To appear in: Directions in Human-Computer Interactions
B. Shneiderman and A. Badre, Editors.
Ablex Publishing Co., 1982

# What Do Novices Know About Programming?

January 1982

***Research Report #218 (Yale)

Elliot Soloway*
Kate Ehrlich*
Jeffrey Bonar**
Judith Greenspan**

***Substitute for COINS Tech Report #81-18

* Department of Computer Science
Yale University
P.O. Box 2158
New Haven, Connecticut 06520

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

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.

## Table of Contents

## 1. Introduction

In a recent book, Shneiderman (1980) points out that:

> "For every professional programmer there are probably ten occasional programmers who write programs for scientific research, engineering development, marketing research, business applications, etc. And finally there are a rapidly growing number of programmer hobbyists working on small business, personal and home computing applications."

In other words, there will be a great number of people who will program computers in carrying out their daily activities. For such casual programmers, initial difficulties in learning a programming language may become a permanent barrier to their continuing interaction with computers. Clearly, programming languages, text editors, command languages, and other applications programs can be designed to better facilitate human interaction. However, in order to build such systems tailored to the needs of the non-professional, we must identify the needs of such individuals. In this volume, there are papers that address themselves to just this issue, e.g., Dumais and Landauer (this volume) explore the factors involved in learnable command languages. Our focus is on programming: what difficulties do non-professional programmers have in learning to program, and what are the sources of these difficulties.

In this paper we will present the results of an exploratory study we conducted with non-professinal programmers who were asked to write programs as solutions to a set of problems. The objective was to study the bugs --- errors in programs --- and misconceptions --- misunderstandings in the minds of the novice programmers. We study bugs because they serve to expose the specific knowledge deficiences and/or confusions which people have [Brown and Burton 1978]. The key to understanding the misconceptions of non-professionals is a theory of programming knowledge, i.e., the knowledge which expert programmers have about programming. We assume that misconceptions are some variant of such expert knowledge. As we will soon see, we have developed a preliminary theory of the high level, plan knowledge which expert programmers appear to have and use. We will describe how this theory guided us in the design of the study reported here, and how it provided the basis for interpreting the bugs in non-professionals′ programs. In this study, then, we attempt to identify the needs of novice programmers by understanding the source of their difficulties. While not the focus of this paper, we do make suggestions as to how these difficulties might be overcome.

## 2. An Example

Consider the following problem:

> Write a program which repeatedly reads in integers until their sum is greater than 100. After reaching 100, the program should print out the average of the integers read in.

A program that solves this problem would need a number of components:

1. a way of reading a new integer
2. a way of accumulating a running total
3. a way of counting the number of integers read, in order to compute the average
4. a loop which repeatedly performs the above operations, and in addition has a terminating condition to prevent numbers being read in indefinitely
5. a way to calculate the average after the appropriate number of integers have been read in
6. a way to print out the final answer.

A Pascal program which correctly solves this problem is given in Figure 1. Although this simple example illustrates many basic notions of programming, such as looping, testing and operating on variables, and reading/writing, the above problem should not be difficult, even for novices. It requires no esoteric language constructs, nor does it require particularly clever code. However, when we gave this and similar problems to students in an introductory Pascal programming class, only 44% of them were able to write correct programs.

```
program Example;
    var Count, Sum, Next : integer;
        Average : real;
    begin
    Count := 0;
    Sum := 0;
    repeat
        Read (Next);
        Sum := Sum + Next;
        Count := Count + 1
    until Sum > 100;
    Average := Sum / Count;
    Writeln ( ´The average is : ´, Average)
    end.
```

Figure 1: A Sample Program

The above is a program which calculates the average of numbers which are repeatedly read until their sum exceeds 100.

To get some idea of where students were going wrong, we broke the program down into its component parts and scored a subject´s program with respect to the correctness of each component part. We then generated a profile of the typical program by integrating the code fragments most commonly written by the subjects for each of the components in the program. The resulting --- and correct --- program was the one shown in Figure 1. Insofar as this method of analysis gives an accurate composite picture of a novice programmer, it suggests that there is no single component of a Pascal program that is troublesome for the majority of students.

The errors in the students´ programs stemmed from two sources: students had difficulty with the syntax and semantics of the various programming language constructs, and they had difficulty determining which constructs to use and how to coordinate them into a unified whole. Examples of the former type include: incorrectly forming assignment statements; explicitly --- and wrongly --- incrementing the index variable in a Pascal for loop, etc. Examples of the latter type include: using an inappropriate loop construct in a problem; failing to initialize variables properly, etc.

Some attention has been paid to identifying and cataloging errors of the first sort (see Gannon [1978]). In contrast, this paper deals primarily with errors of the second sort, i.e., those which reflect a confusion or lack of knowledge about the pragmatic and functional factors involved in using various programming language constructs. In order to identify specific pieces of knowledge which are lacking or not well understood, we will compare the knowledge of expert programmers with that of non-expert programmers. Shneiderman [1976] and Adelson [1981] have shown that experts seem to use high level knowledge to understand programs, while novices tend to focus on the specific statements employed in a program. Building on this work, it is our goal to identify _specific_ knowledge differences between experts and non-experts, and examine how different levels of knowledge affect the performance of non-experts. To this end, we have identified a number of experts′ high level plans, and have used them to guide our empirical work with non-experts; the study described in this paper is the first result of that enterprise.

## 3. Using Expert Knowledge As A Guide

We believe that experts use more than just knowledge of the syntax and semantics of a programming language when they write programs to solve problems. Expert programmers have and use high-level, plan knowledge to direct their programming activities. A plan is a procedure or strategy in which the key elements of the process have been abstracted and represented explicitly.[1] Faced with a new problem, an expert retrieves plans from his/her knowledge base which have proven useful in similar situations, and then weaves them together to fit the demands of the new problem.

[1] In this discussion, we will gloss over many technical aspects of the theory of planning. For our purposes, the commonsense notion of "plan" is sufficient. The interested reader can follow these issues more carefully in texts such as Nilsson [1980].

We have identified[2] a set of plans that we believe experts use when solving problems of the sort typically encountered in introductory programming courses (see Table 1).

Problem 1. Write a program which reads 10 integers and then prints out the average. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

Problem 2. Write a program which repeatedly reads in integers until their sum is greater than 100. After reaching 100, the program should print out the average of the integers entered.

Problem 3. 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, is should not count the final 99999.

Table 1: Three Simple Looping Problems

While these problems are relatively simple, their solutions illustrate many of the basic notions of programming. The problem given at the beginning of Section 2, uses a few repetitive actions involving simple arithmetic operations. The plan commonly used to solve this problem, the Running_Total Loop Plan, describes the various components needed in a program in order to compute a sum.

In order to be precise about the plan knowledge which expert programmers have, we decided to encode this knowledge in a "formal language." That is, we have borrowed a language for representing knowledge from Artificial

[2] The authors of this paper have served as the expert programmers in this enterprise. We are currently conducting group tests and gathering video-taped protocols with a number of other expert programmers in order to evaluate our claims.

Intelligence called "frames" [Minsky 1975].[3] This style of representing knowledge has two properties which we find particularly appropriate to our task:

1. Plans are not represented as atomic entities, but rather are represented as knowledge packets which have rich internal structure. A plan is encoded as a frame, which has slot types and slot fillers.
2. Plans are explicitly linked together by various types of relationships; these relationships are encoded as types of arcs.

Below, we will expand on the relevant structural properties of the frame representation and also, we will describe the specific content of our theory of programming, which is encoded in the frame representation.

As we indicated above, we have identified and represented a fragment of an expert´s programming knowledge; the frames used to represent this knowledge are shown in Figure 2. These frame representations of loop plans are composed of "slot types" (e.g., descriptions, variables) and "slot fillers." The former delimit general properties or aspects of loop plans, while the latter are specific values which capture the individual characteristics of a particular loop plan. For example, both the Running_Total Loop Plan and the Find_Satisfactory Loop Plan have the slot type "description," but the slot filler for each of these plans is different. In effect, a frame represents a template, which is customized to the particular features of the concept being represented.

Plans are linked together, a feature that both highlights the commonalities and differences of plans, and reflects hierachical ordering relations among

[3] We have implemented a version of this language in LISP on a computer, and have used the implemented language to actually represent the knowledge described below. This machine readable knowledge base is used by our intelligent tutoring system, MENO-II [Soloway et al. 1981c].

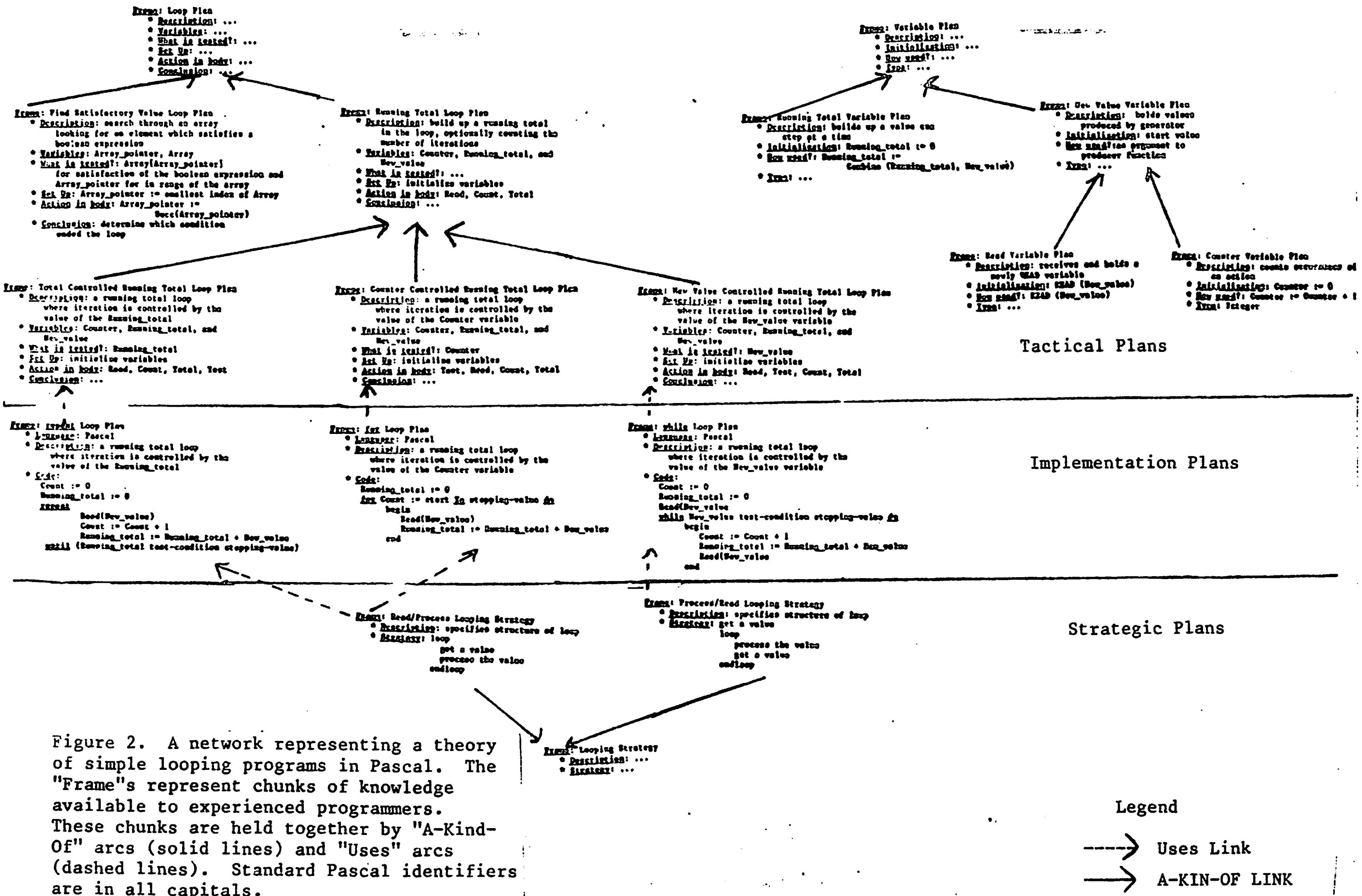
Frame: Loop Plan
• Description: ...
• Variables: ...
• What is tested?: ...
• Set Up: ...
• Action in body: ...
• Conclusion: ...
Frame: Find Satisfactory Value Loop Plan
• Description: search through an array looking for an element which satisfies a boolean expression
• Variables: Array_pointer, Array
• What is tested?: Array[Array_pointer] for satisfaction of the boolean expression and Array_pointer for in range of the array
• Set Up: Array_pointer := smallest index of Array
• Action in body: Array_pointer := Succ(Array_pointer)
• Conclusion: determine which condition ended the loop
Frame: Running Total Loop Plan
• Description: build up a running total in the loop, optionally counting the number of iterations
• Variables: Counter, Running_total, and New_value
• What is tested?: ...
• Set Up: initialize variables
• Action in body: Read, Count, Total
• Conclusion: ...
Frame: Variable Plan
• Description: ...
• Initialization: ...
• How used?: ...
• Type: ...
Frame: Running Total Variable Plan
• Description: builds up a value one step at a time
• Initialization: Running_total := 0
• How used?: Running_total := Combine (Running_total, New_value)
• Type: ...
Frame: New Value Variable Plan
• Description: holds values produced by generator
• Initialization: start value
• How used?: as argument to producer function
• Type: ...
Frame: Read Variable Plan
• Description: receives and holds a newly READ variable
• Initialization: READ (New_value)
• How used?: READ (New_value)
• Type: ...
Frame: Counter Variable Plan
• Description: counts occurrences of an action
• Initialization: Counter := 0
• How used?: Counter := Counter + 1
• Type: Integer
Frame: Total Controlled Running Total Loop Plan
• Description: a running total loop where iteration is controlled by the value of the Running_total
• Variables: Counter, Running_total, and New_value
• What is tested?: Running_total
• Set Up: initialize variables
• Action in body: Read, Count, Total, Test
• Conclusion: ...
Frame: Counter Controlled Running Total Loop Plan
• Description: a running total loop where iteration is controlled by the value of the Counter variable
• Variables: Counter, Running_total, and New_value
• What is tested?: Counter
• Set Up: initialize variables
• Action in body: Test, Read, Count, Total
• Conclusion: ...
Frame: New Value Controlled Running Total Loop Plan
• Description: a running total loop where iteration is controlled by the value of the New_value variable
• Variables: Counter, Running_total, and New_value
• What is tested?: New_value
• Set Up: initialize variables
• Action in body: Read, Test, Count, Total
• Conclusion: ...
Tactical Plans
Frame: repeat Loop Plan
• Language: Pascal
• Description: a running total loop where iteration is controlled by the value of the Running_total
• Code:
Count := 0
Running_total := 0
repeat
Read(New_value)
Count := Count + 1
Running_total := Running_total + New_value
until (Running_total test-condition stopping-value)
Frame: for Loop Plan
• Language: Pascal
• Description: a running total loop where iteration is controlled by the value of the Counter variable
• Code:
Running_total := 0
for Count := start to stopping-value do
begin
Read(New_value)
Running_total := Running_total + New_value
end
Frame: while Loop Plan
• Language: Pascal
• Description: a running total loop where iteration is controlled by the value of the New_value variable
• Code:
Count := 0
Running_total := 0
Read(New_value
while New_value test-condition stopping-value do
begin
Count := Count + 1
Running_total := Running_total + New_value
Read(New_value
end
Implementation Plans
Frame: Read/Process Looping Strategy
• Description: specifies structure of loop
• Strategy: loop
get a value
process the value
endloop
Frame: Process/Read Looping Strategy
• Description: specifies structure of loop
• Strategy: get a value
loop
process the value
get a value
endloop
Strategic Plans
Frame: Looping Strategy
• Description: ...
• Strategy: ...
Legend
Uses Link
A-KIN-OF LINK

Figure 2. A network representing a theory of simple looping programs in Pascal. The "Frame"s represent chunks of knowledge available to experienced programmers. These chunks are held together by "A-Kind-Of" arcs (solid lines) and "Uses" arcs (dashed lines). Standard Pascal identifiers are in all capitals.

Note: In order to minimize crossovers, we have not shown Variable Plans being used by Looping Plans.

the plans. Currently, there are two relationship links in our frame network: A_KIND_OF and USES. For example, the hierarchical A_KIND_OF relation connects both the Running_Total Loop Plan and the Find_Satisfactory_Value Loop Plan to the Loop_Plan, since the two former plans are specializations of the latter plan.

In addition to identifying abstract programming plans, we sought to understand the roles variables play in programs. Just as actors take on different roles in a play, variables take on different functions in a program. We have again used a frame representation to describe the knowledge associated with variable plans. Figure 2 depicts a number of roles which variables commonly play in simple looping problems. For example, one variable, the Counter, is used to count the number of elements being accumulated, e.g., Count:=Count+1. Similarly, the Running_Total Variable is used to accumulate a total, e.g., Sum:=Sum+Nu_Value. While both variables are updated using an assignment statement, they do not seem to be perceived, at least by experts, as "simply" variables. That is, a variable can be perceived as simply containing an arbitrary value or it can be perceived as containing values which have a specific purpose. The roles which we have described reflect the special purposes which values, and their variables, play in programs. In fact, the data we will present suggests that even novices perceive a Counter Variable as different from a Running_Total Variable.

In writing a program using a particular loop plan, some variables are needed; they are indicated in the frame for that particular plan in the "variables" slot. In order to get a more complete description of such a variable, one must go to the frame which describes it. Variables are associated with loop plans via the USES relationship in Figure 2. As with the plan frames, variable frames consist of slots and slot fillers which serve to describe the defining characteristics of each type of variable.

Figure 2 includes three other loop plans which are specializations of the Running_Total Loop Plan; one major difference among them is the value of the slot called "what is tested." For example, the loop plan appropriate to the problem given in Section 2 is the Total_Controlled Running_Total Loop Plan; the termination condition on the loop in that problem is when the Running_Total Variable reaches a certain value (i.e., 100). In contrast, consider the first problem in Table 1 for which the Counter_Controlled Running_Total Loop Plan is appropriate; this problem requests that exactly 10 numbers be read in and summed.

There is an additional layer of structure apparent in Figure 2. Namely, we have classified plans into 3 categories: (1) Strategic Plans and (2) Tactical Plans, which are both language independent plans, and (3) Implementation Plans, which are language specific plans. Strategic Plans specify a global strategy used in an algorithm. For example, the Read/Process Strategy specifies that the actions "read a value, then process it" are nested in a repetition loop. The program in Figure 1 illustrates the use of this strategy; the variable Next is first Read, then it is processed in the statement Sum := Sum+Next. It turns out that the Strategic Plans Read/Process and Process/Read are very important, and we will discuss them further in Section 6.3.

Tactical Plans specify a local strategy for solving a problem. For example, the Counter_Controlled Running_Total Loop Plan describes how a sum can be accumulated. This Loop Plan specifies an algorithm for solving a specific problem. Implementation Plans, on the other hand, specify language dependent techniques for realizing Tactical and Strategic Plans. Thus, the <u>for</u> Loop Plan is a technique for implementing the Counter_Controlled Running_Total Loop Plan in Pascal.

As we stated in the introduction, our goal is to identify the knowledge differences between expert and non-expert programmers. On the basis of the framework outlined in this section, we have designed and analyzed a broad study of the behavior of non-expert programmers. While the knowledge base in Figure 2 is clearly just a beginning, we are pleased and excited by the leverage it has already given us in understanding programming.

## 4. A Description of The Empirical Study

We wanted to test whether or not non-experts have the kind of plan knowledge described above. In particular, we were interested in seeing if non-experts could distinguish the appropriate context in which to use each of Pascal's looping constructs (the for, repeat, and while loops), i.e., do non-experts know when it is appropriate to use each of the three Pascal implementation loop plans. The problems displayed in Table 1 were constructed specifically to test this question. All three problems require that the average of a set of numbers be computed. However, each problem differs with respect to the condition for terminating the loop. Each of the three terminating conditions is implemented in Pascal using a different loop construct. For example, in the Problem 1 (Table 1) the value on which the loop will terminate is known beforehand; in this type of situation the most appropriate Pascal loop construct is the for loop [Wirth 1971]. Pascal was specifically designed to minimize redundant primitive commands.

> In view of its intended usage as a convenient basis to teach programming .... emphasis was placed on keeping the number of fundamental concepts reasonably small.
>
> Wirth 1971

Thus, the choice of the most appropriate loop construct in a given situation is not merely a matter of individual taste, but rather the choice can be based quite squarely on reason.

In Problem 1, the student is asked to write a program which reads in a specific number of integers, 10 in this case, and then computes their average.

As argued above, the Pascal loop construct most appropriate for this problem is the *for* loop. A correct solution to this problem using this construct is given in Figure 3.

```
program Student12_Problem1;
    var Count, Sum, Next : integer;
        Average : real;
    begin
    Sum := 0;
    for Count := 1 to 10 do
        begin
        Read (Next);
        Sum := Sum + Next
        end;
    Average := Sum / 10;
    Writeln ( 'The average is : ', Average)
    end.
```

Figure 3: The Appropriate Use of a *for* Loop

The above is a program which calculates the average of 10 numbers input by a user. It is a solution to Problem 1 of Table 1. Note that the above is an actual student's program which has been only minimally edited in order to facilitate readability.

While in Problem 1 the termination condition is a specific number of integers to be read, in Problem 2 it involves the variable which accumulates the sum, i.e., the program should stop reading when this variable is greater than 100. The Pascal loop construct most appropriate[4] to this problem is the *repeat* loop (see Figure 1). In Problem 3, the loop termination condition is a special value (99999 in this case) which, when read as one of the integers, signals a stop. The Pascal loop construct most appropriate to this problem is the *while* construct (Figure 4).[5]

[4] In Section 6.1 we will present more detailed arguments to support this claim.

[5] A better version of this program would also test that some data had been read in, and thus no divsion by zero was occurring. However, very few students actually included this type of test; the program in Figure 4, generated by a student, is typical of observed programs in this regard.

```
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 4: The Appropriate Use of a while Loop

The above is a stylistically correct solution to Problem 3 in Table 1. Note that it is an actual student's program which has been only minimally edited in order to facilitate readability.

Even though each Pascal loop construct was designed for a specific looping situation, a correct program can usually be written which does not use the most appropriate loop construct. In particular, the while loop is the most general construct and can be used to simulate the other two. However, using an inappropriate loop construct can become quite burdensome. In this case, the programmer must add extra code in order to compensate for the poor choice of loop construct. For example, Figure 5 shows two programs which correctly solve Problem 3. Although the while loop is the most appropriate construct for this problem, the students chose to use the repeat construct; in order to follow through with this decision, additional code was required (compare the programs in Figure 4 and Figure 5). The increased complexity of the code caused by the choice of an inappropriate loop construct would presumably increase the opportunity for error, and the data bear this out.

We administered the three problems described Table 1 as a non-credit quiz to two groups of students. The first group, consisting of 31 students in an a)

```
program Student146_Problem3;
    var Number, Count, Total : integer;
        Finished : boolean;
    begin
    Count := 0;
    Total := 0;
    repeat
        Read (Number);
        Finished := (Number = 99999);
        if not Finished then
            begin
            Count := Count + 1;
            Total := Total + Number
            end
    until Finished;
    Writeln (´The average is ´, Total/Count)
    end.
```

b)

```
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.
```

The above are two solutions to Problem 3, the while loop problem, of Table 1 Note that they are actual students´ programs which have been only minimally edited in order to facilitate readability.

Figure 5: Simulating Read/Process with a repeat Loop

introductory Pascal programming class, was tested in the final week of their summer session course. The second group included 52 students enrolled in a second course in programming (a data structures course which used Pascal); they were tested in the 10th week of a 16-week semester. The two groups will be referred to as "novices" and "intermediates" respectively.[6]

In using the terms "novice" and "intermediate", we do not mean to imply that the two groups necessarily differ either uniformly or consistently in their ability to program. As Moher and Schneider (1981) point out, people in the same class may differ considerably in their aptitude for programming and thus care must be exercised in evaluating experimental results. Our goal in using people with varying amounts of programming experience is to focus on the levels that people pass through as they learn to program. That is, we are interested more in the kinds of errors that people make, than in which group of people made the errors. Thus, we feel that this type of exploratory study serves to identify a broad range of interesting behaviors which can then be studied using more carefully controlled experimental techniques.

Scoring this type of production data is difficult.[7] Based on the programming knowledge described earlier, we developed a number of categories for coding the students' programs. For the most part, these categories reflected the functional characteristics of the program. For example, we scored a program with respect to the use/misuse of the Counter Variable, Running_Total Variable, etc (see Figure 2). This approach goes beyond more straightforward measures such as scoring a program with respect to the

[6] In their introductory courses both groups of students were taught and used all three Pascal loop constructs: while, repeat, for.

[7] Since students wrote the solutions on paper without access to a computer, we felt that it would be counterproductive to score as incorrect, programs which had only minor syntactic errors. Also, we did not mark as incorrect, programs in which there was no check for the count being zero before attempting to divide by it in the average calculation. We made this decision since only a small handful of programs did make the test.

use/misuse of a programming language construct (e.g., did the student correctly use an assignment statement). In effect, the latter technique would simply access the student´s understanding of the syntax and semantics of the programming language constructs which, while undeniably important, are not the only components of programming knowledge. For example, if we had just looked at assignment statement usage, we would have missed the distinction between the Counter Variable and the Running_Total Variable. Our goal in developing scoring categories based on programming plans was to tap into the students´ understanding of the functional characteristics of constructs, as well as their understanding of the syntax and semantics of those constructs.

## 5. Overall Performance

Table 2 displays the percentage of novices and intermediates who gave a correct solution to the problems. Unquestionably, the figures are low. Novices scored less than 50% on every problem, while the intermediates scored less than 60% on every problem.

| | Problem 1<br>for problem | Problem 2<br>repeat problem | Problem 3<br>while problem |
|---|---|---|---|
| Novices<br>Correct | 11/27<br>(41%) | 12/27<br>(44%) | 9 23<br>(3 %) |
| Intermediates<br>Correct | 29/57<br>(57%) | 28/49<br>(57%) | 18/43<br>(42% |

Table 2: Overall performance of novices and intermediates.

The majority of errors are not explainable simply as momentary "mental slips," such as failure to initialize a variable. Performance improved only slightly if errors of this sort were overlooked. For example, if we counted as correct all those programs in which the only error was a missing initialization, the percentage of correct programs (across all 3 problems) would only go up by 3% (to 45%) for the novices, and by 6% (to 59%) for the intermediates. Thus, we believe that the errors represented by this data reflect real misconceptions that students have about programming, and cannot be accounted for by simply saying "They made silly errors."

## 6. Bugs and Misconceptions: Looping

As stated earlier, our objective in this study was to build up a catalogue of common programming bugs and their underlying misconceptions. We begin this discussion with a description of bugs and misconceptions concerning the choice of looping construct and looping strategy.

### 6.1. The Looping Constructs In Pascal

Do novice and intermediate programmers distinguish among the three Pascal loop constructs and use them in the appropriate context? The answer, based on our data is, less than half the time. Table 3 lists the percentage of novices and intermediates who used each particular loop construct on each problem.

Let us first consider the performance of the novice group. Their performance data on Problem 1 are surprising. This problem is clearly a for loop problem; the for loop was designed for use in situations in which termination of the loop must occur when the Counter Variable exceeds some specific value, e.g., in Problem 1, the Counter Variable must range between 1 and 10. However, only 22% used a for loop; 37% used a repeat loop and 37% used a while loop, even though these constructs required the students to do more work by having to make explicit those operations done implicitly by the for loop (i.e., initialize counter, test counter for stopping, increment counter).

Problem 2 (see Table 1) was a repeat loop problem; the variable that controlled the loop, "sum," needed to be assigned a value in the loop before it could reasonably be tested. 59% of the novices used the repeat loop and 37% used the while loop; no novices used a for loop here. For Problem 3 (Table 1), the appropriate loop construct is while; the loop must not be

| | Problem 1<br>*for* problem | Problem 2<br>*repeat* problem | Problem 3<br>*while* problem |
|---|---|---|---|
| **Novices** | | | |
| Used *for* | 6<br>(22%) | -<br>(-) | 1<br>(4%) |
| Used *repeat* | 10<br>(37%) | 16<br>(59%) | 8<br>(35%) |
| Used *while* | 10<br>(37%) | 10<br>(37%) | 10<br>(37%) |
| Used other | 1<br>(4%) | 1<br>(4%) | 4<br>(17%) |
| Total attempting | 27 | 27 | 23 |
| Used appropriate construct and correct | 4/6 | 7/16 | 5/10 |
| Used inappropriate construct and incorrect | 7/21 | 5/11 | 4/13 |
| Overall correct | 11/27 | 12/27 | 9/23 |
| **Intermediates** | | | |
| Used *for* | 31<br>(61%) | 1<br>(2%) | -<br>(-) |
| Used *repeat* | -<br>(-) | 5<br>(10%) | 7<br>(16%) |
| Used *while* | 20<br>(39%) | 38<br>(78%) | 30<br>(70%) |
| Used other | -<br>(-) | 5<br>(10%) | 6<br>(14%) |
| Total attempting | 51 | 49 | 43 |
| Used appropriate construct and correct | 21/31 | 3/5 | 14/30 |
| Used inappropriate construct and correct | 8/20 | 25/44 | 4/13 |
| Overall Correct | 29/51 | 28/49 | 18/43 |

Table 3: Looping constructs used by Novices and Intermediates

executed if the controlling variable has a specified value and therefore the test must be placed at the head of the loop.[8] 37% of the novices used a *while* loop as opposed to 35% who used a *repeat* loop.

The intermediate group had a different pattern of performance. On Problem 1, the intermediates *did* seem to recognize it as being a *for* loop problem, in contrast to the novice group. However, on Problem 2, in which the *repeat* loop was the most appropriate construct, only a few used it. Rather, the overwhelming choice was the *while* loop. Counter-intutively, more novices used the appropriate loop construct in this problem than did intermediates. Intermediates again chose the *while* loop in Problem 3. However, given the overall usage of the *while* loop by the intermediates, it appears that the intermediates no longer have the *repeat* loop in their bag of tools; they simply use the *while* loop in all situations. Since the course in which they were tested was a "data structures" course, and did not focus on being tested on Pascal *per se*, they apparently had no motivation to remain familiar with the *repeat* loop; the *while* loop could always be coerced into service.

Based on this simple test, it appears that novices and intermediates do not distinguish among the three Pascal loop structures as experts might. A student we interviewed, when asked why he chose to use the *while* construct rather than one of the other two, responded:

> "When I don't know what is going on, I use a *while* loop."

At first, we found the results of the *for* loop problem (Table 2, Problem 1) counter-intuitive. After all, since the *for* loop does so much work automatically, we thought it would be the easiest to understand and use. On second thought, we decided that these automatic, implicit aspects of the *for*

[8] This, in turn, requires a curious coding structure which we examine in the next section.

loop might in fact be the problem. Since the for loop does a number of actions automatically, students might be uncertain about exactly how it works. To gain control over the program, students might choose a repeat or while loop and do the extra work to add the required looping machinery themselves.[9]

The difference between the repeat and while loops is subtle. Moreover, since one construct can simulate the other, the distinction is hard to enforce. We feel that textbooks significantly contribute to the confusion. 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]

> 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]

As opposed to these "syntax level" descriptions, we feel that the "deep structure" of these constructions should be emphasized. A better explanation is:

> 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

[9] Some support for this interpretation comes from the following fact: students in these classes were not taught the goto, and never constructed loops out of more primitive constructs. Thus, they might have been unsure of the ingredients of a loop, and uncomfortable with all the magic implicit in a for loop.

loop, then a *repeat* loop is the appropriate iteration construct.

In otherwords, there is a close connection between the loop test, the loop construct, loop plan, and the problem; the choice of loop construct is dependent on when the test ought to be performed. This observation is consciously reflected in the frame representation of looping plans in Figure 2.

The frames and slots in Figure 2 reflect the close connections between the test in the loop and the loop construct, and the connection between the test and the *problem*.

A final comment: the reader might feel that the distinction among the three loop types is not important *for a novice*. To some extent, we might agree. The problem is that courses and textbooks do teach all three loop types --- and students are expected to understand them. Given the above data, however, one wonders if teaching all 3 is a good strategy.

### 6.2. Choosing the Appropriate Loop Construct: Predictor of Success? No

Does choosing the appropriate loop construct (i.e., *for*, *repeat*, or *while* depending on context) for a given problem facilitate the production of a correct program solution? In effect, choosing the appropriate loop construct might "get you into the right ballpark." If a student does choose the correct loop construct, then

1. he might know more about programming, and thus be more likely to write a correct program,
2. and/or the loop construct itself might impose constraints that help the student to get the components of the loop correct.

Thus we ask: what was the pattern of performance for those students who *did* choose the appropriate loop construct? Keep in mind that the students in general did not use the loop construct that was appropriate to a problem, and their accuracy was low.

The students (novices and intermediates) who chose the appropriate loop construct were only slightly more likely to write correct programs (see Table 3). In other words, simply choosing the appropriate loop construct is not a predictor of program correctness. As we shall describe in the next section, choosing the appropriate looping strategy can be a predictor of correctness.

### 6.3. Choosing the Appropriate Looping Strategy: Predictor of Success? Yes

If the choice of loop construct is not a predictor of success then, is the choice of loop strategy? Recall from Figure 2 that a looping strategy is a type of Strategic Plan which specifies a global approach to a problem, while a looping construct specifies an Implementation Plan, and is the command in a specific programming language that a programmer selects to realize his strategy. In this section, we will describe two alternative looping strategies, and examine their importance for programming.

We will first focus our attention on the performance pattern of our subjects on Problem 3. While from Table 2 we see that the novices did equally poorly on all 3 problems, the performance of the intermediates is quite different; 57% correctly solved Problems 1 and 2, but only 42% correctly solved Problem 3. While Problem 3 is different from Problems 1 and 2, there is no a priori reason to expect students to perform any differently than they did on the other problems.

The stylistically correct solution to this problem, using a while loop [see Wirth 1974], requires a curious coding structure:

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

The loop must not be executed if the test variable has the specified value, and this value could turn up on the first read; thus, a read outside the loop is necessary in order to start the loop off. This results in the loop processing being "behind the read;" the ith input is processed and then the next ith (i + 1) is fetched. We call this looping strategy the "process i/read next-i" strategy.

We feel this looping strategy to be unnecessarily awkward and downright confusing (see also Knuth [1974]). A more "natural" looping strategy would be to read the ith value and then process it; we call this latter strategy the "read i/process i" looping strategy. Only through circuitous means can one encode a read i/process i strategy with a while loop in problems such as Problem 3. For example, one can embed an if test inside the loop which repeats the test in the while statement itself, or one can use a Boolean variable and assorted assignments (see Figure 6). Hence, the while loop facilitates the coding of a process i/read next-i strategy, but does not facilitate the coding of a read i/process i strategy.

Soloway et al. [1981a] described an experiment which demonstrated that students do in fact prefer to solve problems using a read i/process i strategy, as opposed to a process i/read next-i strategy. Also, their ability to write correct programs is enhanced if the loop construct facilitates the former strategy. In other words, people do have natural looping strategies, and we need to pay attention to the match between such strategies and the strategies facilitated by programming language constructs.

Given the existence of natural looping strategies, such as read i/process i, and their apparent importance, is the choice of looping strategy a predictor of success? The data in this experiment seem to indicate yes --- at least for Problem 3. Of the 24 students (novices and intermediates) who used a process i/read next-i strategy, 17 got the program correct, while of the 40 students who used the while construct, only 19 got the program correct. For the group choosing the process i/read next-i strategy, the difference between those getting the program correct and those getting it incorrect is

```
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.
```

The above is a correct solution for Problem 3 of Table 1. Note that it is an actual student's program which has been only minimally edited in order to facilitate readability.

Figure 6: Simulating Read/Process with a while Loop

significant (Sign test: N = 24, x = 7, p < 0 .03 ), while the corresponding difference for the group who chose the while loop is not significant (n = 40, x = 19, p > .10). Choice of appropriate strategy (i.e., process i/read next-i) did predict program accuracy. Choice of appropriate loop construct (i.e., while), did not predict accuracy.

Figure 6 displays a program illustrating the potency of the looping strategies which students bring to a problem --- and to a programming language. (Recall our earlier discussion of how a while loop can be used to encode a read i/process i strategy with an embedded test.) We also saw more unusual constructions. In Figure 7 we see a program which has a while loop

```
program Student17_Problem3;
    var Sum, Count, Num : integer;
        Average : real;
    begin
    Sum := 0;
    Count := 0;
    repeat
        Read (Num);
        while Num <> 99999 do
            begin
            Sum := Sum + Num;
            Count := Count + 1
            end
    until Num = 99999;
    Average := Sum / Count;
    Writeln ('The average is ', Average)
    end.
```

This program attempts to solve Problem 3 from Table 1. This student is clearly trying to code the repeat loop body as "read/process" and using the embedded while as a way to (incorrectly) force an exit from the middle of the repeat loop. Note that the above is an actual student's program which has been only minimally edited in order to facilitate readability.

Figure 7: A "Multi-Loop" Program

embedded in a repeat loop.[10] One interpretation is that this student wanted to input a value first, then test it, and, when necessary, jump out of the loop. However, he was not able to marshall the programming constructs to correctly achieve this goal.

We call the kinds of constructions described above (see Figures 6 and 7) "multi-loops." The frequency of multi-loops was not randomly distributed over the three problems. In the novice group, there were 11 "multi-loop" programs, 8 of which occurred on Problem 3; in the intermediate group, there were also

[10] Recall that students in these classes were not taught the goto construct; thus we did not see programs which escaped from the middle of the loop via an if coupled with a goto.

11 multi-loop programs, 8 of which occurred on Problem 3. Our interpretation is that students wanted to implement their read i/process next-i strategy, and they went to great lengths to do so in Problem 3.

## 7. Bugs and Misconceptions: Variables

In this section, we will describe a selection of bugs and misconceptions associated with variables found in the novice and intermediate groups' programs. The three types of variables posited earlier (the Read Variable, Counter Variable, and Running_Total Variable) will help to focus this discussion.

### 7.1. Counter Variable vs. Running_Total Variable: Update Problems

Typically, instructional texts treat the update of the Counter Variable and the update of the Running_Total Variable as instances of the same basic construct, i.e., the assignment statement. If students do treat the two types of variables as the same, we would expect to see no difference in performance, as students should be as likely to handle correctly the Counter Variable update as the Running_Total Variable update. The data tell a different story. On Problem 3, 100% of the novices wrote a correct Counter Variable update, while only 83% wrote a correct Running_Total Variable update. In contrast, the intermediates performed essentially the same on both updates: 91% correct on the Counter Variable update versus 95% correct on the Running_Total Variable update.[11]

Why should the Running_Total Variable update be "harder" than the Counter Variable update for the novices? After all, both require assignment statements of the same form. Below, we list three hypotheses which could

[11] Here we did not include the performance of students on Problems 1 and 2, since Problem 1 focused attention on the Counter Variable and Problem 2 focused attention on the Running_Total Variable.

account for this observation.

1. The activity of counting and the activity of accumulating a total are <u>different</u> activities. Whereas counting is naturally represented as a "successor function" (i.e., increase the previous number by 1), one does not traditionally add up a column of numbers using a running total algorithm. (With the increasing use of pocket calculators, this might change.) However, the standard assignment statement does not distinguish between these two cases, and in fact requires that both activities be encoded in the <u>same</u> way. Thus, the mismatch between one's standard algorithm and the programming language results in the Running_Total Variable update being harder to learn than the Counter Variable update.

2. Students might learn the notion of a counter as a special entity, i.e., they see the instructor and the textbook always using I := I + 1 whenever a counter is needed, and thus they memorize this pattern as an indivisible unit. In other words, they do not decompose I := I + 1 into a left hand variable which has its value changed by the right hand expression. By this hypothesis, students are not viewing I := I + 1 as an example of an assignment statement.[12] When faced with developing an assignment statement for the "running total function" students must really confront their understanding of the particular type of assignment statement needed in this context. The poorer performance in this situation reflects a misunderstanding of how the assignment statement works, and how to encode a running total with this type of statement.

3. The assignment statement in the Running_Total Variable update case requires a variable while the assignment statement in the Counter Variable update case requires only a constant; the difference in performance reflects the fact that variables are "harder" than constants for novices. While this hypothesis might seem like the obvious one, it unfortunately does not provide an adequate <u>explanation</u> for the data. Why should variables be harder than constants? At the syntactic level, writing down a symbol string does not seem all that much harder than writing down a number. At the semantic level, both are the same -- the value of the name is used in the computation. One must go after deeper differences, differences which get at why and when variables are used as opposed to when constants are used. Hypotheses 1 and 2 above attempt to explain the performance difference by describing the circumstances under which each entity is most appropriate. This hypothesis, while appealing on the surface, lacks the explanatory power that we

[12] Performance data on Problem 3 suggest some intriguing corroborating evidence. More novices got the count action correct (e.g., I := I + 1) than got the count initialization correct (e.g., I := 1)! Maybe this was due to sloppiness. However, if I := I + 1 <u>is</u> a unit unto itself, then possibly the students do not see the need to initialize the variable.

seek.

Another curious coding technique, which we observed in students' programs, provides additional supportive evidence that students perceive the the Running_Total Variable differently from the Counter Variable The program in Figure 8 contains a Running_Total update which is "fractured," i.e., split over two lines of code (Y:=X+Z, Z:=Y).

```
program Student21_Problem3;
    var C, X, Y, Z : integer;
    begin
    C := 0;
    Z := 0;
    while X < 99999 do
        begin
        Read (X);
        C := C + 1;
        Y := X + Z;
        Z := Y
        end;
    A := Y div C;
    Writeln (A, ' Average')
    end.
```

The above program contains a fractured Running_Total assignment statement in the body of the *while* loop. Note that it is an actual student's program which has been only minimally edited in order to facilitate readability.

Figure 8: A Fractured Running-Total Update

While not incorrect, this technique is not good programming practice. It probably reflects a confusion of the equal sign in traditional algebra with the assignment operator in programming (see Soloway, Lochhead, Clement [1981b]).

If students perceived the Counter Variable update to be of the same sort as the Running_Total Variable update, then we would expect to see "fractured updates" used in both the Counter Variable update and the Running_Total Variable update. Fractured updates were observed in 7 programs from the novice group. However, in all instances this occurred with respect to a Running_Total Variable update, not a Counter Variable update. While the numbers are small, and thus conclusions drawn from them must be treated with caution, these data nonetheless are consistent with our claim that students do perceive the two variables as being different.

The observations in this section again illustrate our experimental method: analyze programming knowledge with respect to functional characteristics, rather than simply in terms of the syntactic and semantic characteristics.

7.2. The Read Variable

In all three problems (Table 1), a correct solution required that the program "get a new value with a read." 23% of all the programs written by novices did not perform this function correctly. Since "read a value" into a variable is a basic technique which is used continually, we were surprised at the high percentage of novices who could not do this at the end of a semester of programming.

Consider, for example, the program depicted in Figure 9, which is typical of programs in the data. This program is an attempt at solving Problem 1. Notice that there is a read before the loop on the variable Num, but there is no read in the loop. Instead, the variable Num is operated on by the arithmetic operation "subtract 1." What misconception might explain this apparently bizarre program? Recall that in Figure 2 we suggested that the Read Variable and the Counter Variable were both specializations of the New_Value Variable, i.e., both the Read Variable and the Counter Variable were used to hold successively generated values. Possibly, the student who wrote the above program confused a Read Variable with a Counter Variable. Since these variables are "sibling" concepts this confusion seems quite plausible. Thus, a student might well believe that if subtracting 1 from the value of a Counter Variable will return the previous value of the Counter, then

```
program Student19_Problem1;
        var Num, Prev_num, Count : integer;
        begin
        Count := 0;
        Read (Num);
        Sum := 0;
        repeat
                Prev_num := Num - 1;
                Sum := Num + Prev_num;
                Sum := Sum + 1;
                Count := Count + 1;
        until Count = 10;
        Average := Sum / Count;
        Writeln
           ('Average of ten integers is equal to ':2)
        end.
```

This program is an attempt at Problem 1 in Table 1. Note that it is an actual student's program which has been only minimally edited in order to facilitate readability.

Figure 9: Overgeneralizing a Counter Variable to a Read Variable

subtracting one from the variable Num should also return the previous value of Num. As corroborating evidence, notice that the variable which is assigned the value of Num - 1 is called Prev_num.

There is another hypothesis which could account for the difficulty some students seemed to have with the read. It could be that students just didn't understand the semantics of the Pascal read statement. They may not have perceived that a read does 2 things: it gets a new value, and assigns that value to a variable. A language which treated I/O calls as special values that can be assigned "to" a variable or "from" a variable might be more palatable to beginning programmers, e.g.,

```
New_value := Read_from_terminal, or,
Write_to_terminal := Running_sum / Count.
```

## 7.3. Mushed Variables

While experts can distinguish among different kinds of variables, and correctly note when such variables should be used, we found that a substantial percentage of novice programs (27%) used the same variable incorrectly for more than one role. In Figure 10 we display a program with just this sort of "mushed variable" error. In particular, notice that the variable X is used both to store a value being read in {Read (X)} and to hold the running total {X := X+X}. While two distinct variables are needed in order to realize these functions in the program, the program in Figure 10 uses only one variable.

```
program Student26_Problem2;
    var X, Ave : integer;
    begin
    repeat
        Read (X);
        X := X + X
    until X + X > 100;
    Ave := X div Nx;
    Write (Ave)
    end.
```

Note that the above is an actual student's program which has been only minimally edited in order to facilitate readability.

Figure 10: Mushed Variables: An Example

Frankly, it is quite difficult to conjure up a simple explanation for why students created such programs. For example, it is possible that a student did recognize that the same variable played two different roles, but assumed that the computer also recognized this and would use the different values appropriately. Often novices impute significant power and wisdom to a computer. We are currently carrying out video-tape interviews with students as they write programs. From this type of data we hope to develop clearer pictures of why students make this type of error.

Only 8% (11/143) of the intermediates' programs contained mushed variables as against 27% (21/77) for the novices. The difference between this value and that for the novices was significant at the $p < .001$ level (chi square = 15.44). One would expect that a student would need to clear up this problem in order to progress in computing. That is, one can understand the intermediates not knowing when to use the _for_ and _repeat_ loops, but still performing quite successfully with only the _while_ loop. Mushing variables, however, is a catastrophic conceptual error which must be overcome in order to proceed.

## 8. Related Work

Some fine work has been done in Artificial Intelligence and Cognitive Science on understanding problem solving in complex domains. In particular, Collins [1978] has argued for the key role which tacit knowledge plays in the problem solving of experts. Papert's [1980] theory of learning, and his work on the design of LOGO, have impressed upon us the importance of the match between cognitive abilities and programming language constructs. Brown and Burton [1978] and Brown and VanLehn [1980] have developed a sophisticated model to explain the origin of bugs which people exhibit in their subtraction algorithms; the development of a similar model to explain the bugs and misconceptions described here is one of our goals.

Others have mapped out the knowledge which experts use in problem solving in other domains. For example, Polya [1973] and Rissland [1978] have described the knowledge which mathematicians use; they emphasize that mathematicians do not actually do mathematics in the "definition, theorem, proof" style employed by most textbooks, but rather use plans and goals to guide their thinking. Larkin et al. [1980] have made similar arguments for expert problem solving behavior in physics. A number of researchers have described various knowledge bases for different areas of programming, e.g., Barstow [1979], Rich [1981], Rich and Shrobe [1978], Miller [1978], Waters [1979].

Some of the issues we are addressing have also been studied by researchers in the emerging field of "software psychology." For example, Gannon [1978] reports on a set of bugs which he found to occur in student programs. M. Miller and Becker [1974] attempted to identify the natural problem solving strategies of non-programmers. Sheppard et al. [1979] explore, among other issues, the effect on understanding of different looping structures. Mayer [1980] has explored the utility of providing explicit models as an aid to learning programming. Shneiderman [1976] and Adelson [1981] have done experiments which show that experts tend to see programs in terms of functional units, while novices tend to focus on the individual statements in a program. Shneiderman [1980], in a recent book which brings together much of the research in this field, also emphasizes the important role of higher level, plan knowledge in programming.

## 9. Concluding Remarks

People will need to program with some degree of proficiency in order to continue active participation in our technological society. To facilitate their education we need to develop programming languages, programming environments, and instructional materials which are tuned to the cognitive abilities of this audience. We are now engaged in building a system which can help students in introductory programming courses to debug their programs and come to grips with their misconceptions [Soloway, et al. 1981c]. A key component of this project is the empirical work reported here. In order to design and implement material which facilitates learning, we need to understand the sources of the bugs and misconceptions that students have.

In this paper we have examined some of those misconceptions by looking beyond "syntax and semantics" errors to the plans and strategies that people use when they program. By focusing on a few critical aspects of programming we have shown that looping <u>strategy</u> is a key factor in programming. Similarly, we found that students are quite sensitive to the <u>pragmatic</u>

differences among variables. Finally, while we did not set out in this paper to examine issues pertinent to education, our data leads us to the belief that more emphasis should be placed on teaching the plans and strategies relevant to programming than is done now.

**Acknowledgements** A number of people have read and commented on drafts of this paper: Larry Birnbaum, George Cherry, Ann Drinan, Jim Galambos, and Jerry Leichter, G. Michael Schneider, Ben Shneiderman, and Bonnie Webber. We sincerely thank them for their help.

## 10. Bibliography

- Adelson, B. (1981) "Problem Solving and the Development of Abstract Categories in Programming Languages," Memory and Cognition, vol. 9., pp. 422-433.

- Barstow, D. (1979) Knowledge-Based Program Construction, Elsevier North Holland, Inc., New York.

- Brown, J.S. and Burton, R.R. (1978) "Diagnostic Models for Procedural Bugs in Mathematics," Cognitive Science, June.

- Brown, J.S. and VanLehn, K. (1980) "Repair Theory: A Generative Theory of Bugs in Procedural Skills," Cognitive Science, Vol. 4, #4, Oct.

- Collins, A. (1978) "Explicating the Tacit Knowledge in Teaching and Learning," presented at the American Education Research Association (also BBN Technical Report 3889).

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

- Gannon, J.D. (1978) "Characteristic Errors in Programming Languages," Proc. of 1978 Annual Conference of the ACM, Washington, D.C.

- Jensen, K. and Wirth, N. (1974) Pascal User Manual and Report, Springer-Verlag, New York.

- Knuth, D. (1974) "Structured Programming With go to Statements," ACM Computing Surveys, Vol. 6, No. 4.

- Larkin, J., McDermott, J., Simon, D. and Simon, H. (1980) "Expert and Novice Performance in Solving Physics Problems," Science, 208.

- Mayer, R. (1980) "Contributions of Cognitive Science and Related Research in Learning to the Design of Computer Literacy Curricula," Conference on National Computer Literacy Goals for 1985, Virginia.

- Miller, M. (1978) "A Structured Planning and Debugging Environment for Elementary Programming," Int. J. Man-Machine Studies, 11, pp. 79-95.

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

- Minsky, M. (1975) "A Framework for Representing Knowledge," in The Psychology of Computer Vision (P.H. Winston, ed.), McGraw-Hill, New York.

- Moher, T. and Schneider, G. M. (1981) "A Methodology for Improving Experimentation in Software Engineering," IEEE Fifth International Symposium on Software Engineering, San Diego, Calif.

- Nilsson, N. (1980) Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, California.

- Papert, S. (1980) Mindstorms, Children, Computers and Powerful Ideas, Basic Books, Inc., New York.

- Polya, G. (1973) How To Solve It, 2nd Ed., Princeton University Press, New Jersey.

- Rich, C. (1981) "A Formal Representation for Plans in the Programmer's Apprentice", Proceedings of IJCAI-81, Vancouver,B.C.

- Rich, C. and Shrobe, H. (1978) "Initial report on a LISP Programmer's Apprentice", IEEE Transactions on Software Engineering, V4, no.6, November.

- Rissland, (Michener) E. (1978) "Understanding Understanding Mathematics Cognitive Science, vol. 2, no. 4.

- Sheppard, S.B., Curtis, B., Milliman, P, and Love, T. (1979) "Modern Coding Practices and Programmer Performance," Computer, December.

- Shneiderman, B. (1976) "Exploratory Experiments in Programmer Behavior," International Journal of Computer and Information Sciences,5,2, 123-143.

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

- Soloway, E. and Woolf, B. (1980) "Problems, Plans, and Programs," in Proc. of Eleventh ACM Technical Symposium on Computer Science Education, Kansas City.

- Soloway, E., Bonar, J., Ehrlich, K. (1981a) "Cognitive Factors in

Programming: An Empirical Study of Looping Constructs." Technical Report 81-10, Dept of Computer and Information Science, Univ. of Mass., Amherst.

- Soloway, E., Lochhead, J., Clement, J. (1981b) "Does Computer Programming Enhance Problem Solving Ablility? Some Positive Evidence on Algebra Word Problems, in National Goals for Computer Literacy in 1985, R. Seidel (ed.), in press.

- Soloway, E., Woolf, B., Rubin, E. and Barth, P. (1981c) "Meno-II: An Intelligent Tutoring System for Novice Programmers, Proceedings of IJCAI-81, Vancouver,B.C.

- Waters, R.C. (1979) "A Method for Analyzing Loop Programs," IEEE Trans. on Software Engineering, SE-5:3, May.

- Wirth, N. (1971) "The Programming Language Pascal," Acta Informatica Vol. 1, No. 1.

- Wirth, N. (1974) "On The Composition of Well-Structured Programs," ACM Computing Surveys, Vol. 6, No. 4.