June 1981

Cognitive Factors in Programming: An Empirical Study of Looping Constructs

Elliot Soloway*
Jeff Bonar*
Kate Ehrlich**

*Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

**Psychology Department
University of Massachusetts
Amherst, Massachusetts 01003

COINS Technical Report 81-10

This work was supported by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant No. MDA903-80-C-0508.

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 which tests the following hypothesis:

> A programming language construct which has a closer "cognitive fit" with individuals' natural problem solving strategy will be easier to use effectively.

After analyzing novice Pascal programs which 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 i/process i strategy), (2) on the ith pass the ith element is processed and the next-ith element is read (the process i/read next-i strategy). We feel the former to be the more natural strategy, especially for novices. Moreover, we argue that the latter strategy is associated with the appropriate use of the Pascal _while_ construct. In contrast, we feel that the Ada-like _loop...leave...again_ construct, by allowing an exit from the middle of the loop, facilitates the former, read i/process i strategy.

In our experiment, subjects were asked to solve a problem which required a simple loop program. We sought to identify people's natural problem solving strategy and to compare the above two looping constructs. Our results demonstrate a strong preference for a read i/process i strategy. When writing a simple looping program, those using the _loop...leave...again_ were more often correct then were those using the standard Pascal loop constructs. These results were obtained for advanced as well as for novice and intermediate level programmers.

1.0 Introduction

The need for the public to be literate in computing is rapidly gaining acceptance. One aspect of such literacy is programming. While we do not believe that everyone need become a professional programmer, it is increasingly important to be able to describe how the computer is 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 novices' problem solving skills, then a barrier has been erected to the use of computers by novices.

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. [1980] compared an editing language whose syntax was based on English with an a standard notational editing language, and found that the English based language was preferred by the subjects and facilitated better performance. Welty and Stemple [1981] compared a procedural query language with a non-procedural query language using novices; they found that subjects performed at a higher level of accuracy with the procedural language when writing moderate to difficult queries. Miller and Becker [1974] explored individuals' natural problem solving strategies in their study of how subjects expressed a procedure in natural language. Shneiderman [1980], in a recent book on the field of "software

psychology," presents a cognitive model of the programming process. We report here on experiments which study the relationship between the natural problem solving strategies of individuals and various programming language looping constructs.

2.0 Two Strategies: Read I/Process I Vs. Process I/Read Next-I

Consider the following problem:

> Write a program which 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 not tricky nor esoteric; one would think that this problem would be easy for students at the end of a semester course on Pascal. In fact, students did surprisingly poor on this and related problems [1].

In this problem the loop is dependent on the the variable which holds the new values successively read in. [2] The Pascal loop construct most appropriate to this type of problem is the while construct. In Figure 1 we depict the stylistically correct Pascal solution to this problem.

Stepping back from the code, the strategy which this program

---

[1] Only 38% of the students were able to write a correct program for this problem; this test was given to students on the last day of classes after a semester course on Pascal programming [Soloway, et al. 1981].

---

[2] Loops can also be dependent on variables playing other roles, e.g., the counter, the running-total. We discuss this issue at greater length in Section 6.0. See also [Soloway, et al. 1981].

embodies can be characterized as:

```
read  first-value
while test ith value
    process ith value
    read    next-ith value
```

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 behind the read; on the ith pass through the loop, the ith value is processed and then the ith + 1 value is read in. We call this strategy "process i/read next-i".

Intuitively, we felt this coding to be unnecessarily awkward and confusing. A more "natural" coding strategy would be to read the ith value and process it on the ith pass through the loop; we call this the "read i/process i" coding strategy. Figure 2 depicts Pascal programs which use while and repeat loops and implement the read i/process i strategy. These are actual student programs produced in our initial experiment. The programs in Figures 2a and 2b use an embedded if statement to effect the read i/process i strategy. In the former case a boolean variable is used to control the outside while loop, while in the latter case the same test is performed twice. In each case the program contains code which is extraneous to the actual problem. In Figure 2c we see a program in which a repeat loop is used to implement the read i/process i strategy; the stop value is simply subtracted from the total. All three programs, however, would not be considered to be in good style. The goto

was not taught to students in this class, thus it does not appear in programs. {3}

Knuth [1974] and Wegner [1979] have also pointed out the awkward coding strategy that is required in the above sorts of problems using the while construct. Knuth suggests that Dahl's loop is a better construct, i.e., loop; S; while not B: T; repeat;, where S and T are zero or more statements and B is the test condition. In Ada, the new DOD language, one can use a similar construct, i.e., loop; S; exit when B; T; end loop. In what follows we shall discuss a syntactic variant of the Ada loop, namely, the loop...leave...again construct.

The read i/process i and the process i/read next-i strategies can both be encoded in the loop...leave...again construct. If S is empty, then the test is at the top of the loop, thereby creating a standard while loop. However, the loop...leave...again construct clearly facilitates the read i/process i strategy better than the standard while construct. In Figure 3 we depict the problem described above encoded using Pascal-L. Note that unlike the programs in Figure 2, no extraneous machinery is required to encode the read i/process i strategy. In what follows we will refer to Pascal with loop...leave...again as the only looping construct as Pascal-L.

{3} Almost none of the students, introductory or advanced, tested to see if count was zero. For the purposes of our experiment, we did not deduct for this omission.

3.0 Hypotheses

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

Pascal, with the normal _while_ and _repeat_ constructs, can be used to implement either the read i/process i strategy or the process i/read next-i 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 i/read next-i strategy whereas Pascal-L facilitates a read i/process i strategy. Our claim, then, is that for the type of problems discussed above, people should find Pascal-L, a language that facilitates the read i/process i strategy, easier to program than Pascal, a language that facilitates process i/read next-i.

Our hypothesis leads us to ask three particular questions:

Question 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 paper.

Once having determined whether people will adopt a read i/process i or a process i/read next-i type of strategy when they think about the problem, we can go on to ask:

> Question 2: Will people write correct programs more often when using the language that facilitates their natural strategy?

Thus, if people use a read i/process i 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 i/process i strategy, as compared with 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:

> Question 3: Does natural strategy or program accuracy vary with experience?

4.0 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 is reproduced in Figure 4. Here we asked them to write a plan which would solve the stated problem. The second part of the test is depicted in Figure 5. Half the students were asked to write a Pascal program which

solved the problem, while the other half were asked to solve the problem using Pascal-L. Each group was given a 1 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, while the Pascal group was given a one page description of the for, repeat, and while constructs. The one page on the loop..leave...again construct of Pascal-L contained two examples; we were careful to include instances of using the if..leave which branched off the top of the loop (which is equivalent to a while) as well as in the middle. As much time as needed was given to students taking this test.

This test was administered to three different groups: novices, intermediates, and advanced. Novices were students currently taking a first programming course in Pascal. The test was administered after the students had been taught about and had experience with the while loop; this occurred 3/4 of the way through the semester. Intermediates were students currently taking a second course in programming, e.g., either a data structures course using Pascal, or an assembly language course. The advanced group were juinors and seniors in systems programming and programming methodology courses.

5.0 Results

With respect to Question 1 above, let us first look at the performance of our three populations on the part of the test in which we asked people to write down their plans for solving the problem. {4} The results are shown in Table 1 and they clearly indicate that all three populations demonstrated a strong preference for the read i/process i strategy, when it was possible to discern any strategy at all. Across all three groups, 82% of the students used the read i/process i strategy, while only 18% used the process i/read next-i strategy in their plans. Now consider Table 1b; there we show the strategy choice of students on the program, irrespective of language. Except for the advanced group, again we see a strong preference for the read i/process i strategy. That is, 73% of the subjects used the read i/process i strategy, while only 27% used the process i/read next-i strategy. Thus, it can be argued that people's natural strategy, in terms of our two alternatives, is for read i/process i rather than process i/read next-i.

Question 2 is addressed by the data in Table 2; there we depict the accuracy of the program as a function of language. It can be seen that people are significantly more likely to write a correct program using Pascal-L, which facilitates a read i/process i strategy, than using Pascal. {5} 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 overall high

performance with respect to correctness of the Pascal-L users.

Some interesting findings also emerge when we consider the performance for each of the three groups of subjects (Question 3). As expected, accuracy improves from 19% for the novice to 49% for the intermediate group to 83% for the advanced group. However, some changes in the way people think about the problem can also be detected. In Table 1 it can be seen that the novices have many more miscellaneous responses relative to the intermediate group; that is, typically their plans were too sketchy to identify a strategy. The advanced group, however, had a greater proportion of plans using a process i/read next-i strategy compared with the intermediate group. This observation suggests that experience might be influencing the way people naturally think about the problem. Moreover, the reason that the advanced group did not seem to prefer the read i/process i strategy on their programs (Table 1b), can be seen by examining Table 3, where the strategy choices are shown broken down by langauge. The advanced group preferred a read i/process i strategy when using Pascal-L, and a process i/read next-i strategy when using Pascal. Apparently, they seemed to recognize which strategy a language facilitated. It should be stressed however, that for all groups, a read i/process i strategy did

---

(4) Half of the intermediate group were asked to write a plan, and half were asked to write a flowchart. We found no significance difference in their choice of strategies, however. For reporting purposes, we have thus combined the results of these two groups.

---

(5) Although the novices show the same effect as the other groups, the difference in their performance is not statistically significant due to the large number of incorrect programs.

predominate in their plans. Moreover, all groups were more accurate when asked to write a program in the language (Pascal-L) that facilitated this strategy.

6.0 Pervasiveness Of Process I/Read Next-i Strategy

We shall now extend the foregoing analysis along two dimensions: a cognitive one, and a programming one. First, why do people seem to prefer the read i/process i strategy over the process i/read next-i strategy? We speculate that it is a question of synchrony. In the read i/process i strategy, one gets an entity and acts on it in the same pass; in the process i/read next-i strategy, one acts on the entity which was gotten on the previous pass. In otherwords, the latter strategy forces processing to be "out of sync" with the fetching action, while the former strategy allows the fetching and the processing to be "in sync."

Second, we feel that read i/process i (and process i/read next-i) can be generalized to "get a value/process a value" (and "process a value/get next-value"). That is, read functions as a way to "get a new value", and there are other ways to achieve this goal, e.g., i := i+1 also generates a "new value" for a variable acting as a counter or index.

Consider, then, the following program fragment:

```
1.  sum := 0
2.  i := 1
3.  while a[i] <> -1 do
4.     begin
5.        sum := sum + a[i]
6.        i := i+1
7.     end
```

This program fragment sums up the numbers in an array until it reaches the number -1; note that the number -1 is not included in the sum. We claim that this program too embodies the process i/read next-i strategy. However, instead of read, we have the assignment statement in line (6) which repeatedly generates a new value for the index variable i; the process component is line (5). Thus, the notion of a "read" can be generalized to a "get a value."

In the above examples, the process i/get next-i strategy has always been associated with a while loop program. Is this necessarily so? We believe so; in fact we claim that:

> by its very nature, the appropriate use of the while loop construct will always be associated with a process a value/get a value strategy.

This is a strong claim; it rests on understanding the contexts in which the while loop is the appropriate loop construct.

We feel that textbooks typically give a "surfacy" distinction between the while and repeat loops. For example,

> The principle difference is this: in the WHILE statement, the loop condition is tested before each iteration of the loop; in the REPEAT statement, the loop condition is tested after each iteration of the loop.
>
> Findlay and Watt [1978]

Or, from the Pascal User's Manual,

> If the number of repetitions is known beforehand, i.e., before the repetitions are started, the for statement is the appropriate construct to express this situation; otherwise the while or repeat should be used. ... The statement [in a while body] is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement is not executed at all. ... The sequence of statements between the symbols repeat and until is repeatedly executed (at least once) until the expression becomes true.
>
> Jensen and Wirth [1974]

The deeper distinction, however, revolves around when the test variable can reasonably be tested. Thus, we would distinguish between the two loop constructs by saying:

> If the test variable will have a meaningful value as the loop is entered, i.e., a value that could prevent the loop from being executed even once, then a while loop is appropriate. If, however, the first meaningful value of the test variable is assigned to it during the loop, then a repeat loop is the appropriate iteration construct.

In the above problems, the New_Value Variable (e.g., the read variable, or the index variable) had meaningful values as the loop was entered; thus a while loop was the most appropriate control structure.

Now consider the following program, which adds up the integers until their sum is greater than 100.

```
1.  sum := 0
2.  i := 0
3.  while sum < 100 do
4.     begin
5.        i := i+1
6.        sum := sum+i
7.     end
```

This program looks like a counter-example to the above claim, since it contains a while loop and it appears to employ a get i/process i strategy (line 5 is the get i component, while line 6 is the process i component). [6] However, we feel that the use of the while construct, though not incorrect, is inappropriate.

Consistent with our discussion of the difference between a while loop and a repeat loop, a better solution to this problem would be:

```
sum := 0
i := 0
repeat
   i := i+1
   sum := sum+1
   until sum ≥ 100
```

We call this type of loop a "Running-Total Controlled Loop", since the test variable is a running-total variable, i.e., the variable sum accumulates a total [Soloway, et al, 1981]. A repeat loop is the appropriate construct here, since the test variable in this type of problem is assigned a meaningful value inside the loop, and thus the test should be made after the

---

[6] This problem sums up the natural numbers in order, until the sum is greater than 100. The same analysis would apply to a program in which the numbers were read in.

assignment.

The above discussion gives rise to a dilemma. On the one hand, if one argues that our distinction between the while and repeat loops is idiosyncratic, and thus that the above program should be coded using a while loop, one thereby throws grave doubt on the status of the repeat construct. Either the repeat construct is appropriate in certain cases, which are different than those when the while is appropriate, or these two constructs are redundant and interchangable. If the latter, this apparently flies in the face of the design goal of Pascal, namely:

> "keeping the number of fundamental concepts reasonably small" (Abstract, Wirth [1971])

Also, one wonders why so much emphasis is placed on teaching both constructs, if in fact, one will do.

On the other hand, if the while loop has uses distinct from the repeat loop, then our data suggest that both novices and intermediates will have significant difficulty learning to use the while construct, since it is associated with the apparently unnatural process a value/get a value strategy.

7.0 Implications

The design of a programming language is influenced by a myriad of factors. Our work suggests that cognitive factors need to receive more attention; if one wants to design a language which novices and casual programmers can learn to use, one needs to take their natural problem solving skills and strategies into

consideration. A mismatch between the language's constructs and people's natural abilities can create a barrier to learning.

We are not claiming that programming languages for professionals should necessarily cater to the needs of the novice. In fact, one might even want to claim that the difference between a professional and a novice is that the professional has learned to constrain his or her natural tendencies, and it is the goal of programming languages to aid in this re-training process. If so, then it is still important to identify explicitly the specific tendencies which need to be re-trained. Moreover, one needs to then see -- by empirical testing -- if a particular programming language construct does in fact constrain the tendency in question.

Our results are also relevant to computer science eduation and computer literacy. In particular, the test problem we used is not, by any stretch of the imagination, tricky or esoteric. If students can't write an 8 line program which sums and averages a sequence of numbers at the end of a programming course --- or in the middle of their second course --- then something is drastically wrong. One possible explanation is that the basic programming issues are not so simple; our results suggest that students have significant difficulty translating their natural understanding of looping to meet the constraints of Pascal. Possibly, if instruction were to explicitly address itself to this type of problem, learning and understanding would be facilitated.

Finally, the literature is repleat with claims for the readability, debuggability, learnability, etc. of programming language X (see Soloway, et.al. [1981]). Without empirical research, of the sort described here, these claims remain on the same footing as mystically divined truths. Moreover, if we are going to take seriously the notion of a computer literate public, and if, as it seems, people are going to need to program to some degree, then computer science must play closer attention to the cognitive factors involved in programming.

Acknowledgements

We gratefully acknowledge the contributions of Klaus Schultz and Chuck Schmidt, both of whom suggested important improvements in our experimental design. And we thank Janet Turnbull, who as usual helped conquer the various logistical hurdles involved in carrying out this type of research.

8.0 References

Findlay, William and Watt, David (1978) Pascal: An Introduction to Methodical Programming, Computer Science Press, Inc., Potomac, Maryland.

Jensen, Kathleen and Wirth, Niklaus (1974) Pascal User's Manual and Report, Springer-Verlag, New York.

Knuth, Donald (1974) "Structured Programming with go to Statements", Computing Surveys, Vol. 6, No. 4, December.

Ledgard, H., Whiteside, J., Singer, A., Seymour, W. (1980) "The Natural Language of Interactive Systems," in CACM, Vol. 23, No. 10.

Miller, L., Becker, C. (1974) "Programming in Natural English," IBM Technical Report RC 5137, November.

Shneiderman, B. (1980) Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., Cambridge, Mass.

Soloway, E., Bonar, J., Woolf, B., Barth, P., Rubin, E., and Ehrlich, K. (1981) "Cognition and Programming: Why Your Students Write Those Crazy Programs," Proceedings of the National Educational Computing Conference, Texas, 1981.

Wirth, Niklaus (1971) "The Programming Language Pascal," Acta Informatica 1, 1 pp. 35-63.

Wegner, Peter (1979) "Programming Languages - Concepts and Research Directions," in Research Directions in Software Technology, P. Wegner (Ed.), MIT Press, Cambridge.

Welty, C. and Stemple, D.W. (1978) "Human Factors Comparison of a Procedural and a Nonprocedural Query Language," to appear in Trans. on Database Systems.

```
program Student6_Problem3;
        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;
        Average := Sum / Count;
        Writeln (Average)
        end.
```

FIGURE 1

A Stylistically Correct Pascal Program

```
program Student7_Problem3;
      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.
```

FIGURE 2a

Using a Boolean Variable and a Nested Conditional
To Effect a read i/process i Strategy

```
program Student14_Problem3;
      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.
```

FIGURE 2b

Using a Nested Conditional and a Repeated Test
To Effect a read i/process i Strategy

```
program Student16_Problem3;
      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
      end.
```

FIGURE 2c

Using a repeat Loop and Backing Values Down
To Effect a read i/process i Strategy

```
program Pascal-L;
   var Count, Sum, Nu_Value: integer;
          Avg : real;
   begin
   Count := 0;
   Sum := 0;
   loop
        read (Nu_Value);
        if Nu_Value = 99999 then leave;
        Sum := Sum + Nu_Value;
        Count := Count + 1;
        again
   Avg := Sum/Count;
   Writeln (Avg);
   end
```

FIGURE 3

The Averaging Problem using Pascal-L:
Pascal Augmented with the loop...leave Construct

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 YOU 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

First Question; Asked of All Subjects

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 ... *)
```

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 repitition 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.

FIGURE 5a

Half of Subjects Given Standard Pascal Version

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 ... *)
```

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 then do the loop all over again.

In other words, as long as the expression stays FALSE, all the statements between 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);
    VAR i :  INTEGER;
    BEGIN
    i := 1;
    LOOP
         writeln(i);
         IF i >= 10 LEAVE;
         i := i + 1
    AGAIN
END.
```

```
PROGRAM example2(output);
    VAR i :  INTEGER;
    BEGIN
    i := 1;
    LOOP
         IF i > 10 LEAVE;
          writeln(i);
         i := i + 1
    AGAIN
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 you cooperation.

FIGURE 5b

Half of Subjects Given Pascal-L Version

	READ/ PROCESS	PROCESS/ READ	N	MISC.{1}	SIG.
NOVICES (116)	82%	18%	39	71	.000 *
INTERMEDIATES (112)	91%	9%	90	22	.000 *
ADVANCED (52)	67%	33%	48	4	.02 *

TABLE 1a

Strategy on Plan

	READ/ PROCESS	PROCESS/ READ	N	MISC.{1}	SIG.
NOVICES (116)	89%	14%	64	52	.000 *
INTERMEDIATES (112)	72%	28%	90	22	.000 *
ADVANCED (52)	60%	40%	49	3	.199

TABLE 1b

Strategy on Program

{1} This column depicts the number of individuals for which we could not identify a strategy in their plan or program. The percentages under the read i/process i process i/read next-i columns do not take this figure into account.

* -- Statiscally Significant

	CORRECT	INCORRECT	N	SIG.
		NOVICES (116)		
Pascal-L	24%	76%	58	.155
Pascal	14%	86%	58	
		INTERMEDIATES (112)		
Pascal-L	61%	34%	59	.008 *
Pascal	36%	64%	53	
		ADVANCED (52)		
Pascal-L	96%	4%	26	.010 *
Pascal	69%	31%	26	

TABLE 2

Program Correctness With Respect to Language

* -- Statistically Significant

	READ/ PROCESS	PROCESS/ READ	N	MISC.{1}	SIG.
		NOVICES (116)			
Pascal-L	97%	3%	30	28	.02 *
Pascal	77%	23%	34	24	
		INTERMEDIATES (112)			
Pascal-L	86%	14%	50	9	.0008 *
Pascal	54%	46%	39	14	
		ADVANCED (52)			
Pascal-L	92%	8%	26	0	.0001 *
Pascal	22%	78%	23	3	

TABLE 3

Strategy on Program With Respect to Language

{1} This column depicts the number of individuals for which we could not identify a strategy in their plan or program. The percentages under the read i/process i process i/read next-i columns do not take this figure into account.

* -- Statistically Significant