and 1978, respectively.

He is an Assistant Professor of Computer Science at Yale University, New Haven, CT, and a Vice President of Compu-Tech, Inc., New Haven, an educational software company. His research interests include the cognitive underpinnings of programming, artificial intelligence, and computer science education.

Dr. Soloway is a member of the Association for Computing Machinery and the Cognitive Science Society.

Kate Ehrlich received the B.A. degree from the University of London, London, England, in 1973 and a Ph.D. degree from the University of Sussex, Sussex, England, in 1979.

She is a Human Factors Psychologist at Honeywell Information Systems, Inc., Waltham, MA. Her research interests include human-computer interaction, expert/novice difference in problem solving, artificial intelligence, and computers and education.

Dr. Ehrlich is a member of the Association for Computing Machinery, the American Association for Artificial Intelligence, the Cognitive Science Society, the Human Factors Society, and the Psychonomic Society.