Cognition and Programming: Why Your Students Write Those Crazy Programs

Elliot Soloway, Jeff Bonar, Beverly Woolf,
Paul Barth, Eric Rubin, and Kate Ehrlich

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

TR# 81-05

Abstract

In this paper, we first argue that there is a great need for empirical research coupled with carefully articulated theories. Psychological claims of language designers and advocates call out for evaluation. Moreover, computing education, to rise above instruction in only the syntax and semantics of language constructs, needs a description of the knowledge programming experts know and use. We then present a network which represents the high-level, plan knowledge an expert may possess with respect to aspects of looping and assignment. Based on this knowledge, we look at actual student programs, and attempt to understand the possible misconceptions students had, which manifested themselves as buggy programs. Finally, we make suggestions for computing education which reflect the insights gained in developing this knowledge network and in the analysis of the buggy programs.

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.

I. Introduction

I.1 Introductory Polemic

Designers and advocates of programming languages are continually making claims that their languages are simple, readable, encourage better programming, or encourage natural problem solving habits.

> The development of the language Pascal is based on two principle aims. The first is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language. ... The syntax of Pascal has been kept as simple as possible. ... This property facilitates both the understanding of programs by human readers and the processing by computers.
>
> Wirth [1977]

> If ordinary persons are to use a computer, there must be simple computer languages for them. BASIC caters to this need ....
>
> Kurtz [1978]

> Happily, LISP is easy to learn. A few hours of study is enough to understand some amazing programs. ..... One reason LISP is so easy to learn is that its syntax is extremely simple.
>
> Winston [1981]

APL is one of the most concise, consistant, and powerful programming languages ever devised.
McCracken [1976]

Notice that the above claims are psycnological claims and thus are open to empirical inquiry. Where, then, is the empirical research backing up these claims? Moreover, it is not enough to have data -- one needs theories which explain and account for the data. What are needed, then, are carefully articulated theories which suggest why program construct x should be more readable, useful, etc. than program construct y?

Moreover, in the rush to claim readability, etc. for their particular programming language, designers have often forgotten a key aspect of cognition: learning. Can individuals learn to understand program construct x? How easily? The extensive use of Pascal, Basic, Lisp, and APL does not negate the relevance of the learnability question: humans can adapt to most any environment. Thus, no matter how difficult a particular language might be to learn, many individuals can and do persevere to conquer it. But at what cost? And what about the ones that don't succeed? Do we want to simply write them off? As we all know, with the coming of the computer age, everyone will need to know how to program to some degree.

The picture is not as bleak as we've painted it. Fortunately, studies have been and are being done, which attempt to understand the relationship between cognition and programming. For example, Ledgard, et. al. [1980] found that a command language which has a natural language structure, as opposed to the typically baroque constructs, facilitates easier and more effective use by humans. Welty and Stemple [1981] compared the performance of novice users with a procedural and non-procedural query langauge and found that novices using the procedural language did better when formulating moderate to difficult queries. Dunsmore and Gannon [1978] explored the factors that contribute to program complexity, while Gannon [1978] catalogued bugs in student programs. L. Miller [1978] has explored a whole range of behavioral issues in programming. Among other issues, Meyer [1980] has explored the use of concrete models in programming education. Finally, Shneiderman [1980] contains a distillation of issues in, as he puts it, software psycnology.

A word of caution: this type of research is difficult to carry out. Experiments can always be criticized; theoretical interpretations can always be faulted. Those in this paper are no exception. Moreover, the climate in computer science has not always been receptive to, or even tolerant of, this type of work. The need to study cognitive factors in computing, however, grows ever more apparent. Thus, we feel it imperative that such research be encouraged, and the support mechanisms to insure its existence arise.

I.2 Substantive Introduction

This paper will serve as a progress report on the current work of the Meno Research Group at UMass. The goals of our project are twofold. First, we are building Meno-II, a run-time support environment for novice Pascal users; this system will catch run-time errors (not compile-time errors [1]), and help the student debug the program -- and his/her understanding -- by providing tutoring in the areas underlying the errors. The other, complementary goal of this project is to understand:

1. what one knows when one solves problems of the sort used in introductory Pascal courses,
2. what program bugs are typically made by novice programmers,
3. how knowing about what should be understood can be used to explain the mind bugs (misconceptions) students have which result in program bugs.

In this paper, we describe our efforts to date on this latter goal.

[1] We are not concentrating on compile-time errors, since systems which cope with such errors already exist. For example, Teitelbaum [1980] has built a programming environment which helps eliminate syntax errors by not allowing programmers even to enter ill-formed structures.

The perspective we have taken in this research is based on the following observation: as experts, when we write programs to solve problems, we use lots of high-level knowledge. Most of this knowledge is much deeper than the syntax and semantics of any given programming language. In particular, an expert knows how to decompose problems into fairly standard tasks, e.g., accumulate a running total and search through the elements of this array. This knowledge is highly organized into bundles, or frames [Minsky, 1975], and the rich fabric of bundles are knit together by relationships. The key role which this tacit knowledge plays in the problem solving behavior of experts in a wide variety of areas is gaining increasing attention [Collins, 1978]. The difficult problem is to ferret out this knowledge and make it explicit, much as Dijkstra [1976], Wirth [1977], and Friedman [1974] have attempted to do in their books on programming. Thus, it is this knowledge, rather than just the syntax and semantics of the particular programming language, that needs to be taught to the growing numbers of computing students.

In the following section we present a first-pass at describing the knowledge an expert might know about aspects of looping and assignment. Section III concentrates on how this knowledge can be used to predict and explain students' bugs. There we examine data -- bugs -- collected from a test we administered to introductory Pascal programmers. In Section IV we summarize the key theses of the paper.

II. Fragments of an Expert's Knowledge: The Theory

We commented above that an expert programmer knows about structures that help him/her relate problems to programs. These intermediate entities reflect a process of abstraction and condensation. That is, problems, and their solutions, are grouped together based on various criteria of similarity and their salient characteristics are highlighted. When confronted with an ostensibly new problem, the expert calls upon these structures and tries to find an old problem (and solution) which is similar to the new problem. Solving the new problem, then, is based on modifying the solution to the old problem. These intermediate structures are crucial for this type of problem solving (Papert, [1980]; Rissland and Soloway, [1980]).

While others have described the knowledge possessed by experts in mathematics (e.g., Polya [1963], Rissland [1980]) and physics (e.g., Larkin, et al. [1980]), we have attempted here to articulate the knowledge a programming expert might have with respect to a few problem types and a few related program constructs. We have borrowed from artificial intelligence a representation of knowledge, called frames, in which to encode our efforts. One key idea behind this particular knowledge representation -- and one consistent with our view of what an expert might know -- is that concepts are not necessarily atomic entities; rather they can have internal structure, such as descriptor types and descriptor values. Thus, a frame represents a template or boilerplate with slots which can be filled in. Finally, frames are tied together by different types of relationships, e.g., A-KIND-OF, or USES. [1]

Figure 1 depicts a fragment of an expert's knowledge -- plans -- encoded as frames. Consider, in particular, the Running_Total Plan Frame. This structure reflects the observation that many problems, especially ones used in introductory courses, require accumulating a total (e.g., the sum or product of the first n integers). Thus, given the ubiquity of this problem type an expert might explicitly represent it in his/her memory. From Figure 1, we also see that this frame is A-KIND-OF Loop Plan, i.e., a specialization of the general notion of a loop plan, and further that the Counter Loop Plan Frame is a specialization of Running Total Loop Plan. We see that various specific Pascal loop constructs (repeat, while, for) are associated with this plan. Finally, we see that two variable frames, the Running_total Variable and the New_Value Variable are used to describe the Running Total Loop

[1] In this discussion, we have glossed over many technical distinctions. However, for our purposes, the commonsense notion of "plan" and "frame" are sufficient. The interested reader can follow these issues more carefully in texts such as Nilsson [1980].

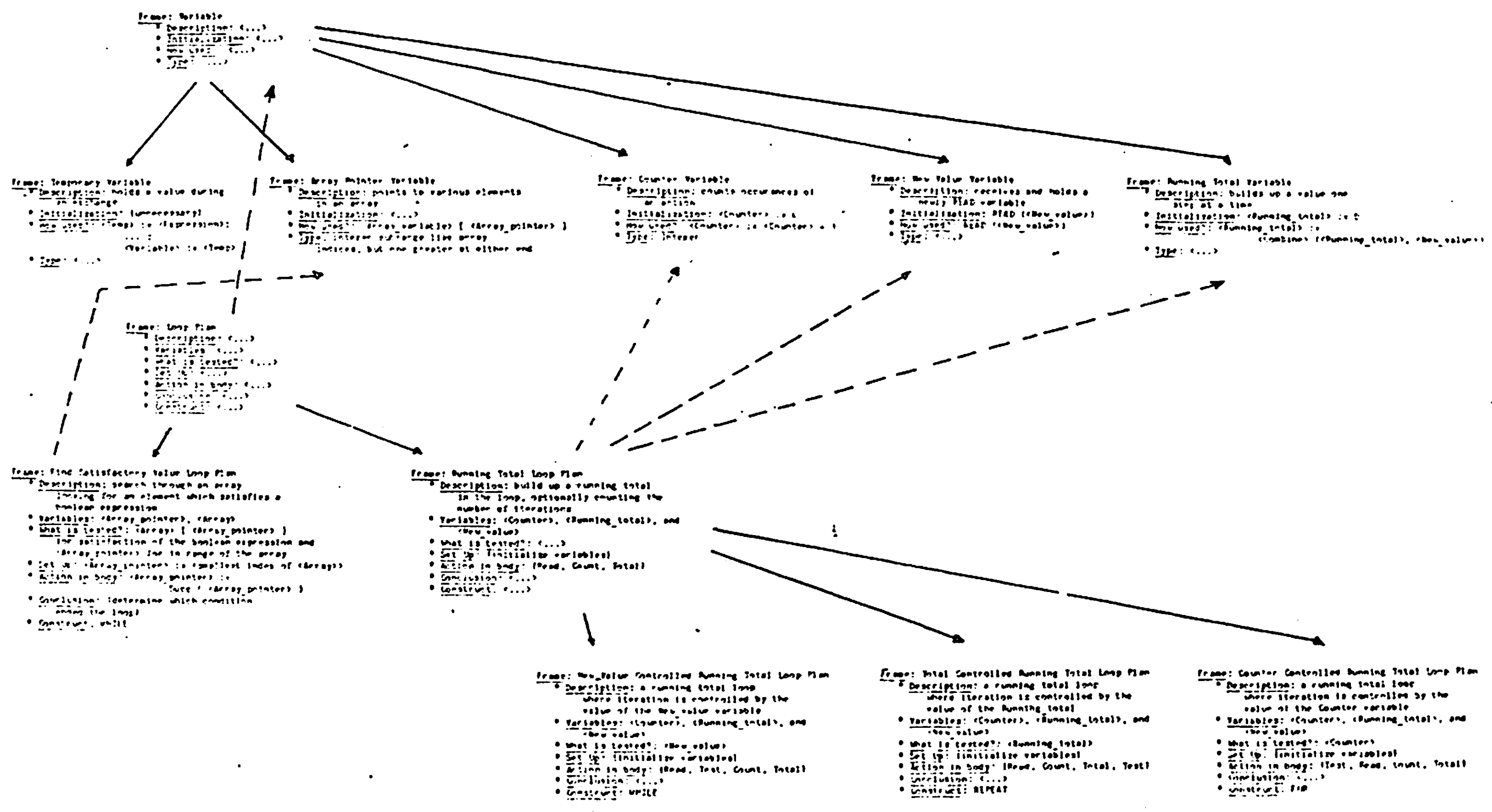
Frame: Variable
Frame: Temporary Variable
Frame: Array Pointer Variable
Frame: Counter Variable
Frame: New Value Variable
Frame: Running Total Variable
Frame: Loop Plan
Frame: Find Satisfactory Value Loop Plan
Frame: Running Total Loop Plan
Frame: New_Value Controlled Running Total Loop Plan
Frame: Total Controlled Running Total Loop Plan
Frame: Counter Controlled Running Total Loop Plan

Figure 1. 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). Items in brackets are descriptions of things that must be inserted when the Frames are used. Standard Pascal identifiers are in all capitals.

Plan.

From a syntax and semantics view, the distinctions made in Figure 1 would disappear. For example, the Running_Total Variable and the Counter Variable are the same at those levels. However, at the functional, pragmatic level we see differences in the roles which variables and constructs play. Figure 1 highlights those functional differences. Moreover, as we shall see in our discussion of program bugs (next section), students performed differently on constructs, which on the syntax and semantics level, are the same.

We do not mean to suggest that the knowledge incorporated into Figure 1 is the final word. In fact, other experts have proposed alternative schemes (e.g., Waters [1979]; Soloway and Woolf [1980]; Miller, M. [1978]; Goldstein [1974]). Regardless of the details, experts do have and use structures similar to those in Figure 1, and we should be teaching students based on this view.

Besides using such knowledge for teaching, we see another important use for it. It can help us to understand bugs, program bugs and mind bugs, committed by novice programmers. First, we subscribe to the theory of learning that states that students understand something by constructing their own knowledge frameworks. A mind bug, then, is a "faulty" knowledge net, when compared with an expert's. That is, "bug" is only a bug because it does not agree with some standard. Moreover, a good teacher, who also has a good knowledge net, can use the student's bug as a window into the student's knowledge net. Brown and Burton [1978] use this approach in their analysis of students' buggy subtraction algorithms. Remediation can then be developed which is tuned specifically to the subtleties of the student's buggy model.

In the next major section of this paper, we shall use aspects of the knowledge in Figure 1 to analyze program bugs we collected from novice programmers. We found that <u>without</u> this type of, high-level <u>knowledge</u>, many of the buggy programs seemed anomalous at best, but with the development of knowledge of the sort depicted in Figure 1, these bizarre programs became understandable to a large degree.

III. Understanding Program Bugs and Mind Bugs: The Data

III.1 The Context of the Test and the Test Instrument

Table 1 contains three of the problems we asked 31 introductory Pascal programming students to solve. The course was given in the summer session of 1980; the test was administered on the last day of classes. The students ranged from freshman to graduate students. Table 1 depicts the overall performance of students on these questions; the percentages are surprisingly low, given the ostensibly simple nature of the problems and the advanced state of the class. Even if one ignores what might be called easily detectable mistakes (e.g., no initialization of a variable), the percentages do not rise appreciably.

A reason for this low performance, and an important criticism of programming tests given where students do not have access to a computer is that introductory programming students almost never write a program correctly right off the bat. Rather, they go to the terminal and, over possibly several sessions, evolve a correctly running program. Nonetheless, data gathered from written tests does suggest trouble spots, which if resolved might lead to more effective problem solution/program development. [1]

[1] We are also collecting on-line protocols; as a student sits at a terminal and submits a run to the Pascal system, our system makes a copy of the student's program and saves it. Thus, we have a record of the student's interactions. So far, we have data on 3 introductory Pascal classes. We are analyzing these data now. We also interview students individually, recording these sessions on audio tape (we hope to soon go to video tape). These type of data is particularly illuminating; whereas the on-line data and the test data are simply records of output, of performance, the interview data provides us with a window into the student's thought processes. In related research, we have found this technique to be a source of genuine insights (Clement, Lochhead, Soloway [1980]).

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. Problems used in our test instrument. These problems were given to an introductory programming class on the last day of the course. They are designed to test student knowledge of key differences between different loop constructs in Pascal.

Percentage of Students Which:	Problem 1 The for loop problem	Problem 2 The repeat loop problem	Problem 3 The while loop problem
Used for loop	16%	--	--
Used repeat loop	35.5%	53.6%	45.8%
Used while loop	35.5%	39.3%	41.7%
Used other constructions	12.9%	7.2%	12.6%
Answered problem correctly AND chose appropriate loop construct	9.7%	25%	20.8%
Answered problem correctly	36%	43%	38%

Table 2

The problems referred to in this table are listed in Table 1. The above data is based on a sample size of 31 students.

```
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 2. A stylistically correct solution to problem 3 in table 1. Note the need for two Read calls and the curious "process the last value, read the next value" semantics of the loop body. This program was minimally edited for presentation here. Students wrote these programs in a classroom. They were never submitted to a translator.

III.2 Performance Analysis: Use of Appropriate Loop Construct

In this course students were taught and used all 3 Pascal loop constructs: while, repeat, and for. Each of these constructs is appropriate in different situations (e.g., see Figure 1). Do novice programmers, in fact, distinguish between them and use them in the appropriate context? Answer, based on our data: no.

Table 2 lists the percentage of students using each particular loop construct on each problem; also listed are the percentage of students who used the appropriate loop construct and got the problem correct. The data on problem 1 are surprising. This problem is clearly a for loop problem, since the variable being tested is the counter variable. However, only 16% used a for loop; 35.5% used a repeat loop and 35.5% 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 (initialize counter, test counter for stopping, increment counter). Moreover, of the 36% who got the problem 1 correct, only 27% used the for loop. Clearly, choosing the appropriate loop construct did not contribute to correctly solving the problem.

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 would be reasonable to test it. While more students did in fact choose the most appropriate loop construct, the difference between those choosing the repeat and those choosing the while was not statistically significant. Moreover, as in the above case, choosing the appropriate loop construct did not correlate with solving the problem correctly.

For problem 3 (Table 1), the appropriate loop construct is while; the loop must not be executed if the controlling variable has a specified value and therefore the test must be placed at the head of the loop. (This, in turn, requires a curious coding structure which we examine in the next section.) Here again, we see that the difference between those choosing a repeat loop and those choosing a while loop was not statistically significant. Nor again was the choice of the appropriate loop construct a predictor of the correctness of the solution.

Based on this simple test, it appears that novices do not distinguish between the Pascal loop structures as an expert might. A quote from a student we interviewed is appropriate here. When asked why he chose to use the while construct rather than one of the other two, he responded: "When I don't know what is going on, I use a while loop."

At first, we found the results on 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, however, we decided that the automatic, implicit aspects of the for loop might be the sticky point. That is, since the for loop does do a number of actions automatically, students might be uncertain about all the details of the for loop's actions. Thus, in order to have some control over the beasty, students might choose a repeat or while loop and do the extra work to add the required looping machinery themselves. [1]

Quite simply, the difference between the repeat and while loops is subtle. Moreover, since with the appropriate extra machinery one construct can simulate the other, the distinction is hard to enforce. However, 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]

[1] Some support for this interpretation comes from the following fact: students in this class were not taught the goto, and never constructed loops out of the basic constructs. Thus, these students may be unsure of the ingredients of a loop and might not be comfortable with all the magic implicit in a for loop.

> 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]

We feel that this is a syntax level description. What should be emphasized is the deep structure of this construction:

> Use a repeat loop if the controlling variable requires that it be assigned a value in the loop before it can reasonably be tested.

The frames and slots in Figure 1 reflect the close connection 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 between the three loop types is not important for a novice. To some extent, we agree. We wonder then why textbooks teach all 3.

III.3 Performance Analysis: Read i/Process i vs. Process i/Read Next-i

Let us now take a closer look at problem 3, the while loop problem. The stylistically correct solution 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 get the loop started. However, this results in the loop processing being behind the read; it processes the ith input and then fetches the next-i. We call this structure "process i/read next-i."

Intuitively we felt this coding strategy to be unnecessarily awkward and downright confusing. A more "natural" coding strategy would be to read the ith value and then process it; we call this the "read i/process i" coding strategy. Do novice programmers use the stylistically correct coding strategy (process i/read next-i), or do they add extra machinery to a while or repeat loop (e.g., an embedded if test tied to a boolean variable) in order to force the code into a read i/process i structure?

Table 3 lists the performance of those students who attempted the problem with either a while or repeat loop. Of the nine who solved it correctly, only 2 used the stylistically correct process i/read next-i coding strategy. (See Figure 2 for a solution using this coding strategy.) To correctly solve the problem using either a repeat or while loop and the read i/process i coding strategy requires extra machinery; Figure 3 shows student programs which use this strategy. Nonetheless, the vast majority of students attempted this solution; given the extra complexity needed for a correct solution, it is not surprising that many failed.

It is tempting to conclude that with respect to these types of problems, Pascal requires that students circumvent their natural problem solving intuitions. Before we can actually assert this conclusion, more research needs to be done [1]. But, since we must live with Pascal for some time to come, it would only be responsible for teachers to explicitly teach their students about this peculiar coding strategy. Again, since humans are adaptive, we probably could learn to deal with this awkward and confusing construction.

III.4 Performance Analysis: Getting a New Value

In all 3 problems (Table 1), a correct solution required that the program get a new value with a read. 23% of all the student written programs did not perform this function correctly. Often students try to get the previous or next

	Read i/Process i used			Process i/Read Next-i used		Other
	repeat loop	while loop	other	repeat loop	while loop	
Correct	4	2			2	1
Incorrect	3	5	4			2

Table 3

The numbers in this table refer to the actual number of students, not percentages.

```
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.
```

```
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 3. These programs are attempts at problem 3 described in table 1. They are typical of the contortions students will go through to make this problem fall into a "read a value, process that value" Frame. These programs have been minimally edited for presentation here. Students wrote these programs in a classroom. They were never submitted to a translator.

value of a variable by subtracting or adding one (see Figure 4). [2] We also found programs in which we felt students assumed that each use of Next_value automatically retrieved a new value.

As we indicated in Figure 1, getting a new value is different than, say, accumulating a total. Thus, perhaps students committing the above errors did not understand that read is actually just a special case of assignment. If so, then a language which treated I/O calls as special values which can be assigned to or from might be more palatable to beginning programmers, e.g.,

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

Another possible mind bug which could result in some of the observed errors would be that students incorrectly overgeneralized from the Counter Variable Frame. That is, since the next value of a variable functioning as a counter can be retrieved by simply adding a 1 to the variable, why not get the next value of any variable by simply adding a 1 to it! While reasonable, this is incorrect. This type of overgeneralization could be predicted from the relationship of the frames in Figure 1.

[1] We have designed and pilot-tested the following experiment: first, we ask all students to write a plan or design for problem 3 in Table 1 (the same one examined in this section), in a language other than a programming language. We then ask half the students to write the program in Pascal. For the other half of the group, we provide a one page description of the Ada loop ... exit loop construct. While the sample size was small (13 students), the data are suggestive: invariably the plan of the students was worded in terms of a read i/process i. However, the Pascal versions were typically coded with a process i/read next-i strategy. But, those programs written using the Ada loop ... exit were coded using the read i/process i strategy. Thus, the program coded in Ada more closely matched the students' plans than did those program coded in Pascal. We plan to run this experiment on a larger group.

[2] "Backing up" may be needed when a student does the while loop problem (problem 3) with a repeat loop.

III.5 Performance Analysis: The Different Role of the Assignment Statement in the Counter and the Running Total Templates

If one chose to use either a repeat loop or a while loop in any one of the three problems, one would need to explicitly keep track of at least two quantities: (1) the number of numbers which were read in, and (2) a tally of the sum of the numbers read in. In both cases, one would use a particular type of assignment statement which facilitated a running sum, e.g., Running_total := Running_total + New_value. In the former case, New_value would be the constant 1, while in the latter case New_value would be dependent on the value read in.

Since the underlying programming language construct is the same in both cases, one might think that if a student used the construct correctly in the counter case, then the student would understand the construct and would most likely be able to use it correctly in the analogous case of tallying up the sum. However, the performance results in Table 4 portray a different picture. Namely, significantly more students constructed a correct assignment statement for the counter action than could do so for the running total action.

Why? The knowledge network in Figure 1 suggests, in fact, that the Counter Template and the Running_Total Template are distinct, since the functions they perform, while similar, are still different. Moreover, the Running_Total is more complicated since the New_Value Template needs to be integrated to provide a correct solution. Thus, one possible explanation of the performance difference would be that the students did not fully understand how these two functions were integrated; this added complexity was responsible, then, for the poorer performance.

Yet another interpretation consistent with the frame organization suggested by Figure 1 is the following: students understand the counter action as a whole, and do not decompose I := I + 1 into a left hand variable having its value change by the right hand expression.

	Sample Size	Percentage: Correct Running-Total Assignment	Percentage: Correct Counter Update Assignment	Statistical Significance
Overall (across all problems)	69	68%	81%	.019*
Problem 1	22	59%	77%	.042*
Problem 2	26	69%	69%	1.00
Problem 3	21	76%	100%	.021*

Table 4

The asterisks indicate statistically significant differences.

```
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.


program Student30_Problem2;

   var N, Sum, Score : integer;  Mean : real;

   begin
   N := 0;
   Sum := 0;
   Score := 0;
   while (Sum <= 100) do
           begin
           Score := Score + 1;
           Sum := Sum + Score;
           N := N + 1
           end;
   Mean := Sum / N;
   Writeln ('the mean = ', Mean:10:10)
   end.
```

Figure 4. These programs are attempts at the problems described in table 1. They illustrate student problems with getting a New_value. These programs have been minimally edited for presentation here. Students wrote these programs in a classroom. They were never submitted to a translator.

That is, students do not view I := I + 1 as an example of an assignment statement.[1] Thus, 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.

III.6 Performance Analysis: The "Demon" in the while loop test

Based on our examination of student programs, and on an analysis of the individual interview, we felt that there was a great deal of confusion surrounding the time at which the terminating test in the while loop gets evaluated: is it evaluated once at the top of the loop, or is the test continually evaluated during the execution of the body of the loop? The program given below was also on a written test taken by the 31 summer school students.

```
program Problem4;
    var Count : integer;
    begin
    Count := 0;
    while Count < 7 do
            begin
            Writeln ('*');
            Count := Count + 1;
            Writeln ('/')
            end
    end.
```

If the students felt that the terminating test was evaluated continually, then the loop should terminate before an '/' were printed, thus providing one more '*' and '/'.[2] In otherwords, it is as if the test were a demon watching the statements in the loop body, and waiting for its condition to become true. Of the 31 students, 34% made the above mistake. Since while is commonly used in programs and in the instruction, and since it was the end of the semester, we felt that this was a surprisingly high percentage.

The basis for this confusion is grounded in the mismatch between the semantics of while in a programming language context, and the semantics -- the meaning -- of 'while' in every day experience. In the latter case, 'while' has a global sense: during the course of some event. In contrast, the programming language while requires a local, narrow interpretation: at a specific point in time. Clearly, the names of programming language constructs must rely on real world semantics of their analogs. However, care ought to be exercised in their selection. Since the likelihood of renaming the while construct in Pascal is small, educators must take note of this error, and pay attention to it in their instruction.

IV. Concluding Remarks

In this paper we first argued for the need of empirical research coupled with crisp, detailed theories of the programming process. We then went on to develop an explicit knowledge network which represents what programming experts might know about looping and assignment. We argued that since experts apparently had knowledge, such as plan types, which were structured into bundles, it was only responsible that we as educators teach our students this type of knowledge. Based on this knowledge network, we analyzed buggy programs collected from introductory Pascal students. We expressed the analysis in terms of mind bugs -- misconceptions -- that resulted in program bugs. In addition, we found that Pascal construction themselves might be the cause of some program bugs, since they trigger inappropriate and misleading conceptions in a student's mind.

Without a doubt, the claims we have made are controversial -- in fact, they may even be incorrect. However, we strongly feel that dialogue must be encouraged on this type of research, if

[1] Problem 3 suggest some intriguing corroborating evidence. More students 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 is a unit unto itself, then possibly the students do not see the need to initialize the variable.

[2] We were not interested in the actual number of '*' and '/', because we were not studying the off-by-one bug in this particular problem.

computer science education, programming language design, and computer literacy, are to be advanced.

V. Acknowledgements

We would like to express our appreciation to Steven Levitan for his helpful comments and David Lee for his diligent support work.

V. Bibliography

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

Clement, J., Lochhead, J. and Soloway, E. (1980) "Positive Effects of Computer Programming on Students' Understanding of Variables and Equations," Proc. of National ACM Conference, Nashville.

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

Dijkstra, E.W. (1976) A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Dunsmore, H.E. and Gannon, J.D. (1978) "Programming Factors - Language Features that Help Explain Programming Complexity," Proc. of National ACM Conference.

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

Friedman, Daniel (1974) The Little LISPer, Science Research Associates, Menlo Park, Calif.

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

Goldstein, I. (1974) "Understanding Simple Picture Programs," Technical Report AI-TR-294, M.I.T. A.I. Lab, Cambridge.

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

Kurtz, T.F. (1978) "BASIC," appeared in the Proceedings of the ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Notices vol. 13, num 8, Aug.

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

Ledgard, H., Whiteside, J., Singer, A. and Seymour, W. (1981) "Report on an Experiment on the Design of Interactive Command Languages," to appear in Communications of the ACM.

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.

McCracken, Dan (1976) forward to APL: An Interactive Approach, by Leonard Gilman and Allen J. Rose, John Wiley and Sons, New York.

Miller, Lance (1978) "Behavioral Studies of the Programming Process," IBM Technical Report RC7367, Yorktown Heights, New York.

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

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

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.

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

Rissland, E.R. and Soloway, E.M. (1980) "Generating Examples in LISP: Data and Programs," Technical Report 80-07, Dept. of Computer and Information Science, Univ. of Mass., Amherst.

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.

Teitelbaum, T. and Reps, T. (1980) "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Department of Computer Science, Cornell University, Technical Report 80-421, May.

Waters, R.C. (1979) "A Method for Analyzing Loop Programs," IEEE Trans. on Software Engineering, SE0:3, May.

Welty, C. and Stemple, D.W. (1981) "Human Factors Comparison of a Procedural and a Nonprocedural Query Language," to appear in Trans. on Database Systems.

Winston, P. and Horn, B.K.P. (1981) LISP, Addison-Wesley Publishing Co., Reading, Massachusetts.

Wirth, N. (1976) Algorithms + Data Structures = Programs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Wirth, N. (1977) "The Programming Language Pascal," Acta Informatica, 1, pp. 35-63.