An Analysis of Novice Programmers Learning a Second Language

Jean Scholtz
Computer Science Department
Portland State University, Portland, OR

Susan Wiedenbeck
Department of Computer Science and Engineering
University of Nebraska, Lincoln, NE

Abstract

This research studied novice programmers with some Pascal knowledge during their initial attempts at learning another programming language. We wanted to identify the programming knowledge they had previously acquired and to determine how they were able to use this knowledge in learning a second language. We found that plan structure differences could be used to predict problems programmers encountered. This both strengthens the claim for plan knowledge and suggests some basis for tutoring development. Additionally, we discovered trends showing that when the language was more distant from the one programmers knew a bottom up or plan creation methodology was more successful than a plan retrieval process.

Introduction

During an undergraduate program in Computer Science, the typical student takes such courses as programming languages, data structures, operating systems, algorithms and theory. While studying these courses, students are required to write various types of programs in several different languages. Students typically learn a first language such as Pascal or Modula 2. They then use C in courses such as operating systems and compiler theory. They may use Lisp or Prolog in a course in Artificial Intelligence. They may also be introduced to object oriented languages and parallel languages. Much of the early empirical work has focused on novice programmers, what they know (Soloway et al.,1984; Soloway, Bonar, and Ehrlich, 1988) and misconceptions they have(Bonar and Soloway, 1988; Spohrer, Soloway, and Pope, 1988; Spohrer and Soloway, 1988). While students in their second or third year of study are still relatively new at programming, we feel their process of learning a new language will differ significantly from the process involved in learning a first language. Students will be building on the knowledge they acquired in learning the first language. We felt that an interesting study would be to examine the knowledge novices have about programming and see how or if this knowledge is used in the process of learning a new programming language.

Soloway et al. (1984) formulated the idea of a plan in program knowledge. Rist (1991a) defines a plan as a series of actions that achieve a goal. So a running total plan involves declaring a variable to be used for the counter and incrementing this variable by one after the appropriate condition has occurred. Plan knowledge is classified as strategic, tactical or implementation plan knowledge. Strategic plans are defined to be plans independent of language, having to do with a global strategy. Tactical plans are considered to be language independent but at a more detailed level than the strategic plans. At this level of planning, Soloway maintains that the programmer would consider such items as abstract data structures needed, the action to be accomplished and the output. Implementation plans then deal with the code for this particular tactical plan given a particular programming language. In experimental studies, Soloway discovered that less than 60% of the novice programmers (students in first and second term programming courses were used) successfully completed any of three programs they were asked to write. These three programs emphasized looping constructs. An analysis of the errors subjects made revealed that the majority of the unsuccessful solutions could not be attributed to "silly mistakes". Further analysis also showed that the correct choice of language construct did not predict success in solution generation. However, the correct choice of strategic plan did indeed correlate with success in generating a working solution.

Other researchers have looked at plan knowledge in programming. Rist (1991a) looked at extracting plan structure from program code. Rist makes a distinction between surface plan structure and deep plan structure. We believe that his "deep " plan structure is comparable to Soloway's strategic and tactical plans. His surface plan structure compares to the implementation plans defined by Soloway. A program is composed of many complex plans that are linked together. Complex plans are likewise composed of basic plan units. A basic plan unit possesses the following parts; declaration of variables, initialization (or input) of values, calculation (considered to be the focal portion of the plan) and output. Output may result in a physical print statement or the production of a value which may by used by other plans. Rist presents an algorithm for deriving the plan structure from any given program. Actions or lines of code are connected by data flow and control flow links. The plan dependencies can be one of four types: use, make, obey, or control. Use and make refer to data flow. A calculation line such as *total:= a + b* uses two values, a and b, and makes one value, total. Obey and control refer to control flow. A loop line would control all the lines within its block. The algorithm traces out plan structure by starting from the output and then tracing backward, identifying use links for the variables and control links. See Figures 1, 2 and 3 for sample plan structures obtained using Rist's algorithm. In the plan structure diagrams, control flow is represented by arrows to the left and data flow is represented by arrows to the right.

Rist (1991b) also looked at plan creation and plan retrieval for novice and experienced programmers. Plan retrieval is seen when the elements of the plan appear in a top down fashion. For example, initialization or inputs would appear prior to the focus or calculations line. Plan creation is done by generating backwards from the focal line. Rist found strong evidence for plan creation in solution plans for novices. A shift from plan creation to plan retrieval was shown for the more experienced subjects. Although, even experienced programmers were shown to resort to plan creation for difficult problems.

We wanted to explore several aspects of transfer to a new programming language.

1. If we assume that students have prior programming knowledge in the form of plans, would they attempt to use this knowledge in writing a program in a new language?

2. Would this knowledge help or hinder subjects in producing correct solutions in the new language?

3. Would a correct choice of strategy again correlate with success in producing a correct solution?

4 Would the type of new language have an effect?

In order to find answers to the above questions we conducted exploratory protocol studies with novice programmers writing programs in one of three languages. One language was the language they were all familiar with, Pascal. The other two languages used in this study were Ada and Icon. Our analysis included the following: plan creation versus plan retrieval, percentage of effort devoted to Soloway's three different types of planning, and correlation of this effort to differences in plan structure between programs in the three languages.

Methodology

The subjects used were Computer Science students at either the University of Nebraska -Lincoln or at Portland State University. They had either just finished their first term of Pascal or were just starting the second term. (Pascal was the first language taught at these universities at the time these studies were conducted.) These thirteen subjects were randomly assigned to one of three groups: three students wrote the program in Pascal, five students wrote the program in Icon, and five students wrote the program in Ada. Verbal protocols were collected, transcribed and analyzed to obtain the data discussed in this paper.

The two languages, Ada and Icon, were selected to represent languages not different in type from Pascal. That is, all three languages are procedural in nature. Icon, however, is a string oriented language that possesses many functions that operate on strings as entities. Icon also contains the pattern matching features of SNOBOL. Therefore, tactical plans appropriate in Pascal would not be appropriate in Icon. In this respect, Ada is more similar to Pascal. In this study, the data abstraction features and concurrency features of Ada were not used.

The program that subjects were asked to write a solution to was called the Count A's problem. The following description was given to the subjects:

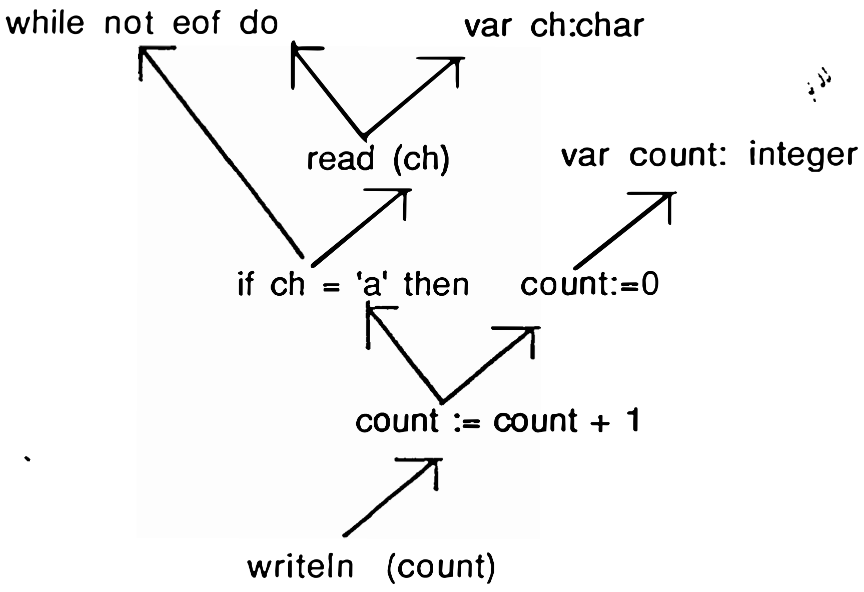
You are asked to write a program which reads in lines of text counting the number of a's and outputting this number.

Subjects were provided with computer facilities and text books on the language in which they were asked to write solutions. Video equipment was used to collect subjects' verbalizations as they "thought aloud" and also to collect the information displayed on the computer monitor during a session. Sessions lasted a maximum of two hours. Subjects were allowed to quit if they became completely frustrated (only one subject took advantage of this option) or if they finished the solution prior to the two hour time limit.

Subjects' protocols were transcribed and analyzed by two independent evaluators. The transcriptions were first broken into episodes. An episode was defined as a behavior distinct from surrounding behavior. Therefore, the start of a new episode could be signaled by a shift in physical behavior. A time frame in which a subject who had been reading the text book and then began writing code would signal the beginning of a new episode. A shift in mental focus would also determine the beginning of a new episode. So a subject who has been looking at for loops and then switches to looking at while loops would exhibit a change in episode. The thirteen protocols contained 1148 episodes. The evaluators then classified each episode as to whether it dealt with syntax, semantics, or one of the three classifications of planning: strategic, tactical or implementation. The evaluators had a 94.8% agreement on their independent classifications. Disagreements were then resolved by discussion between the evaluators. The range on agreement levels was from 88.9% on one Ada transcript to 100% on one Icon transcript.

```
program countas (input,output);
  var count: integer;
     ch: char;
   begin
   writeln ("input text");
   count := 0;
   while not eof do
        begin
        read (ch);
        if ch = "a" then count := count + 1
        end;
    writeln ("number of a's", count)
end.
```

Pascal solution

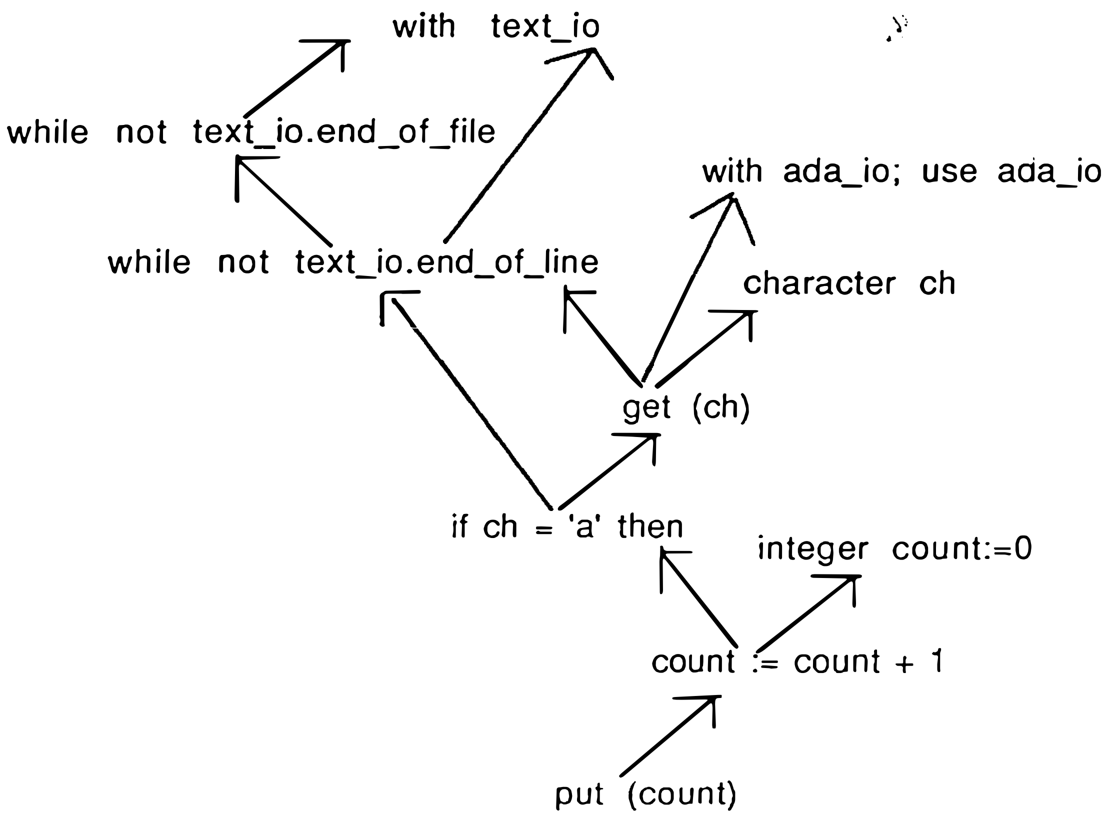


Pascal plan structure

Figure 1: Pascal Solution and Plan Structure

```
with ada_io; use ada_io;
with text_io;
procedure counting is
    count: integer:=0;
    ch: character;
begin
    put ("enter text");
    newline;
    while not text_io.end_of_file loop
     while not text_io.end_of_line loop
         get(ch);
         if ch = 'a' then count := count + 1;
         end if;
      end loop;
     skip_line;
    end loop;
     put ("the number of a's is ");
     put (count);
 end counting;
```

Ada solution



Ada plan structure

Figure 2: Ada Solution and Plan Structure

```
procedure main ()
    count := 0
    while line := read() do {
        every find ("a", line) do
            count +:= 1}
     write ("the number of a's is", count)
end
```

Icon solution

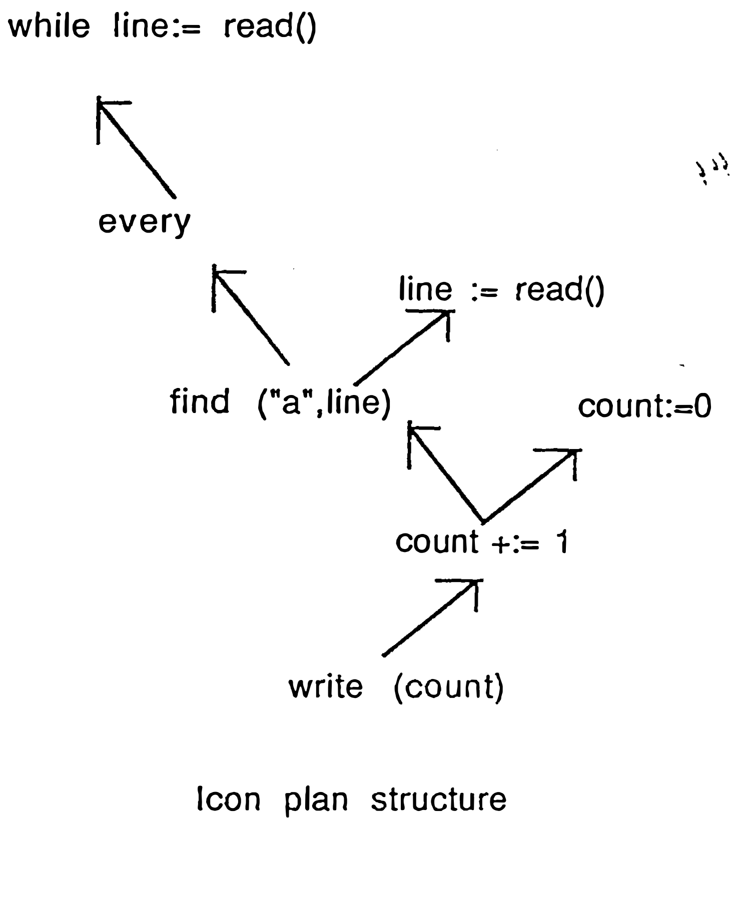


Figure 3: Icon Solution and Plan Structure

Solutions Produced

Figures 1,2 and 3 show one solution to the Count A's problem in each language. Of the three Pascal novices, all three produced a working solution. Two of these solutions closely resembled the solution given in Figure 1. The third subject thought there was a need to retain all input and devised a system of arrays of records with each record containing a line (stored as an array of characters) and the number of characters in it. All of these subjects used an end of file condition to indicate the end of the input. All three solutions contained a label for the output but none prompted the user for the input.

Four of the five Ada subjects produced a working solution. Of these, three used a sentinel character to represent the end of the input. The fourth asked for a specific number of characters to be input and was only attempting to do one of these limited lines. The sole subject trying to use *end_of_file* as a terminating condition did not get his solution to work. Four subjects used input prompts and two used output labels. The input prompts were most likely produced because of the necessity to indicate what the terminating character was.

Three of the five Icon subjects produced working solutions. All of these solutions were similar to the suggested solution in figure 3. A fourth subjects used string scanning and did not get this put into a loop to do more than one line. The fifth subject was attempting to use a sentinel value to terminate input and was using the subscripting feature to examine a character at a time from each line. Four solutions contained input prompts. Three solutions contained labeled output.

It was interesting to note in the protocols that none of the subjects working in Pascal considered using a sentinel character for termination. And only one subject in Icon felt the necessity to do this, having not discovered the success and failure aspect of the read function. In Ada, several subjects did not even attempt to locate an end of file. A few who did quickly gave up and resorted to a sentinel character.

Input prompts and output labels seem to be optional for novice programmers. If this type of input or output seems difficult, it is skipped. This was the case in Ada where *put*, the output procedure, can only be used with one parameter at a time.

Analysis of Program Development

	syntax	semantics	implementation	tactical	strategic
Pascal	6.8%	0	58.4%	26.5%	8.4%
Ada	27.4%	5.2%	52.3%	12.3%	2.8%
Icon	23.7%	17.6%	51.5%	5.4%	1.7%

Figure 4: Percentage of time in each area of program development

All classified episodes were timed. Non classified episodes had to do with concerns about using the text editor or were merely comments uttered by the subjects while waiting for the program to compile. When verbalizations had nothing to do with solving the problem, they were not classified and were not counted in the solution time for the subject. The time that each subject spent in the five areas of program development, the three planning categories previously discussed and syntactic and semantic concerns of the language, were then added up and divided by the total amount of classified time. Figure

4 shows the results of this classification. First of all, we expected that syntactic concerns would consume more effort in the new languages. Subjects working in Icon spent somewhat less of their effort in syntax, most likely because the syntax of Icon is somewhat simplier than that of Ada. Ada has matching keywords, like **if** and **end if** and **loop** and **end loop** while Icon consistently uses curly braces to group statements. The percentage of episodes devoted to semantics is much higher in Icon. This is to be expected as Icon contains many unique functions that students needed to investigate. Implementation planning effort does not differ much from language to language. Pascal subjects spent a greater percentage of their time in tactical and strategic planning than did Ada and Icon subjects. While on the surface this seems contradictory, remember that these Pascal subjects spent little time on syntax and no time on semantic issues. Also, the Pascal subjects were able to exhibit more "thought-out" solutions than were the other subjects.

We constructed "optimal" solutions to the Count A's program in all three languages. An optimal solution to us uses unique constructs that are available in the language to achieve a solution with the most elegance and least possibility of error. Therefore, we consider solutions that decompose and subsequently recompose data structure elements inferior to solutions that operate on the entire data structure. Decomposition followed by composition is more error prone. Figures 1, 2 and 3 contain our solutions and the corresponding plan structure derived using Rist's algorithm. Again, note that in the plan structures shown in those figures, the left arrows represent control flow and the right arrows represent data flow. Rist's algorithm does not incorporate declarations into the plan structure. We felt, however, that the use of a variable in many languages depends on having declared it previous. We felt that subjects would express concerns about this during program development. Therefore we included declarations of variables in our "version" of the algorithm.

Differences in Plan Structure

Looking at the Pascal and Icon solutions and plan structures (figures 1 and 3), the following differences are apparent:

1. In Icon, there is no need to declare variables. A declaration is used only if variables are local to that procedure.

2. In Icon, the auto increment capability of C is used (count+:= 1). However, the longer version (Count : = count + 1) is also syntactically valid.

3. In Icon, *every* is a generator which is used to loop through all characters positions of a string. *Every*, therefore, produces a vector of output.

4. The Icon statement, *while line:= read()* is used for two purposes. *line := read()* reads in a line of input and assigns its value to the variable *line*. The read function also returns an indication of success or failure. That is, if the end of file has been reached, the read function fails, causing the while loop to terminate.

5. *find* is an Icon function that looks in a character string to find the first occurrence of a substring. This function returns either the position of the occurrence or failure.

Looking at the Pascal and Ada solutions and plan structures (figures 1 and 2) there are few large differences. Some minor differences are:

1. In Ada, the procedures *get* and *put* are used for input and output. *Newline* is used to advance to the next output line, whereas Pascal uses *writeln* for writing and automatically supplying a carriage return.

2. In Ada, variables may be initialized at the same time as they are declared. A separate assignment statement (as in Pascal) also serves to initialize variables.

3. The input and output routines in Ada are contained in several packages. In order to use any of these routines, one has to be sure that ***with*** statments have been included that reference those packages. The ***use*** statement indicates that the package name does not have to be declared as a prefix when referencing a routine in the package. This is similar to the dot notation used in Pascal to refer to record fields.

4. The use of an end of file condition in Ada is treated somewhat differently than Pascal. In order to process multiple lines of text using an end of file exception, one must use two loops: an inner one for end of line and an outer loop for end of file. This is necessary as each condition is handled using a different exception handler.

We hypothesized that differences in surface plan structure could be used to predict the kinds of problems programmers would have in attempting to learn a new language. In order to talk about "kind of problem" we will refer to which of the five areas of program development the programmer will need to devote some effort.

We made the following hypotheses about plan structure differences and their effects:

1. If a *makes* or *uses* structure element is missing in the new language, (e.g. declaration statements) the programmer will spend some time validating that these are not needed. This would result in implementation concerns.

2. If the structure of the element maps onto a similar structure but the corresponding statements differ, the programmer will first need to locate the appropriate construct and then understand how it functions, resulting in implementation and semantic concerns. For example, the Ada procedure *get* is used for input as opposed to *read* in Pascal. Programmers need to find the *get* statment (implementation) and verify how it works (semantics).

3. If the control structures of the corresponding elements differ efforts will have to be devoted to tactical plans, that is, a rethinking of how the flow works.

4. If the new language requires *makes* or *uses* structure elements which are not required by the original language, programmers may have difficulty discovering, and using these. This would result in implementation concerns. That is, programmers would attempt to get a particular concept working although its hidden dependency has not been discovered.

To see whether our hypotheses showed any validity we looked at several lines in Icon and Ada which illustrated these structure differences. We then counted the number of episodes in each area of program development per subject for each of these lines. We looked at the number of episodes here rather than time. As we already have an indication of where the majority of the efforts are, we were more interested in seeing the type of concerns expressed for each of these differences.

We looked at the following four differences between Icon and Pascal:

Icon	Pascal	Hypothesis
find ("a", line)	if ch = "a" then	2
while line := read()	while not eof do read (ch)	3
(no declaration needed)	var count: integer; ch: char	1
every	(no corresponding Pascal function)	3

Figure 5: Plan Structure Differences between Icon and Pascal

From our hypotheses we expected that we would see semantic and implementation episodes in *find*, tactical episodes for *while line:= read()*, implementation concerns for the missing declarations, and tactical episodes for *every*.

line	subject	syntax	semantics	implementation	tactical	strategic
find	S1		7	4		
	S2		2			
	S3			1		
	S4		2	6		
	S5	1	5	6		
line= read	S1	1	3	1	1	
	S2		1	4		
	S3	2		6	1	
	S4		3	3	2	
	S5		1		1	
declaration	S1	2				
	S2	1		1		
	S3			1		
	S4			6		
	S5		1	9		
every	S1	2	2	4		
	S2		4	4		
	S3					
	S4			6	1	
	S5					

Figure 6: Number of Program Development Episodes for Plan Structure Difference in Icon

Figure 6 lists the number of episodes per subject per line for Icon subjects. For *find*, there are numerous semantic and implementation episodes, thus supporting hypothesis 2. For *while Line:= read()*, the episodes deal with semantics and implementation but also with tactical concerns, thus supporting hypothesis 3. For declarations, we found implementation episodes, giving support to hypothesis 1. For *every*, we expected to see support for hypothesis 3 and again see more tactical episodes than occurred. This was not the case. A plausible explanation is that subjects in these experiments found several examples of *every* that were easy to extend to the desired result without actually fully understanding the construct. We need to further investigate this.

For Ada, the following differences were examined:

Ada	Pascal	Hypothesis
while not text_io.end_of_file	while not eof	4
get(character)	read(ch)	~~3~~ 2
ada_io	(no correspondence)	4
while not text_io.end_of_line	(no correspondence)	3

Figure 7: Plan Structure Differences between Ada and Pascal

For the references to the IO package, *text_io*, needed to use *end_of_file*, hypotheses 4 predicts that implementation concerns will arise. That is, because *end_of_file* in Ada requires a *with text_io* statement whereas no such statement is required in Pascal. Indeed we did see subjects searching for how to implement this. Subjects who produced working solutions made a decision not to use *end_of_file* and instead used a sentinel character to terminate input, resulting in some tactical and strategic revisions. Had subjects persevered in attempts to use *end_of_file*, more implementation episodes would have occurred. The additional loop on *end_of_line* condition needed for Ada would suggest by hypothesis 3 that tactical episodes are needed. However, as subjects gave up on the *end_of_file* condition, no data was available for *end_of_line*. Subjects simply failed to develop this particular portion of the plan.

The *get* input statement and *put* output statement of Ada are only different constructs from Pascal. Therefore, implementation and semantic concerns will be seen. Indeed, implementation concerns dominate. The one subject showing higher tactical concerns was attempting to read in an entire line but later revised this, thus requiring some tactical planning.

The references to the Ada_io package necessary to use the *get* and *put* procedures have no corresponding requirements in Pascal, therefore hypothesis 4 predicts implementation concerns. This prediction did not hold true. While a few implementation concerns appeared, syntax and semantics were investigated more frequently. A possible explanation seems to be suggested by the protocols. Again, subjects saw examples of programs using the *with ada_io* and *use ada_io* statments and were content to incorporate these into their programs. The questions arose as to what their function was. Most subjects quickly resolved that this was some statement that allowed you to use input and output statements and were content to let it go at that.

Statement	subject	syntax	semantics	implementation	tactical	strategic
end_of_file	S1				1	
	S2					
	S3			3	1	
	S4	2	1	1	1	1
	S5			6		
get/put	S1		1	3	3	
	S2		1	3		
	S3	1		4	1	
	S4	1				
	S5			4		
ada_io	S1		1	1		
	S2		1			
	S3	4				
	S4	1	1			
	S5		2	1		
end_of_line	NA					

Figure 8: Number of Program Development Episodes for Plan Structure Differences in Ada

Although this study was small and exploratory in nature, we feel that the differences in surface plan structure can serve to give some guidance about problems that programmers will have in constructing a solution in a new language.

Another difference in plan structure also exists. The composition or decomposition of plan elements may cause problems. For example, Pascal requires the programmer to use two separate statements to declare and initialize variables. In Ada, an initialization can also be accomplished this way but can also be accomplished by initializing the variable when it is declared. (integer count:=0) Likewise, the autoincrementing feature in Icon (count +:=1) was not discovered as subjects could also use the standard incrementing plan (count := count + 1). These composition /decomposition differences seem to be overlooked in many cases. A question for further investigation is how a programmer eventually discovers some of these unique features. In this study subjects used the two statement approach and never bothered to look further for an alternative approach.

The use of *newline* in Ada to cause the output file to be advanced one line was also investigated. In Pascal, the *writeln* procedure incorporates this automatically. Three Ada subjects found and used *newline*. Two of the subjects used it correctly. The third subject seemed to feel it had something to do with the input file. They had few total concerns however (5 episodes). Implementation concerns made up two episodes with one episode each of tactical, semantic and syntactical concerns. Subjects saw this procedure used in an example and once again, merely duplicated it with few questions.

Plan Creation Versus Plan Retrieval

We also wanted to examine plan retrieval as opposed to plan creation when using a new language. Are novice programmers able to retrieve any plans and use them in a new language? Is this, in fact, an appropriate strategy in transferring to a new language?

Figure 9 shows the plan name and the corresponding statements in each language.

Plan	Pascal	Ada	Icon

Read	writeln("input text")
	read(ch)
		put("input text")
		newline
		get(ch)
			write("input text")
			line:=read()

Running total	count:=0
	count:=count+ 1
		integer count:=0
		count:=count+1
			count:=0
			count +:=1

find a	read(ch)
	if ch ='a' then
		get(ch)
		if ch = 'a' then
			line:=read()
			find('a',line)

output	write("number of a ")
	write(count)
		put("number of a")
		put(count)
			write("number of a")
			write(count)

Figure 9: Plans in Pascal, Ada and Icon

Each plan in figure 9 shows an initialization portion first, followed by the focal or calculation line. If the plan appears in this top down fashion, it indicates retrieval. If a subject creates a plan he starts with the focal line and works backward. The plans in figure 9 would then appear reversed, so creation of the running total plan would be seen if *count:= count + 1* appeared prior to *count:= 0*. To examine creation versus retrieval, we noted the subjects' verbalizations about each plan. We did not analyze the actual appearance of the line of code. We felt that if a subject thought about a particular line, even if it was not written down at the time, he, in fact, had entered it into his mental representation of the solution.

	Pascal(3)		Ada(5)		Icon(5)	
Plan	B	F	B	F	B	F
running total	2	1	4	1	4	1
Read	3	0	5	0	4	1
find A	0	3	0	5	2	3
output	0	3	4	1	4	1

Figure 10: Plan Creation and Retrieval in Each Language

Figure 10 shows the number of subjects in each language producing the plan forward (F) indicating retrieval or backward (B) indicating creation. Note that the Pascal group contained only three subjects.

The two plans where the most differences were shown were the find A and output plan. In the input and output plans a missing prompt or label was scored as appearing later than the focal line. Therefore, as a backward plan or creation of a plan. Pascal subjects all produced a labeled output in a forward fashion. This was not true of the Ada and Icon subjects, who either did not label the output (50%) or created the plan(50%). The find A plan was retrieved and used for both Ada and Pascal. For Icon, three subjects did the plan in a forward fashion. Of these three, only one was able to produce a working solution. Two subjects created the plan. Both of these subjects were able to produce a working solution closely resembling the suggested solution. It is interesting that the find A plan was the plan retrieved more of the time than any other plans regardless of language. The other plans were often created due to the initialization line portion either appearing later of not appearing at all.

We also looked at the order in which each plan making up the loop of the solution was generated. A top down generation would result in the following order:

loop control
read
find A
running total

Notice that the read plan is separated with the initialization or prompt for the input outside the loop control and the actual read statment contained within the loop.

The find A plan is considered the calculation or focal point of the complex plan so a creation of the complex plan would start by looking at this basic plan. For Pascal, only one subject out of the three showed retrieval for this complex plan. The other two developed the enclosed basic plans in order, then put the loop around them. For Ada, one subject out of five was able to do this in a forward fashion. Two subjects used a backward development. A fourth subject developed the body of the loop in order, then enclosed it with the control statements. The fifth did a mixed order, developing the read plan first, then the remainder in a top down fashion. For Icon three subjects used a strictly bottom up or creation approach. All three of these subjects produced good, working solutions. The remaining two subjects, neither of whom produced working solutions showed either forward or mixed development. In Icon, it seems that subjects, who looked at the focal line first discovered the *find* function and the generator, *every*, were more likely to produce a working solution. Again, this is an exploratory study but this trend bears further investigation. Perhaps as one moves to languages that are less similar to familiar languages, the bottom up or creation of plans and complex plans allows the programmer to discover more suitable plans.

An Interesting Observation

The Icon language also contains a *reads* function. This function is used to read a specified number of characters from the input, failing if that many characters do not exist between the current position in the file and the end of file. The default number of characters is one. An alternative solution to the Count A's program would then be the following:

```
procedure main()
          count :=0
          write("input, please")
          while ch := reads() do
               if ch = "a" then count +:= 1
          write ("The number of a", count)
          end
```

This solution produces a plan structure very similar to the Pascal plan structure. Interestly, none of the novice programmers considered using *reads*. Even the one subject who decomposed his line into single characters to check for a's did not consider reading in a single character. Although both *read* and *reads* are described in the appendix of the Icon text, only *read* is used in early examples in the text. This tendency of subjects to reuse code that they see in examples needs to be investigated. What is the interaction, if any, of code reuse and the programmers' plan knowledge?

Conclusions

This study has perhaps raised more questions then it has answered. Although exploratory in nature, definite trends have been revealed in the analysis. First, it seems as if plan structure differences can be used to predict areas of program development that programmers will have concerns about. This assumes a programmer's capability to produce a working, stereotypical solution in the familiar language. Will this generalize to other languages, such as object oriented and declarative? Is the same true of more experienced programmers? This study suggests that as languages become more distant a bottom up or creation model is likely to lead to more success. Again, this needs to be investigated more thoroughly and with respect to language type and expertise.

This study suggests that perhaps a new approach to automated tutoring systems is needed. The model for tutors currently being developed is reactive (Rich and Waters,1990; Johnson and Soloway, 1985; Anderson and Reiser, 1985). That is, once the tutor observes that the student has deviated from the correct plan, it steps in to help. We suggest that a proactive tutor might be more beneficial especially in the early stages of learning. Rather than waiting until an error or deviation has occurred, a proactive tutor would converse with the student and help the student formulate optimal plans initially. The plan structure differences could be used as a basis for building a tutor that would predict the appropriate level of help given. When the tutor sees plan differences in control flow, the tutor might step in and help the student in plan creation. Rather than presenting the new plan in a typically top down fashion, the bottom up or creation model might be more acceptable and natural to the student.

References

Anderson, J. and Reiser, B., (1985). The LISP Tutor. *Byte*. 10(4). pp 159-178.

Bonar, J. and Soloway, E. (1988). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. in E. Soloway and J. Spohrer (Eds.) *Studying the Novice Programmer*. (pp 325 -355). Hillsdale, N.J.: Erlbaum.

Johnson, W. and Soloway, E. (1985). PROUST. *Byte*. 10(4). pp 179-192.

Rich, C. and Waters, C., (1990). *The Programmer's Apprentice*. New York, N.Y: ACM Press.

Rist, R. (1991a). Search through Multiple Representation. *NATO ARW User Centered Requirements for Software Engineering Environments*. Bonas, France. (to be published as an ARW proceedings)

Rist, R. (1991b). Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction*, Vol. 6, No. 1, pp. 1-46.

Rist, R. (1989). Schema Creation in Programming. *Cognitive Science*, 13,3,pp. 389-414.

Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J. (1984). What do Novices Know About Programming? in A. Badre and B. Shneiderman (Eds.) *Directions in*

Human-Computer Interaction (pp 27-54). Norwood: N.J.: Ablex.

Soloway, E., Bonar, J. and Ehrlich, K. (1988). Cognitive Strategies and Looping Constructs: An Empirical Study. in E. Soloway and J. Spohrer (Eds.) *Studying the Novice Programmer* (pp 191-208) Hillsdale, N.J.: Erlbaum.

Spohrer, J. and Soloway, E. (1988). Novice Mistakes: Are the Folk Wisdoms Correct? in E. Soloway and J. Spohrer (Eds.) *Studying the Novice Programmer* (pp 401-416). Hillsdale, N.J.: Erlbaum.

Spohrer, J., Soloway, E. and Pope, E. (1988)., A Goal/Plan Analysis of Buggy Pascal Programs. in E. Soloway and J. Spohrer (Eds.) *Studying the Novice Programmer* (pp 355-400). Hillsdale, N.J.: Erlbaum.