Human Aspects of Computing

Henry F. Ledgard Editor

Cognitive Strategies and Looping Constructs: An Empirical Study

ELLIOT SOLOWAY *Yale University*
JEFFREY BONAR *University of Massachusetts*
KATE EHRLICH *Yale University*

At the Cognition and Programming Project, Soloway, Ehrlich, and Bonar are attempting to (1) understand how programmers—novices and experts—read and write programs, and (2) build software environments based firmly on this psychological research that facilitate the programming process.

Authors' Present Addresses: Elliot Soloway and Kate Ehrlich, Department of Computer Science, Yale University, P.O. Box 2158, New Haven, CT 06520; Jeffrey Bonar, Learning Research and Development Center, Univ. of Pittsburgh, Pittsburgh, PA 15260.

This work was supported by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant No. MDA903-80-C-0508. This work was also supported by the National Science Foundation under NSF Grant SED-81-12403.

Any opinions, findings, conclusions or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

ABSTRACT: In this paper, we describe a study that tests the following hypothesis: A programming language construct that has a closer "cognitive fit" with an individual's preferred cognitive strategy will be easier to use effectively. After analyzing Pascal programs that employed loops, we identified two distinct looping strategies: 1) on the ith pass through the loop, the ith element is both read and processed (the READ/PROCESS strategy); and 2). on the ith pass, the ith element is processed and the next ith element is read (the PROCESS/READ strategy). We argue that the latter strategy is associated with the appropriate use of the Pascal* while *construct. In contrast, we feel that a construct that allows an exit from the middle of the loop (e.g.,* loop . . . leave . . . again*) facilitates the former (READ/PROCESS) strategy. Our results indicate that subjects overwhelmingly preferred a READ/PROCESS strategy over a PROCESS/READ strategy. When writing a simple looping program, those using the* loop . . . leave . . . again *construct were more often correct than were those using the standard Pascal loop constructs.

1. INTRODUCTION

The need for the public to be literate in computing is rapidly being recognized. One aspect of such literacy is programming. While we do not believe that everyone needs to become a professional programmer, it is increasingly important to be able to describe to the computer how it is supposed to realize one's intentions. The characteristics of the language in which novice or casual programmers describe their plans are of critical importance. We might well expect professional programmers to adapt to the constraints and implicit strategies facilitated by a particular language. However, if the language does not "cognitively fit" with the non-professionals' problem-solving skills, then a barrier has been erected to their use of computers.

Concern for finding a better match between a language and an individual's natural skills and abilities is reflected in some recent empirical research. For example, Ledgard et al. [6] compared an editing language whose syntax was based on English with a standard notational editing language, and found that the English-based language was preferred by the subjects and led to better performance. Miller [4] examined the natural problem-solving strategies of nonprogrammers in order to explore the potential for "natural language" programming. One conclusion he draws that is particularly relevant to this paper is that programming language constructs could be developed that were closer to how people "naturally" specified problem solutions. Claims have also been made (e.g., [7]) that the procedurality in programming taps into novices' pre-existing cognitive notions. In support of this claim, Soloway et al. [9] have shown that students write correct equations more often when solving simple word problems using a procedural programming language as opposed to using algebra, a non-procedural language. Similarly, Welty and Stemple [12] compared the performance of novices using a procedural query language with those using a non-procedural query language; they found that subjects performed at a higher level of accuracy with the procedural language when writing moderate to difficult queries.

We report here on an experiment that explores the relationship between the preferred cognitive strategies of individuals and programming language constructs. By *preferred strategy*, we mean the strategy that individuals *spontaneously* use when solving a problem. We will focus on looping strategies and examine the impact they have on the use of looping constructs.

2. TWO STRATEGIES: READ/PROCESS VERSUS PROCESS/READ

Consider the following problem:

> **The Averaging Problem**
> Write a program that repeatedly reads in integers, until it reads the integer 99999. After seeing 99999, it should print out the **correct** average. That is, it should not count the final 99999.

This problem is certainly neither tricky nor esoteric; one would expect this problem to be easy for students at the end of a semester course on Pascal. In fact, we found that students do surprisingly poorly on this and related problems.[1] In this problem, the loop is dependent on the variable that holds the new values as they are successively read in.[2] In this situation, the loop may not be executed even once, and thus the Pascal loop construct most appropriate is the **while** construct [13]. In Figure 1, we depict the stylistically correct Pascal solution to this problem.

Stepping back from the code, the strategy that this program embodies can be characterized as:

```
Read (first value)
while Test (ith value)
   do begin
      Process (ith value)
      Read (i + 1st value)
   end
```

Since the loop may not be executed if the first value read is 99999, a *Read* outside the loop is necessary in order to get the loop started. However, this results in the loop processing being one step behind the *Read*; on the *i*th pass through the loop, the *i*th value is processed and *then* the *i*th + 1 value is read in. We call this strategy "process i/read next i" (henceforth referred to as PROCESS/READ).

We felt this strategy to be unnecessarily awkward and confusing [3, 11]. In effect, processing in the loop would be "out of sync" with reading in the loop. From a cognitive perspective, we speculate that such a strategy puts an extra burden on memory and processing resources. We suggest that a more natural cognitive strategy would be to read the *i*th value and process it on the *i*th pass through the loop; we call the "read i/process i" strategy (henceforth referred to as READ/PROCESS). For example:

```
loop
  do begin
    Read (ith value)
    Test (ith value)
    Process (ith value)
  end
```

This strategy would have the reading and processing "in sync", and should require less cognitive resources than the PROCESS/READ strategy.

Although the PROCESS/READ strategy is facilitated by Pascal's **while** loop, a READ/PROCESS can be encoded using either the **while** or the **repeat** loop. For example, Figure 2 depicts three Pascal programs that use **while** and **repeat** loops and implement the READ/PROCESS strategy.[3] These are actual student programs generated in an earlier experiment [10]. The programs in Figures 2a and 2b use an embedded **if** statement to effect the READ/PROCESS strategy. In the former case, a Boolean variable is used to control the outside **while** loop; in the latter case the same test is performed twice. In Figure 2c, we see a program in which a **repeat** loop is used to implement the READ/PROCESS strategy; the stop value is simply subtracted from the total. While correct, all three programs need to employ a considerable amount of additional code in order to compensate for not employing the appropriate PROCESS/READ strategy.

```
program Example1;
   var Count, Sum, Number : integer;
      Average : real;
   begin
      Count := 0;
      Sum := 0;
      Read (Number);
      while Number <> 99999 do
         begin
            Sum := Sum + Number;
            Count := Count + 1;
            Read (Number)
         end;
      if Count > 0
         then
            begin
               Average := Sum / Count;
               Writeln (Average);
            end
         else
            Writeln ('No numbers input:
                        average undefined');
   end.
```

FIGURE 1. A Stylistically Correct Pascal Solution to the Averaging Problem.

[1] In an earlier study [10], we asked students to write a program that solves the averaging problem stated above. In grading their problems, we overlooked syntax errors; only 38% were able to produce a correct program. This test was given to students on the last day of classes after a semester course on Pascal programming.

[2] Loops can also be dependent on variables playing other roles, e.g., the counter, the running total. If the counter variable controls the loop, then Pascal's **for** loop is most appropriate; if the running total variable (Sum, in Figure 1) controls the loop, then the loop can reasonably be expected to be executed at least once, hence the **repeat** loop is most appropriate (see [10]).

[3] The goto was not taught to students in this class; thus, it does not appear in the students' programs.

```
program Example2a;
   var N, Sum, X : integer;
       Average : real;
       Stop : boolean;
   begin
      Stop := false;
      N := 0;
      Sum := 0;
      while not Stop do
         begin
            Read (X);
            if X = 99999
               then Stop := true
               else
                  begin
                     Sum := Sum + X;
                     N := N + 1
                  end
         end;
      Average := Sum / N;
      Writeln (Average)
   end.
```

2(a)

```
program Example 2b;
   var Num, Sum, N : integer;
       Avg : real;
   begin
      Num := 0;
      N := 0;
      Sum := 0;
      while Num <> 99999 do
         begin
            Read (Num);
            if Num <> 99999 then
               begin
                  Sum := Sum + Num;
                  N := N + 1
               end
         end;
      Avg := Sum / N;
      Writeln (Avg)
   end.
```

2(b)

```
program Example 2c;
   var Count, Sum, Num : integer; Average : real;
   begin
      Count := -1;
      Sum := 0;
      repeat
         Count := Count + 1;
         Read (Num);
         Sum := Sum + Num
      until Num = 99999;
      Sum := Sum - 99999;
      Average := Sum / Count;
      Writeln(Average);
   end.
```

2(c)

FIGURE 2. Effecting a READ/PROCESS Strategy. (a) Using a Boolean Variable and a Nested Condition to Effect a READ/PROCESS Strategy. (b) Using a Nested Conditional and a Repeated Test to Effect a READ/PROCESS Strategy. (c) Using a Repeat Loop and Backing Down to Effect a READ/PROCESS Strategy.

Consider, then, the following looping construct, that is similar to one in Ada, the new DOD language:

```
loop;
   S;
   if B then leave;
   T;
again
```

where S and T are zero or more statements and *B* is the test condition. This construct clearly facilitates a READ/PROCESS strategy, since the test can come in the middle of the loop, between the read and the process. In Figure 3, we depict the averaging problem, described above, encoded using "Pascal L," a version of standard Pascal in which the only looping construct is **loop** . . . **leave** . . . **again**. Note that unlike the programs in Figure 2, no extraneous machinery is required to encode the READ/PROCESS strategy. Note too, however, that the **loop** . . . **leave** . . . **again** construct can also be used to encode a PROCESS/READ strategy: If S is empty, then the test is at the top of the loop, thereby creating a standard Pascal **while** loop.

3. HYPOTHESES

As stated above, we are interested in the strategies that people prefer to use to solve problems and the degree to which those strategies are compatible with the constructs of programming languages. In particular, we hypothesize: *People will find it easier to program correctly when the language facilitates their preferred strategy.*

Pascal, with the normal **while** and **repeat** constructs, can be used to implement either the READ/PROCESS strategy or the PROCESS/READ strategy. Moreover, Pascal L (Pascal with only the **loop** . . . **leave** . . . **again** construct) can also be used to encode either strategy. However, for problems in which the loop test is dependent on the values read in, Pascal's **while** construct facilitates a PROCESS/READ strategy whereas Pascal L facilitates a READ/PROCESS strategy. Our claim, then, is that for the type of problem discussed above, people should find Pascal L easier to program correctly than Pascal.

Our hypothesis leads us to ask three particular questions: 1. *Which strategy do people naturally use?* To answer this question we need to examine which strategy people adopt when they think about the problem and commit their thoughts to

```
program Pascal L;
   var Count, Sum, NewValue: integer;
          Average: real;
   begin
      Count := 0;
      Sum := 0;
      loop
         Read (NewValue);
         if NewValue = 99999 then leave;
         Sum := Sum + NewValue;
         Count := Count + 1;
      again
      if Count > 0
         then
            begin
               Average := Sum / Count;
               Writeln (Average);
            end
         else
               Writeln ('No numbers input:
                          average undefined');
   end.
```

FIGURE 3. The Averaging Problem Using Pascal *L*.

paper using a natural language—English—that is neutral with respect to READ/PROCESS or PROCESS/READ.

Once having determined whether people will adopt a READ/PROCESS or a PROCESS/READ type of strategy, we can go on to ask: 2(a). *Will people write correct programs more often when using the language that facilitates their preferred strategy?* Thus, if people use a READ/PROCESS strategy in their initial thinking, we would predict that they should write correct programs more often when using Pascal L, since this language facilitates a READ/PROCESS strategy, as compared with Pascal.

An ancillary question to 2(a) is: 2(b). *Irrespective of whether a strategy is preferred or not, will people write correct programs more often when using the strategy facilitated by the language?* That is, will people who use a READ/PROCESS strategy in Pascal L write correct programs more often than those using a PROCESS/READ strategy in Pascal L? Similarly, will people who use a PROCESS/READ strategy in Pascal write correct programs more often than those using a READ/PROCESS strategy in Pascal?

A third question of interest concerns the influence of programming experience on performance. We expect accuracy to improve when people have more experience in using a particular language. It is less clear, however, whether this experience will change the way people think about a problem. We need to ask: 3. *Do the following vary with experience: accuracy of solution, preference for a particular strategy, sensitivity to the strategy facilitated by a language.*

4. EXPERIMENTAL DESIGN

In order to gather empirical data on these questions, we designed the study described below. Students were given a two-part test, the first part of which is reproduced in Figure 4, where we asked them to write a plan that would solve the stated problem. The second part of the test is depicted in Figures 5 and 6. Half the students were asked to write a

Please write a PLAN which solves the problem described below, and which you would use to guide eventual program development. The plan should NOT be in a programming language; other than that restriction, the choice of "plan language" is up to you.

PLEASE SHOW ALL YOUR WORK!!!!! DO NOT ERASE!!!!!!

PROBLEM:
Write a plan for a program which reads in a series of integers, and which computes the average of these numbers. The program should stop reading integers when it has read the number 99999. NOTE: the final 99999 should NOT be reflected in the average you calculate.

FIGURE 4. All Subjects Were Asked to Produce a Plan.

Standard Pascal provides three looping statements: WHILE, REPEAT, and FOR. Below is a brief review of these statements. Please read the review carefully

```
 WHILE expression
   DO statements
```

A WHILE loop repeatedly does the *statements* while the *expression* is true. In other words, *expression* is tested initially and after each execution of the *statements*.

```
  REPEAT
    statements
  UNTIL expression
```

A REPEAT loop repeatedly does the *statements* until the *expression* is true. That is, *statements* are executed initially and then expression is tested for each repetition of the loop.

```
  FOR identifier :=
       expression-alpha TO expression-
  beta
    DO statements
```

A FOR loop does the statements for each value of the identifier from *expression-alpha* to *expression-beta*. First, *identifier* is set to the value of *expression-alpha* and the *statements* are executed. Then, *identifier* is set to the value of *expression-alpha* + 1 and the *statements* are again executed. This continues until *identifier* is finally set to the value of *expression-beta* and the *statements* are executed for the last time.

PROBLEM

Write a Pascal program which reads in a series of integers, and which computes the average of these numbers. The program should stop reading integers when it has read the number 99999. NOTE: the final 99999 should NOT be reflected in the average you calculate.

REMEMBER, you should use standard Pascal.
(Please use the program outline provided. DO NOT ERASE ANY WORK. If you want to start fresh, use a new program outline. *Turn in all work.*)

```
  PROGRAM PROBLEM (INPUT OUTPUT);
  VAR
  (* BEGIN YOUR STATEMENTS HERE ... *)
```

FIGURE 5. The Pascal Version of the Study.

We have just designed a new language called Pascal L. It is just like standard Pascal except that it does NOT have the WHILE, REPEAT, and FOR looping statements. Rather, Pascal L has a new kind of statement: LOOP..LEAVE..AGAIN.

The following describes how this new looping statement works:

```
LOOP
  statements-alpha
  IF expression LEAVE
  statements-beta
AGAIN
```

means:

- execute *statements-alpha*, which could be zero or more legal Pascal statements,
- then, test *expression*,
  - ▶ if *expression* is TRUE, skip to the statement AFTER the AGAIN
  - ▶ if *expression* is FALSE, continue through the loop and execute *statements-beta*, which could be zero or more legal Pascal statements, and do the loop all over again.

In other words, as long as the *expression* stays FALSE, all the statements before LOOP and AGAIN will continue to be executed.

For example, the following Pascal-L programs print out the numbers 1 through 10 and only use the LOOP . . . LEAVE . . . AGAIN loop construction:

```
PROGRAM example1(output);  PROGRAM example2(output);  PROGRAM example3(output);
  VAR i : INTEGER;           VAR i : INTEGER;            VAR i := INTEGER;
  BEGIN                      BEGIN                       BEGIN
  i := 1;                    i := 1;                     i := 1;
  LOOP                       LOOP                        LOOP
    Writeln(i);                IF i > 10 LEAVE;            Writeln(i);
    IF i >= 10 LEAVE;          Writeln(i);                 i := i + 1;
    i := i + 1                 i := i + 1                  IF i > 10 LEAVE
  AGAIN                      AGAIN                       AGAIN
END.                         END.                        END.
```

We would like you to use the LOOP..LEAVE..AGAIN statement in the program you write for the problem described on the next page. Thank you for your cooperation.

PROBLEM

Write a Pascal-L program which reads in a series of integers, and which computes the average of these numbers. The program should stop reading integers when it has read the number 99999.
NOTE: the final 99999 should NOT be reflected in the average you calculate.
REMEMBER, you may only use the
LOOP . . . LEAVE . . . AGAIN looping statement.

(Please use the program outline provided. DO NOT ERASE ANY WORK. If you want to start fresh, use a new program outline. *Turn in all work.*)

```
PROGRAM PROBLEM (INPUT, OUTPUT)
VAR
(* BEGIN YOUR STATEMENTS HERE ... *)
```

FIGURE 6. The Pascal *L* Version of the Study.

Pascal program that solved the problem, while the other half were asked to solve the problem using Pascal L. Each group was given a one-page discussion of the respective loop constructs, i.e., the Pascal L group was given a one-page description of the **loop . . . leave . . . again** construct (Figure 6), while the Pascal group was given a one-page description of the **for**, **repeat**, and **while** constructs (Figure 5). The one page on the **loop . . . leave . . . again** construct of Pascal L contained three examples; we were careful to include an instance of using the **if . . . leave** that branched off the top of the loop (that is equivalent to a **while**), an instance of using the **if . . . leave** that branched at the bottom of the loop (that is equivalent to a **repeat**), as well as an instance that branched in the middle. As much time as necessary was given to students taking this test although subjects typically finished in 10–15 minutes.

This test was administered to three different groups: novices, intermediates, and advanced students. Novices were students currently taking a first programming course in Pascal. The test was administered after the novices had been taught about and had experience with the **while** loop and the other two looping constructs; this occurred three-quarters of the way through the semester. Intermediates were students currently two-thirds through a second course in programming (e.g., either a data structures course using Pascal or an assembly language course). The advanced group were juniors and seniors in systems programming and programming methodology courses.

TABLE I: Strategy on Plans

	READ/ PROCESS[1]	PROCESS/ READ[1]	N	Misc.[2]
Novices	82%	18%	39	77
Intermediates	91%	9%	90	22
Advanced	67%	33%	48	4

A Chi-square test was used to analyze these data: $\chi^2 = 12.96$, $p < 0.01$

[1] The percentages are based on N, the number of people who had an identifiable strategy, (*i.e.*, they do not include those in the Misc. category).

[2] This column depicts the number of individuals for which we could not identify a strategy in their plan.

TABLE II: Strategy on Programs

	READ/ PROCESS[1]	PROCESS/ READ[1]	N	Misc.[2]
Novices	86%	14%	64	52
Intermediates	72%	28%	89	23
Advanced	60%	40%	49	3

[1] The percentages are based on N, the number of people who had an identifiable strategy (*i.e.*, they do not include those in the Misc. category).

[2] This column depicts the number of individuals for which we could not identify a strategy in their program.

5. RESULTS

Question 1: Which Strategy Do People Naturally Use?

In Table I, we display the results from the first part of the test where we asked people to write down their plans for solving the averaging problem. (Half of the intermediate group were asked to write a *plan* and half were asked to write a *flowchart*. We found no reliable difference between the two groups in their choice of strategies. Thus, for reporting purposes, we have combined the results of these two groups.) These results clearly indicate that all three populations had a strong preference for the READ/PROCESS strategy when it was possible for us to discern any strategy at all. Across all three groups, of those students who had a discernible strategy, 80% used the READ/PROCESS strategy, while only 20% used the PROCESS/READ strategy in their plans.[4]

Now consider Table II, where we show the strategy choice on the program, irrespective of language (see Table V). Except for the advanced group, we again see a strong preference for the READ/PROCESS strategy. That is, over all three groups, of the subjects who had a clearly discernible strategy, 73% of them used the READ/PROCESS strategy while only 27% used the PROCESS/READ strategy. These data support the claim that given the two alternatives, the preferred strategy is READ/PROCESS rather than PROCESS/READ.

It is also illuminating to look at the students who used the same strategy on both plan and program, and those who did not, i.e., those that changed strategies. Of the 158[5] students who had a discernible strategy for both plan and program, 78% (123) used the same strategy on both plan and program, while only 22% (35) switched strategies. Of the ones who did not switch, 82% (101) used a READ/PROCESS strategy on both plan and program. Again, this supports our claim that READ/PROCESS is the preferred strategy.

Interestingly, some students appeared to be sensitive to the strategy facilitated by the programming language: in Table III we show a breakdown by language type for those subjects who did (and did not) switch strategies between the plan and the program. These data indicate that subjects in the Pascal group switched more often than did subjects in the Pascal L group. This is as expected: by comparison, there were many more READ/PROCESS plans than there were PROCESS/READ plans; thus, Pascal L subjects could stay with a READ/PROCESS strategy, while Pascal subjects who were sensitive to the fact that the appropriate strategy for the problem was PROCESS/READ needed to switch strategies.

[4] The category of "Miscellaneous" was made up of those plans in which we could not discern a clear READ/PROCESS or PROCESS/READ strategy. Of the 77 novices and 22 intermediates with miscellaneous plans, 40 novices and 11 intermediates had plans that were too sketchy for categorization; 18 novices and 5 intermediates wrote nonprocedural plans—typically they simply restated the problem; 17 novices and 3 intermediates solved the wrong problem, and 2 novices and 3 intermediates wrote no plan. Clearly, the large number in this category is interesting in its own right; however, we feel that explanations for these data can reasonably be decoupled from the specific issues raised in this paper.

[5] There were fewer plans (177) than programs (202) in which we could clearly detect a strategy. However, of the former group, there were 19 students who did not have a discernible strategy on their program; hence, there were only 158 students who had discernible strategies on both plan and program.

Question 2a: Will People Write Correct Programs More Often When Using The Language That Facilitates Their Preferred Strategy?

While it seems clear that people prefer a READ/PROCESS strategy, the key question is whether or not this preference can lead to program correctness. From the data shown in Table IV, it can be seen that more people wrote a correct program using Pascal L, the language that facilitates a READ/PROCESS strategy, than did those using Pascal. The incorrect programs exhibited a number of standard bugs, in particular the "off-by-1" bug. Students would typically employ a READ/PROCESS

TABLE III: People Who Did and Did Not Switch Strategies

	People Switching Strategies from Plan to Program	People NOT Switching Strategies from Plan to Program	Statistical Significance[1]
Pascal Group	23	50	$\chi^2 = 6.89$
Pascal *L* Group	12	73	$p < 0.01$
	35	123	

[1] A Chi-square test was used in this comparison.

TABLE IV: Program Correctness with Respect to Language

	Correct[1]	Incorrect[1]	N	Statistical[2] Significance
Novices				
Pascal *L* Group	24% (14)	76% (44)	58	Not Significant
Pascal Group	14% (8)	86% (50)	58	
Intermediates				
Pascal *L* Group	61% (36)	39% (23)	59	($\chi^2 = 7.08$)
Pascal Group	36% (19)	64% (34)	53	$p = 0.01$
Advanced				
Pascal *L* Group	96% (25)	4% (1)	26	($\chi^2 = 6.58$)
Pascal Group	69% (18)	31% (8)	26	$p < 0.02$
Novices + Intermediates + Advanced				
Pascal *L* Group	52% (75)	48% (68)	143	($\chi^2 = 10.98$)
Pascal Group	33% (45)	67% (92)	137	$p < 0.001$

[1] The numbers in parentheses represent actual numbers, not percentages.

[2] A Chi-square test was used in this comparison.

TABLE V: Program Correctness with Respect to Language and Strategy

	Strategy on Program[3]	Correct[1]	Incorrect[1]	*N*	Statistical[2] Significance
Novices					
Pascal *L* Group	* READ/PROCESS	48% (14)	52% (15)	29	
	PROCESS/READ	0% (0)	100% (1)	1	See note[4]
	Misc.			28	
Pascal Group	READ/PROCESS	0% (0)	100% (26)	26	
	* PROCESS/READ	100% (8)	0% (0)	8	See note[4]
	Misc.			24	
Intermediates					
Pascal *L* Group	* READ/PROCESS	79% (34)	21% (9)	43	$(\chi^2 = 7.62)$
	PROCESS/READ	29% (2)	71% (5)	7	$p < 0.01$
	Misc.			9	
Pascal Group	READ/PROCESS	14% (3)	86% (18)	21	$(\chi^2 = 21.6)$
	* PROCESS/READ	89% (16)	11% (2)	18	$p < 0.001$
	Misc.			14	
Advanced					
Pascal *L* Group	* READ/PROCESS	96% (23)	4% (1)	24	
	PROCESS/READ	100% (2)	0% (0)	2	See note[4]
	Misc.			0	
Pascal Group	READ/PROCESS	40% (2)	60% (3)	5	$(\chi^2 = 5.50)$
	* PROCESS/READ	89% (16)	11% (2)	18	$p < 0.02$
	Misc.			3	

[1] The numbers in parentheses represent actual numbers, not percentages.
[2] A Chi-square test was used in this comparison. Note that the *Misc.* category was not used in the Chi-square calculation.
[3] Asterisks indicate the strategy that was appropriate for the language.
[4] Although the numbers are in the predicted direction, there are too few individuals in some of the cells to permit a Chi-square analysis.

strategy and thus include the final 99999 both in the sum and in the count of numbers. Almost none of the students, introductory or advanced, tested to see if the count was zero. For the purposes of our experiment, we did not count such programs as incorrect, if that were the only bug. Also, we did not count as incorrect programs that were only syntactically incorrect (e.g., missing semicolons).

Except for the novice group, all other groups showed significant improvement with respect to correctness when using Pascal L as compared to standard Pascal. (Although the novices show the same direction of effect as the other groups, the difference in their performance is not significant due to the large number of incorrect programs.) Given that students were exposed to the **loop... leave... again** construct of Pascal L for only a few minutes, and given that they had much more familiarity and experience with Pascal's standard loop constructs, we were quite impressed with the high performance of the Pascal L users. Thus, these data support the claim that people will write correct programs more often if they use the language that facilitates their preferred strategy.

Question 2b: Irrespective Of Whether A Strategy Is Preferred Or Not, Will People Write Correct Programs More Often When Using The Strategy Facilitated By The Language?
In order to answer Question 2a we needed to compare performance across languages (Pascal versus Pascal L). However, in order to answer Question 2b, we need to look at correctness within a language as a function of strategy (Table V). First consider the intermediate group's performance; there we see quite clearly that those in the Pascal L group who used a READ/PROCESS strategy on their program were able to write a correct program more often than those who used a PROCESS/READ strategy. Similarly, those in the Pascal group who used a PROCESS/READ strategy on their program were able to write a correct program more often than those who used a READ/PROCESS strategy. Thus, it seems that a sensitivity to the strategy facilitated by the language constructs can have a significant effect on performance.

Question 3: Does Preference For A Strategy Vary With Experience? Does Program Accuracy Vary With Experience?
As expected, accuracy improves from 19% for the novice group to 49% for the intermediate group to 83% for the advanced group ($\chi^2 = 61.3$, $p < 0.001$). We also examined whether the difference in performance between Pascal and Pascal L was affected by the level of experience of the group (see Table IV). The significance of level of experience and language type was only marginal,[6] suggesting that all levels of experience benefited equally from Pascal L.

We can also see a shift in strategy preference: the trend in the data in Table I suggests that the more experienced programmers were beginning to more consistently adopt a PROCESS/READ strategy. Finally, sensitivity to the strategy that implicitly underlies a language construct also seems to increase with experience. This trend can be seen by asking the following question of the data in Table V: what percentage of programmers used the strategy appropriate to the language (irrespective of language type and irrespective of program correctness)? The answers are that 58% (37/64) of the novices, 68% (61/89) of the intermediates, and 80% (42/49) of the advanced programmers employed the strategy appropriate to the lan-

[6] Novice vs. expert: $z = 1.62$, $p = .10$; intermediate vs. (expert + novice) $z = .91$, not significant. The interaction was analysed using an arcsin transformation (see [1] p. 368).

guage, e.g., a PROCESS/READ strategy for Pascal, and READ/PROCESS strategy for Pascal L ($\chi^2 = 10.20$, $p < 0.01$).

In summary, the data gathered in this study support the following claims:

- people's preferred cognitive strategy seems to be READ/PROCESS as opposed to PROCESS/READ, at least on problems of the sort used in this study.
- people can write correct programs more often using a language that facilitates their preferred cognitive strategy; and
- people's accuracy, sensitivity to underlying strategy, and preference for a particular strategy can shift with experience.

6. CONCLUDING REMARKS AND IMPLICATIONS

In this study, we have documented some of the difficulties that arise when a programming language construct requires a cognitive strategy that differs from the preferred strategy. In particular, we have focused on Pascal's **while** construct, and have shown that the strategy that underlies the correct use of that construct— a PROCESS/READ strategy—is clearly not the preferred strategy. Moreover, we have demonstrated the significant increase in performance that results when subjects are given a construct, e.g., the **loop... leave... again** construct, that facilitates the READ/PROCESS strategy, their preferred cognitive strategy. Clearly, care must be taken in generalizing our results: the task used in our study required only a small program and the subjects were not professional programmers. However, at a minimum, programming instruction needs to attend to the bugs and misconceptions that arise in this sort of situation. For example, students need to be made aware of the existence of the different strategies. Also, students need to be taught explicitly about the characteristics of problems that require the unusual strategy. In this way, students might be made more conscious of the potential pitfalls. It would be an interesting experiment to see if, with such explicit instruction, the number of bugs and misconceptions could be reduced.

Another observation can also be drawn from our study: students write programs correctly more often using a construct that permits them to exit from the middle of the loop. Strong claims have been made against this sort of construct as it is argued that one should exit a loop from the top or the bottom, not the middle. For example, Ledgard [5], argues that *"forcing loop exits to the beginning or end of a loop in the long run is superior. In particular, it forces the programmer to state the loop-terminating condition at the entrance to or exit from the loop. While this may be more difficult to write initially, in the long run it forces a good program structure and leads to more maintainable programs. Exiting from the middle of a loop, while convenient, may readily lead to confusing program logic."* It is further claimed that the readability of a program is hampered if exits from the middle of the loop are allowed. (See also [2, 13].) Our study did not examine the readability claim, since we looked only at program generation. However, a series of studies by Sheppard et al. [8] suggest that in fact a construct that permits an exit in the middle does not interfere with readability. They compared a "strictly structured" looping construct [2], that did not permit an exit from the middle with a "naturally structured" construct, that did permit an exit from the middle, and found that their programmers showed no reliable difference in performance between these two constructs on a program comprehension task. They also examined these constructs in modification and debugging tasks and again found no statistical difference between the programmers' performance. Moreover, their studies were with professional programmers. Thus, there appears to be empirical evidence that an exit from the middle of the loop is not as harmful as was conjectured.

Finally, our study suggests that insights can come from looking beyond the syntax and semantics of language constructs to the cognitive demands that those constructs place on programmers. This appears to be especially relevant to the training of non-professional programmers since programming is a demanding skill and unnecessary hurdles serve only to complicate the learning process further. By being sensitive to the problem-solving skills that people bring to programming, and to those required by programming, we might be better able to assist people in making the necessary transitions.

Acknowledgments. The authors would like to thank the reviewers of this paper for their constructive comments. We would also like to thank Lewis Johnson, Steve Draper, Jerry Leichter, Jim Galambos, and John Black for their helpful comments on earlier drafts of this paper.

REFERENCES

1. Bishop, Y.M.N., Feinberg, S.E., and Holland, P.W. *Discrete Multivariate Analysis: Theory and Practice.* MIT Press, Cambridge, Massachusetts, 1975.
2. Dijkstra, E.W. Notes on structured programming. In *Structured Programming*, O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, (eds.), Academic Press, New York, 1972.
3. Knuth, D. Structured programming with GOTO statements. *Comput. Surv. 6*, 4 (December 1974), 261–301.
4. Miller, L.A. Natural language programming: styles, strategies, and contrasts. *IBM Syst. J. 20*, 2 (1981), 184–215.
5. Ledgard, H.F. and Marcotty, M. A genealogy of control structures. *Commun. ACM 18* 11 (1975), 629–638.
6. Ledgard, H., Whiteside, J., Singer, A., and Seymour, W. The natural language of interactive systems. *Commun. ACM 23*, 10 (1980), 556–563.
7. Papert, S. *Mindstorms, Children, Computers and Powerful Ideas.* Basic Books, Inc., New York, 1980.
8. Sheppard, S.B., Curtis, B., Milliman, P., and Love, T. Modern coding practices and programmer performance. *Computer* (December 1979), 41–49.
9. Soloway, E., Lochhead, J., and Clement, J. Does computer programming enhance problem solving ability? Some positive evidence on algebra word problems. In *Computer Literacy*, R. Seidel, R. Anderson, B. Hunter (eds.), Academic Press, New York, 1982, pp. 171–215.
10. Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. What do novices know about programming? In *Directions in Human-Computer Interactions*, B. Shneiderman and A. Badre (eds.), Ablex, Inc., 1983.
11. Wegner, P. Programming languages—Concepts and research directions. In *Research Directions in Software Technology*, MIT Press, Cambridge, Massachusetts, 1979.
12. Welty, C. and Stemple, D. Human factors comparison of a procedural and a nonprocedural query langauge. *ACM Trans. Database Syst. 6*, 4 (1981), 626–649.
13. Wirth, N. On the composition of well-structured programs. *ACM Comput. Surv. 6*, 4 (1974).

CR Categories and Subject Descriptors: D.m[**Software**]: Miscellaneous—*software psychology*; D.3.3 [**Programming Languages**]: Language Constructs—*control structures*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms: Human Factors

Additional Key Words and Phrases: cognitive factors in programming, cognitive strategies, looping constructs.

Received 7/81; revised 7/82; accepted 9/82