# Understanding the Bugs of Novice Programmers

Jeffrey Bonar

COINS Technical Report 85-12

University of Massachusetts
Computer Science Department
Amherst, MA 01003

April 1985

This work was primarily supported by the National Science Foundation under NSF Grant SED-81-12403. Currently, the author is supported by the Office of Naval Research under Contract Numbers N00014-83-6-0148 and N00014-83-K-0655.

## Table of Contents

## List of Figures

## List of Tables

## Abstract

Why do people have trouble learning to program? We present a theory of novice programming bugs motivated by interviews with novice programmers solving simple programming problems. Novice programming knowledge is represented both with fragments of expert programming knowledge (PK) and step-by-step Natural Language procedural knowledge (SSK) representing the experience a novice brings to programming from Natural Language. Both kinds of knowledge are represented with programming **plans**. When a novice is programming and encounters a gap or inconsistency in the PK, he or she has reached an impasse. The theory proposes **bug generators** as strategies for patching the impasse and continue developing a solution. Usually this patch introduces a bug. The theory is evaluated by analyzing the interviews of novice programmers. A key set of predictions from the theory are formulated and an analysis of the protocol data is described. Based on an analysis of four protocols the predictions of the theory are supported.

## 1. Introduction

Why do people have trouble learning to program? Specifically, is there a way to systematically explain the bugs[1] made by novice programmers. We present a theory of novice programming bugs motivated by interviews with novice programmers solving simple programming problems. While much of the programming methodology literature focuses on mathematical formalization of programming, we instead focus on understanding the knowledge that novice and expert programmers bring to a programming problem. Our key idea is that many novice programming bugs can be explained as systematic patterns of inappropriate knowledge use. In particular, the novice has inappropriately used knowledge of writing step-by-step procedural specifications in Natural Language.[2]

Most modern programming textbooks contain discussions of "structured design", "stepwise refinement", and other ideas that have emerged from professional programming methodology. We feel that weak novice understanding of these design techniques cannot account for most

[1]This usage of "bug" reflects the common usage in computer science: an error in a written program. In cognitive science literature the term "bugs" is more commonly used to refer to errors in a persons procedure for performing a skill. For example, Brown and VanLehn [Brown and VanLehn 80] and Resnick [Resnick 82] have discussed bugs in children's multi-column subtraction algorithms. Confusion can arise in this work because we are concerned with both mental "programs" and computer programs.

[2]Throughout, the term Natural Language will be used to refer to the language in which step-by-step procedures are written. "English", the other obvious choice, was not used because it unnecessarily implied that the novice programming phenomena discussed here were limited to English.

As we discuss below, these procedures are often not "step-by-step" in a strict sense. For example, some steps modify earlier steps or establish a global condition to be checked.

novice bugs. While clearly an important part of programming education, the techniques seem at too high a level for novices who are still struggling with, for example, correctly specifying a loop. Often, the design techniques simply do not make sense to novices. Novices often report that a piece of pseudo-code "looks reasonable" but "its not the way I would have done it." Novices do not understand the principles that allow one to produce useful pseudo-code. In this article, we attempt to show that novice programmers have difficulty connecting the relatively low-level syntax and semantics of constructs in the programming language and higher-level design concerns.

We call the bridge between programming language syntax and semantics and higher-level design concerns **pragmatics**. Pragmatics is how the language constructs are used to accomplish standard programming tasks. Pragmatics capture the role played by a piece of code. Pragmatics is the information captured by good comments. It is the level of detail used by programmers talking to one another: "Well, here we loop through the table, building up a total until we see the sentinel value". An appreciation of programming pragmatics is missing from most programming courses and texts. Much of the work reported here is concerned with characterizing programming pragmatics more closely. In particular, pragmatic programming knowledge will be captured in frame or schema-like structures called **programming plans**. We will describe programming plans that capture pragmatic notions like "counter variable" and "sentinel controlled loop."

It is not surprising that pragmatics is often overlooked in programming instruction. Investigations of many domains have revealed that experts have critical **tacit** knowledge about the role and purpose of various primitives. In physics, for example, experts categorize problems according to abstract physics principles (e.g. "conservation of energy") while novices categorize the same problems by literal features (e.g. "inclined plane") [Chi et al. 81]. Consider an example from a programming textbook. The following is a typical explanation of the **repeat** and **while** looping constructs in Pascal:

> The action of a **repeat** structure will take place at least once; the **while** loop's action may not be executed at all. The **repeat** structure's exit condition, when met, causes looping to **stop**. The **while**'s entry condition, when met, causes looping to **continue**. (page 187) [Cooper and Clancy 82]

This is a relatively low level explanation that assumes the reader understands the situations in which an "action . . . will take place at least once" etc. Consider, in contrast, an explanation related to the *roles* played by these constructs:

> The while construct is for loops controlled by a new value either read or created within the loop. The repeat construct is for loops controlled by a value built up within the loop (like a running total).

(See Soloway et al. [Soloway et al. 82a] for a complete discussion of how novices use the three Pascal looping constructs.) The standard explanation is correct, of course, but at the level of semantics, not pragmatics.

Along with our focus on pragmatic programming knowledge, we focus on bugs. In particular, bugs arising out of missing or mis-applied pragmatic knowledge. Bugs are a powerful window into a novice's understanding of a domain. Bugs appear where a novice has missing or inaccurate knowledge. We discuss bugs in elementary programming, and focus on mismatches between what a novice knows from non-programming experience and the pragmatic programming knowledge that the novice needs to learn. In particular, we find that a novice uses experience with step-by-step Natural Language procedures to supply missing knowledge about programming language pragmatics. This and other similar novice strategies for dealing with missing or mis-applied pragmatic knowledge are called **bug generators**.

A bug study of programming is challenging because of the richness of the programming domain. Problem solutions typically involve many lines of code and not a single numerical answer, as in a domain like elementary arithmetic. Programming also requires several kinds of knowledge about both the problem and how to use the programming language.

## 1.1. Motivation

In this section we motivate our theory of novice programming bugs. The theory was developed to provide a systematic account of novice bugs. Previous explanations typically were based on **bug stories** - plausible descriptions of the bugs occuring in a program - and lacked a systematic or empirical basis. In this section we illustrate the problems with bug stories. We next present a series of patterns seen in the work of novice programmers. Our theory of novice programming bugs is built on these patterns.

In cognitive science it is commonly assumed that bugs often have a rational basis [Chi et al. 82, Clement 82, Pollatsek et al. 81, Resnick 83, Young and O'Shea 81]. One can demonstrate rational bug explanations with **bug stories** [Brown and VanLehn 80] - plausible explanations for a novice bug. Bug stories have been developed to explain the bugs found in novice programs Bonar [Bonar 79]. Figure 1-1, for example, includes a simple programming problem, a buggy

## The Ending Value Averaging Problem

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

## The Buggy Novice Program (actual novice program)

```
Program Student5problem5 ( Input, Output);
Var num, nextnum, count, s : integer;
    av : real;
Begin (*program*)
    num:=0;
    nextnum:=0;
    count  :=0;
        IF nextnum <> 9999
        THEN Begin (*IF THEN*)
            read (nextnum);
            count:=count+1;
            S : = num + nextnum;
        END   (*IF THEN*)
    ELSE
        Begin (ELSE*)
            Av:= S/count;
            writeln (av:6:2)
        END  (ELSE*)
END.  (*program*)
```

## The Bug Story

In this program the student used the Pascal **if then else** construct as if it were a looping construct. This stems from people using the Natural Language phrase "if then" to imply looping. For example: the phrase "see that hallway, *if* a locker is open, *then* close it" implies looping through all the lockers.

Notice that the running total is not really being accumulated in s. Instead, the assignment to s reflects reasoning like "to get the next sum, add the current number (num) in s and the next number (nextnum) that has just been read".

Note finally, how, even if s were getting the correct value, it and count would be off because nextnum, the value read in, is not tested until after it has been counted and summed. This is due to novice preference for the pattern of *read a value, process that value.* The appropriate pattern for solving this problem in Pascal - *process the last value, read the next value* - enables the program to test a value before it is incorporated into the sum and counted.

**Figure 1-1:** Example Bug Story For a Novice Program

program from an actual subject, and plausible explanations for the bugs in that program. The problem presented requires the subject to read in a series of numbers, watching for an ending value of 99999, and producing the average of the numbers read in before the 99999. Producing the average requires the student to accumulate both a sum and count of the numbers read. The program produced by the subject has three basic difficulties. For simplicity, each difficulty is described as if everything else in the program is correct:

1. Where the loop should be the student has used an **if then else** statement. The test is the correct test for the loop, the **then** branch contains the main elements of the loop body, and the **else** branch contains the code that should be below the loop.

2. The summation of the numbers read is not done correctly. Numbers are read into the variable **nextnum** and are incorrectly summed with the statement S := **sum + nextnum**.

3. The loop sums and counts the terminating 99999, only testing for it's presence after the summation and count.

In the bug story we attempt to explain the bugs in the program. We do this by inferring a purpose the novice intended for each component of the program. For each bug we then propose an explanation for in terms of preferences the novice has for certain incorrect implementation techniques. Unfortunately, there is no systematic or empirical basis for infering those particular purposes and preferences. While valuable for developing hypotheses about novice programming behavior, bug stories must give way to a more systematic theory to account for how each bug occurs. Here we present the beginning of such a theory.

The theory was developed after several years working with college student novice programmers. Our background work included teaching introductory programming courses, writing an introductory programming text [Bonar 85], detailed study of novice programs that compile correctly but contain run-time errors [Bonar et al. 82, Johnson et al. 82]. The specific work reported here is based on a detailed study of video-taped interviews with novice programmers solving programming problems. (See [Bonar 84] for a complete discussion.) These interviews were first used to develop the intuitions that lie beneath the theory presented in this chapter. Later, they were systematically analyzed to refine the theory.

Preliminary work with interviews of novice programmers solving programming problems revealed four patterns:

1. **Novice programming knowledge is fragmentary.** We have observed that novices know only some programming knowledge, and what they know is organized into incomplete clusters or fragments. This is not a surprising observation: if their knowledge consisted of more than fragments, they wouldn't be novices. Furthermore, the observation is consistent with a large body of work on novices learning in many different domains (for a summary of this work see Resnick [Resnick 83]). Work in these other domains has also shown that what novices do know is stored in clusters, not as isolated bits of information. We make explicit this non-surprising observation about fragmentary novice programming knowledge for two reasons. First, it is worth noting that novices studying programming show similar cognitive behavior to novices learning physics, geometry, or etc. Second, the theory presented below relies on the notion of gaps in the novice knowledge. These gaps are the knowledge not in the fragments.

2. **Novice programming knowledge is often organised pragmatically.** We have observed that novices often know more about their programs than is apparent in the programming code they produce. In particular, they seem to organize their understanding of the program pragmatically, describing the role of various statements and variables in the program. This is in contrast to their weak understanding of the syntactic or semantic features of the program statements and variables.

To clarify these distinctions, consider the following example statements from Pascal:

| | |
|---|---|
| A counter: | `Count := Count + 1` |
| A running total: | `Total := Total + New_Value` |

These two statements are quite similar. Syntactically they both have the same form:

*Variable* := *Variable* + *Value*

Semantically they both use assignment statements, and both involve adding to a variable and then replacing the value of that variable with the sum. Where they differ, however, is in their *pragmatics*: the two statements have quite different roles. The counter is typically used to count iterations of a loop while the running total is used to build up a sum based on successive new values. While novices may be confused about the syntax and semantics of a programming language statement, they are usually clear about the tactics they intend for that statement. One of our subjects, for example, correctly noted that in the Fixed Count Averaging Problem (see

Figure 1-2, the problem asks for the average of ten numbers read from the user), the counter assignment (i.e. `Count := Count + 1`) "keeps the loop under control" while the running total assignment (i.e. `Sum := Sum + New`) "has something to do with something you are gonna . . . take out of the loop with you".

---

Write a program which reads in 10 integers and prints the average of those integers.

**Figure 1-2:** The Fixed Count Averaging Problem

---

3. **Novice programmers frequently reason about programming statements from knowledge of step-by-step Natural Language procedures.** Novice programmers often seem to reason about their programs using their experience with step-by-step Natural Language procedures (e.g. directions to someone's house, assembly instructions).[3] The Factory Gate Problem (shown in Figure 1-3) asks the subject to write a step-by-step Natural Language procedure for a junior clerk to collect payroll information from workers coming out of a factory gate. The clerk needs to report on the average salary for all the workers that leave the gate before the first supervisor leaves the gate. This problem was designed to parallel the Ending Value Averaging Problem (shown at the top of Figure 1-1) in terms of requiring an average and having a distinguished input value used to stop the loop.

---

Please write a set of explicit instructions to help a junior clerk collect payroll information for a factory. At the end of the next payday, the clerk will be sitting in front of the factory gates and has permission to look at employee pay checks. The clerk is to produce the average salary for the workers who come out of the door. This average should include only those workers who come out before the first supervisor comes out, and should not include the supervisor's salary.

**Figure 1-3:** The Factory Gate Problem

---

[3]Note that these procedures are not strictly step-by-step, though they often take the form of a numbered list of statements. In actual usage, a number might label one step, several steps, or even a "step" that actually modifies one or more previous numbered statements.

a. Identify worker, check name on list, check wages
b. Write it down
c. Wait for next worker, identify next, check name, and so on
d. When super comes out, stop
e. Add number of workers you've written down
f. Add all the wages
g. Divide the wages by the number of workers

(This procedure written by Subject 13.)

**Figure 1-4:** Typical Answer for the Factory Gate Problem

There are both similarities and incompatibilities between the conventions of step-by-step Natural Language procedures and the conventions of computer programs. A sample solution to the Factory Gate Problem, shown in Figure 1-3, illustrates several conventions of step-by-step Natural Language procedures. In particular, consider step 4, the stopping condition for the loop. This condition is phrased as a continuously active test, always watching the action of the loop for the exit condition to become true.[4] This kind of control structure is unusual in a programming language. More typical is a construct where the loop condition gets tested once per loop iteration, e.g. the `while` loop in Pascal. In Natural Language, the word "while" is more typically used as a continuously active test, as in: "**while** the highway stays along the coast, keep following it north". Novice programmers are easily confused by this difference between programming languages and Natural Language. One of our subjects even inferred a semantics for the continuously active test `while` loop: "every time [the variable tested in the `while` condition] is assigned a new value, the machine needs to check that value . . . ".

4. **Novices patch programming impasses.** The fourth pattern we noticed is in the way novice errors are actually made. Novices often indicate when they've reached a topic where their programming knowledge is weak. They say things like:

"I'm not sure here"
"I don't think I can use that."

[4] This is sometimes refered to as a "demon" control structure.

"I'm sure I'm leaving something out here"

At these points, we say the novice has reached an **impasse**[5] . When novices reach such an impasse they often propose a series of possible corrections to get past the impasse. These corrections usually produce a bug, and the novice usually knows that the corrections are suspicious. Because of they are likely to produce bugs, we call these corrections **patches**. These patches sometimes yield correct programs, but often introduce errors.

As an example of these four patterns, consider Subject 13 working on the Ending Value Averaging Problem (see the problem statement at the top of Figure 1-1, page 4).

First of all, he is clearly at an impasse. When constructing the counter statements inside the body of a repeat loop, he trys several different forms. It seems that he knows how such a counter should work (see segments 115 and 121 below), but does not know the right form in Pascal. Finally he settles on a form, but adds "its not right, I don't think, but I'm gonna leave it that way for the moment". In the following, N is the counter variable and I is the new value (Read) variable. During the time discussed in the following protocol he works on the following code:

```
N := 0
Repeat
   N := ~~1~~← I   (THE 1 WAS CROSSED OUT)
```

108 **Subject 13:** N equals, I have here [points to initialization of N] N equals 0

109 **Interviewer:** Yeah

110 **Subject 13:** So N equals zero plus one, and then, N equal, ehh, and that would [looking at initialization step, motioning "next"[6] ] reach number N and, all right, we'll try it that way.

111 **Subject 13:** N equals, ahh, [**WRITES:** N := 1] N eq, ahh, N is the number

112 **Interviewer:** Uh huh

113 **Subject 13:** And so, I want [pause]

---

[5]We use this term in a similar way to Brown and VanLehn [Brown and VanLehn 80]

[6]Motioning as follows: The hand starts palm down and near the body. It then describes a 180 degree arc of a circle, ending up with the palm up and away from the body. During this motion, the index finger is pointing slightly. Several subjects used this motion.

114 **Interviewer:** What are you thinking now?

115 **Subject 13:** I want to get a statement that is going to be clear that we're going to add the numbers [points to *N* := *1*], each number entered, we'll have the tally of the, number of integers entered.

116 **Interviewer:** Ok, and so, whats going to do that?

117 **Subject 13:** Ahhh, N equals, ahmm [pointing], integer [changes the *1* in *N* := *1* to *I*]

118 **Interviewer:** Ah huh, how will that work?

119 **Subject 13:** No, N, well,

120 **Interviewer:** What are you thinking now?

121 **Subject 13:** Not to have this [points to *I* in *N* = *I*] be the num, the actual value of the integer, but the, the ahh, frequency of the integer

122 **Interviewer:** Ok

123 **Subject 13:** Ahhh, so if I put, N equals 1, or N equals integer [points to *I*], and that's not gonna necessarily do it, because the integer, we're gonna repeat this

124 **Subject 13:** Ummm, ummm, well I think its, ahh, its [*N* := *I*] not right I don't think, but I I'm gonna leave it that way for the moment.

Subject 13 seems to understand the role of the counter, *N*. In addition, he knows that the value of *N* will increase by one for each number entered. He does not know how this is implemented in Pascal. The transcript shows him trying several solutions to this impasse. First he talks about the execution behavior "... N equals zero plus one and then ..." (segment 110), and writes *N* := *1* to implement the counter in the loop body. Next, he wants a "... tally of the, number of integers entered" (segment 115) and changes *N* := *1* to *N* := *I*. Next, he discusses that the *I* is wrong because he doesn't want "the actual value of the integer, but the, the ahhh, frequency of the integer" (segment 121). He makes no further changes in this excerpt from the protocol. He settles on *N* := *I*, but adds "it's not right I don't think, but I'm gonna leave it that way for the moment" (segment 124).

This transcript also illustrates the other three patterns. Subject 13 knows something about implementing counters and running totals in Pascal (pattern: fragments of programming knowledge). Even though he is confused about their implementation, he reasons about the pragmatics of the counter and the variable receiving the new value read (pattern: knowledge organized pragmatically). He also shows several examples of reasoning based on step-by-step Natural Language: "we'll have the tally of the, number entered" (segment 115), "the frequency of the integer" (segment 121) (pattern: reasoning based on step-by-step Natural Language procedural knowledge).

The theory of novice programming bugs is an embodiment of these patterns. In the rest of the article we begin by presenting the theory. Next we detail a representation of the plans and bug generators. We present evidence for our theory from analyzed protocols of novice programmers at work. We conclude with a discussion of the implications of this work.

## 2. A Theory of Novice Programming Bugs

In this section we present the two key aspects of our theory of novice programming bugs. First, we propose a **representation** for Novice programming knowledge based on the patterns of:

1. fragmentary novice programming knowledge,
2. pragmatic organization of programming knowledge, and
3. step-by-step Natural Language procedural knowledge.

Second, we describe the **process** by which novices produce bugs, based on the pattern of novice impasses and patches.

### 2.1. Representation of Novice Programming Knowledge

The theory of Novice programming bugs begins with a representation for novice programming knowledge. The representation encompasses both kinds of knowledge discussed above: step-by-step Natural Language procedure knowledge and fragments of programming knowledge. The representation is organized pragmatically - that is, with reference to the role and purpose of each chunk of programming knowledge. Finally, the representation is designed to allow a novice to reason about his or her programs with step-by-step Natural Language procedural knowledge. This is done with connections between the step-by-step Natural Language procedural knowledge and the programming knowledge.

A schematic for the representation of novice programming knowledge is shown in Figure 2-1. The three major parts are the **Novice Fragmentary Pragmatic Programming Knowledge** (abbreviated as "PK"), the parallel **Novice Step-By-Step Procedure Natural**

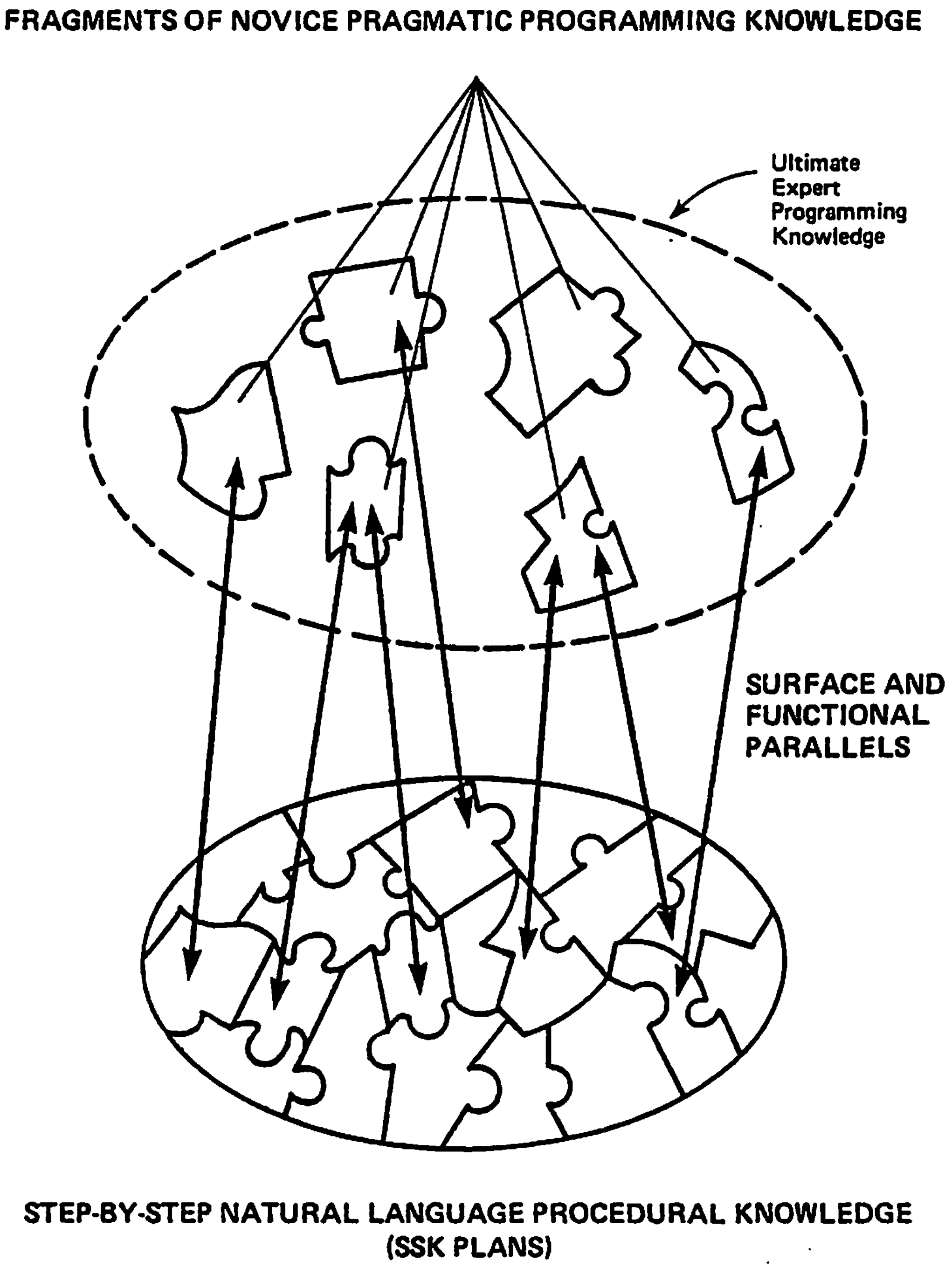
FRAGMENTS OF NOVICE PRAGMATIC PROGRAMMING KNOWLEDGE
Ultimate Expert Programming Knowledge
SURFACE AND FUNCTIONAL PARALLELS
STEP-BY-STEP NATURAL LANGUAGE PROCEDURAL KNOWLEDGE (SSK PLANS)

**Figure 2-1:** Representation of Novice Programming Knowledge

**Language Knowledge** (abbreviated as "SSK"), and a set of **Functional and Surface Links** that specifically relate components from the two sets of knowledge. The PK represents the pragmatic fragments of programming knowledge learned by the novice. We call each of these pragmatic fragments a **plan.**[7] The PK is some fraction of the complete set of programming knowledge known to a programming expert. It contains a number of relatively disconnected plans representing the fragments of novice knowledge about programming discussed above. (This is indicated in Figure 2-1 by drawing the PK plans as isolated tiles and small clusters of tiles in the larger circle of complete expert knowledge.)

The novice SSK represents what the novice knows about step-by-step procedures in Natural Language. Since novices have all the knowledge they need to write step-by-step procedures in Natural Language, SSK is not fragmentary. (This is indicated in Figure 2-1 by drawing the SSK plans as tiles completely covering the larger circle of complete step-by-step Natural Language procedure knowledge.) Like PK, SSK knowledge is represented in pragmatic chunks called **plans.**

These two sets of knowledge parallel each other in two ways. **Functionally**, there are parallels between pieces of knowledge that accomplish similar functions in the two domains. For example, there is knowledge about how to do looping in both the SSK and PK. **Surface** parallels are based on common lexical features. For example, a keyword in the PK might be connected to the corresponding English word. Although the word is the same, the semantics can be different. Consider the English phrase "See that hall, **if** a door is open, **then** close it. In this case the English implies a loop, while in Pascal `if ... then` is a conditional construct. In the representation, there are **functional and surface links** to capture these two kinds of parallels. Although we distinguish between these two kinds of links, a critical component of the theory is that a novice usually does not make this distinction.

We now describe each component of the representation in detail.

## 2.1.1. Fragments of Pragmatic Programming Knowledge

PK represents the knowledge that allows novices to write some parts of a program correctly. Consistent with the pattern of fragmentary novice programming knowledge, the PK consists of some plans and structure from an expert's pragmatic knowledge. Organizing PK as

[7] "Plan" is used here in the sense of "a scheme ... for making, doing, or arranging something" [Webster 78]. Using the word "plan" to refer to programming knowledge structures is from Rich and Schrobe [Rich and Shrobe 78]

plans, a kind of pragmatic fragment, is consistent with other work on the structure of expert programming knowledge [Ehrlich and Soloway 83, Rich 81, Soloway et al. 82a]. It is also consistent with a common cognitive science and Artificial Intelligence view of knowledge [Minsky 75, Resnick 83]. Nonetheless, there is little direct evidence about the specific cognitive structure of PK plans.[8] As part of this work we have constructed a database of approximately 40 PK plans that cover elementary novice programming. This set includes many plans we expect a novice to know for variables, assignment, conditionals, loops, input and output (see Bonar [Bonar 84] for the complete list). We do not consider this set definitive. Instead, it is a reasonable partitioning of the knowledge an ideal novice would know after an introductory programming course. A smaller set, used in the protocol analysis described in Section 4, is described in Appendix I.

The PK is designed to represent the subset of programming knowledge actually possessed by a novice. We have not designed the PK to represent inaccurate or errorful knowledge about programming. In our theory, bugs are the result of processes called **bug generators**, presented in Section 2.2. The errorful or inappropriate reasoning that a novice uses when making programming errors is modeled as a dynamic process involving the PK, SSK, and the links between them.

### 2.1.2. Novice Step-By-Step Natural Language Procedure Knowledge

In parallel with Fragmentary Pragmatic Programming Knowledge (PK), novices also use step-by-step Natural Language procedure knowledge (SSK). Like PK, SSK can be used to accomplish certain tasks: looping, making choices, specifying sequences of actions. Unlike PK, there is no generally accepted paradigm (and very little discussion) about how SSK knowledge is organized. In order to better understand SSK, we conducted a study of non-programmers writing step-by-step Natural Language procedures. The study is discussed in detail in Bonar [Bonar 84].

A key feature of our study of step-by-step Natural Language procedures is that SSK, although capable of characterizing step-by-step tasks, is less formal than the PK. This is not surprising since SSK is used by humans and intended for humans. Humans can bring all their Natural Language understanding skills to bear on understanding such step-by-step procedures. Aspects of SSK informality include flexibility of lexicon and syntax, and explanation on several different levels of detail. For example, consider the Factory Gate Problem discussed earlier (see

[8]Ehrlich and Soloway [Ehrlich and Soloway 83] *do* present a methodology for viewing the cognitive structure of plans and some results from early studies with this methodology.

Figure 1-3 on page 7) and the typical response (see Figure 1-4 on page 8). SSK flexibility is illustrated in lines 5 and 7 of Figure 1-4. The subject has used the word "Add" in both lines. In line 5 it means "count" and in line 7 it means "sum up". Different levels of detail are also illustrated in that line 4 actually modifies the previous three steps. Also, those steps both perform the first iteration of the loop and imply the other iterations.

Given the informality, is SSK actually used effectively and understood? People reading Natural Language procedures like that in Figure 1-4 are able to infer the key features of the solution: an average is desired and only those workers who exit before a supervisor should be included in the average. Since most people describe or carry out step-by-step procedures as part of their daily life, the theory assumes that we are all SSK experts. Complicated tasks stretch the ability of Natural Language to unambiguously express a procedure (see, for example, Miller [Miller 81] or Sime et al. [Sime et al. 77]), but for simple tasks people seem to have no trouble. Notice, finally, that in this work there is a twist on the question "could one program in Natural Language?". Here we are not concerned with whether SSK is able to express procedures with the power of PK, but instead with how a novice is influenced by SSK in their understanding and use of PK.

### 2.1.3. Representing Functional and Surface Parallels

The final aspect of the knowledge representation used for the theory is representing the parallels between the PK and SSK knowledge bases. It is no accident that there are parallels - the programming language Pascal and its parents were designed to be like Natural Language. Many programming texts suggest this connection:

> A *programming language* is a subset of the English language that allow the programmer to give unambiguous commands to the computer. [Zaks 80] (page 1)

This has been even further emphasized in the texts that encourage students to write "English language algorithms" before coding in Pascal:

> Stepwise refinement also lets a programmer plan most of a program *without actually writing in Pascal*. It's easier to think in English than in any sort of computerese. [Cooper and Clancy 82] (authors emphasis, page 82)

Most texts fail to adequately distinguish between functional and surface parallels between the two kinds of languages. Functional parallels exist because both languages are concerned with repeated actions, choice between conditions, counting, and etc. Functional parallels are concerned with the **purpose** of the plans in each of the two knowledge bases. Surface parallels exist because the programming language Pascal (and many others as well) shares many words with Natural Language. Above, we mentioned a subject who made such a surface connection between his "continual test" while loop and the Natural Language usage where "while" refers to a continually active test. In order to represent functional and surface parallels, we use **functional and surface links** between analogous plans. (In Figure 2-1 these links are indicated as lines connecting the PK and SSK circles.) In the next section we discuss **bug generators**, the processes accounting for novice use of links between PK and SSK, and novice production of bugs.

## 2.2. The Process of Generating a Bug

When studying protocols of novices solving programming problems we noticed that there are gaps in the novices' knowledge which lead to **impasses** in their developing programming solutions. When confronted with an impasse in their developing programming solution, novices often propose a series of possible corrections to get past the impasse. These "corrections" usually produce a bug, and the novice usually knows that the "corrections" are suspicious. Because they are likely to produce bugs, we call these corrections **patches**. Because the correctness of computer programs is very sensitive to modifications, most patches will produce bugs. **Bug generators** are patterns of patching that are used when specific knowledge is missing, that is when the novice is at an impasse.

Consider a novice working on a programming problem. This work can be characterized as a path through the Novice PK, using the knowledge in various plans there (see Figure 2-2). At some point though, the novice comes to a gap in the PK knowledge base: he or she has reached an impasse. The novice may, for example, have come to an inconsistency between plans or need information from plans that are not yet acquired (in Figure 2-2 an impasse is depicted as the solution path coming to the edge of a PK plan). The novice needs a way to patch the impasse so that he or she can continue to work in the known parts of the PK (such a patch is shown in Figure 2-2 as a path between known PK fragments). Bug generators are the processes that produce this patch.[9] The programming process just described can be cast in terms of the information processing model of Newell and Simon [Newell and Simon 72]. Plans are the operators that allow the programmer to move between states of the partially formulated program. Each problem state holds some partially formulated program solution. Experts are able to apply an appropriate plan to a partially formulated program state and thereby move to a

[9]Repair theory [Brown and VanLehn 80] uses the phrase "repair heuristic" for the analogous concept.

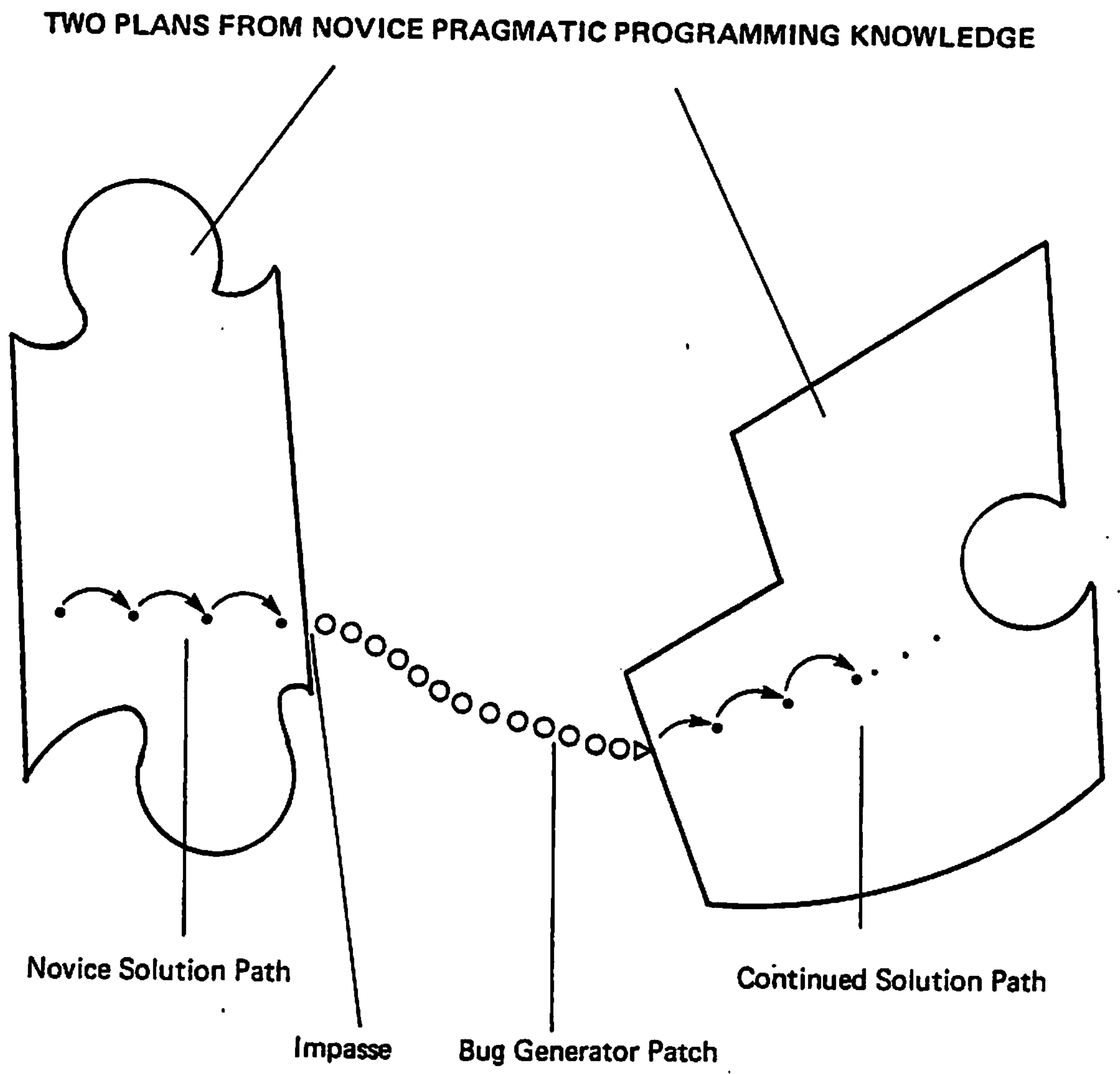

**Figure 2-2:** The Process of Bug Generation

state with a more completely formulated program. Impasses occur when novices do not have a relevant plan. In this case they use bug generators - operators based on non-programming knowledge or other heuristics.

Bug generators operate on available knowledge: the plans known to the novice at the time of reaching an impasse. In effect, the bug generator is parameterized by the plans which the novice does know. Depending on the type of bug generator, the knowledge in these plans is exploited in some heuristic way to create a patch. Of key interest are the *step-by-step Natural Language procedural knowledge confounded with programming knowledge* (*SSK Confounds PK*) bug generators. These bug generators take a plan currently used by the student and improperly use the surface links between SSK and PK to reason in PK and create a patch.

In order to develop the idea of bug generators, we present a detailed example.[10] Subject 13 wrote `Sum := 0 + I` as part of a running total update inside a loop body. He has identified `I` as the variable to receive new values, entered by the user. He has also identified `Sum` as the variable to hold the running total. (Note that a correct version of this line would be `Sum := Sum + I`.) Subject 13's trouble with forming the running total update is not, however, the focus of this example. We focus on a bug brought out when Subject 13 points to his line and says "it reads the Sum". The only `Read` statement Subject 13 has used has no arguments and is above the loop along with a `Readln`. What, then does the subject mean by "reading the sum"? The bug analysis describes several relevant bug generators and is shown in Figure 2-3. The impasse in this case is that Subject 13 did not know how to implement a read operation for an Input New Value Variable in Pascal. A correct Pascal implementation uses an explicit `Read(I)` in each iteration of the loop. Three plausible explanations, each based on a different bug generator, and one based on a slip interpretation are shown for the bug. (The reason for multiple explanations is discussed below.)

The first Bug Generator plausibly explaining this bug is the *Programming Language used as if it were Natural Language* (*PL Used as NL*) Bug Generator. This bug generator operates on a specific plan or plans, using surface links between SSK and PK versions of the plan. Even though they are surface links, the novice assumes functional links, assuming that the Natural Language semantics can be used for programming language constructs. The programming language construct is assumed to have the similar semantics to the parallel Natural Language

[10]This example is drawn from protocol 1, which is discussed below in detail. An excerpt of that protocol is shown in Appendix III.

**BUG: "It reads the Sum": `Sum := 0 + I`**
The subject should be saying something like "It reads in a value which is added into the sum".

☞ ***PL Used as NL*** **on Input New Value Variable**
The read operation is done implicitly whenever a value is needed.

☞ ***PL Interpreted as NL*** **on Pascal - `Read/Readln`**
Here he is treating `Read; Readln` pair as if they were declaring that reading is to be done, and the results of the read will be used with Sum.

☞ ***Multi-Role Variable*** **on Arithmetic Sum Variable, Input New Value Value**
The subject is using the variable Sum as if it will automatically get a new value added in whenever that new value is read.

☞ ***Slip***
Just slipped and forgot to say ". . . a value which is added . . . ".

**Figure 2-3:** Bug Analysis showing the operation of Bug Generators

construct. In the example with Subject 13, the *PL Used as NL* Bug Generator operates on the *Input New Value Variable* plan. Getting a new value for the *Input New Value Variable* has been implemented implicitly, as in the Natural Language implementation.

The second explanation for the Bug in Figure 2-3 uses the *Programming Language interpreted as Natural Language* (*PL Interpreted as NL*) bug generator operating on the `Read` statement written above the loop. In this case, the subject is seen to be interpreting the `Read` as if it were a declaration statement. (There is support for this explanation in that the Subject discussed the `Read` statement while discussing other declarations.) The *patch* was to interpret the `Read` as if it were Natural Language. In Natural Language one can declare that reading will be done, and have that reading be implicit in the rest of the step-by-step procedure . Note that the distinction between the *PL Used as NL* Bug Generator and the *PL Interpreted as NL* Bug Generator is fine. In the *PL Interpreted as NL* bug generator, the novice uses a programming language construct to implement a Natural Language plan; a `Read` used as a declaration that reading will occur into a certain variable. In the *PL Used as NL* bug generator, on the other hand, the novice uses programming language constructs as if they had their Natural Language meanings or omits programming language constructs that would not be needed in Natural Language; the implicit `Read(I)` every time a data value is needed.

The third explanation for the bug in Figure 2-3 uses the *One variable assumed to have multiple roles* (*Multi-Role Variable*) Bug Generator. In this interpretation the subject patches an impasse about getting new values in the loop by collapsing the pragmatics for two different variables. In particular, he has collapsed the running total to be accumulated with the new value to be read from a user. In this interpretation, he thinks that the running total happens automatically when a new value is read from the user. In particular, the subject understands the statement `Sum := 0 + I` to mean that the variable `Sum` gets the value of `I` added in every time a new value of `I` gets `Read`.

The fourth explanation is a *Slip* Bug Generator explanation. It says that the subject simply spoke sloppily, understanding that a `Read(I)` statement must appear elsewhere.

Notice that it is possible for two or more (non *Slip*) bug generators to produce plausible explanations for a bug. If several different patches produce the same result, there is no systematic way to choose between the different possible bug generators. In fact, it is reasonable that novices could use more than one bug generator in constructing a patch. Given the current methodology, however, we have no way to relate the novice programmer's behavior to possible interactions between plausible bug generator explanations.

The rest of this section presents an overview of the bug generator set. We discuss the three classes of bug generators: *SSK Confounds PK*, *Intra-PK*, and *Other Confounds PK*. For each class we present an example. Table 2-1 presents a summary of our bug generator set.

### 2.2.1. Bug Generators: *SSK Confounds PK*

The links between the SSK and PK domains are the source of this first class of bug generators. *SSK Confounds PK* bug generators capture the process of a novice exploiting a link to move from a PK plan to an SSK plan. Once in the SSK domain the novice reasons about the impasse with these more familiar plans, and develops one or more solutions. Finally, using the links again, the novice moves back to a PK plan and continues the normal process of problem solving. (This is represented pictorially in Figure 2-4).

Earlier we discussed an example where a novice interprets a Pascal `while` loop as continually testing the loop exit condition (not just at the top of the loop). This novice is at an impasse about how the `while` loop actually implements the looping and when its condition is tested. He resolves the impasse by reasoning based on how the word "while" works in SSK, and patching

## Bug Generator Summary

**Based on Similarities between PK and SSK (SSK Confounds PK):**

**Programming Language Used As Natural Language (*PL Used as NL*)**
Inappropriately used a programming language construct because it is has the same words as a phrase used in the natural language implementation of the relevant plan.

**Programming Language Interpreted as Natural Language (*PL Interpreted as NL*)**
Interprets a programming language construct as if it were a phrase in a Natural Language implementation of the relevant plan.

**New Programming Language Construct From Natural Language (*NL Construct*)**
Invents a new programming construct based on a Natural Language implementation of the relevant plan.

**Based on missing PK (Intra-PK):**

**Statements Ordered In Execution Order (*Trace*)**
Orders the program as if it was an execution trace.

**Variables Named Generically (*Generic Name*)**
Parts of the program or variables are named generically, based on common programming language implementation strategies.

**Programming Language Overgeneralization (*Over Generalize*)**
Over-generalizes from one Pascal implementation plan to another.

**Tactical Similarity (*Tactically Similar*)**
Failure to distinguish between things that are similar on a tactical level, but implemented differently.

**Based on Confounds Between Other Domains and PK (Other Confounds PK:)**

**Multiple Roles For a Variable (*Multi-Role Variable*)**
Uses a single variable but gives multiple roles for that variable.

**Knowledge From Other Domains (*Other Domain*)**
Uses an understanding of the problem from a domain like mathematics to produce an incorrect answer.

**Operating System Confound (*OS Confound*)**
Uses or confuses a programming language construct with some command or operation from the operating system.

**Slips**

A random error produced while the novice is distracted; a speech slip; a typographical error.

**Table 2-1: Summary of Bug Generators**

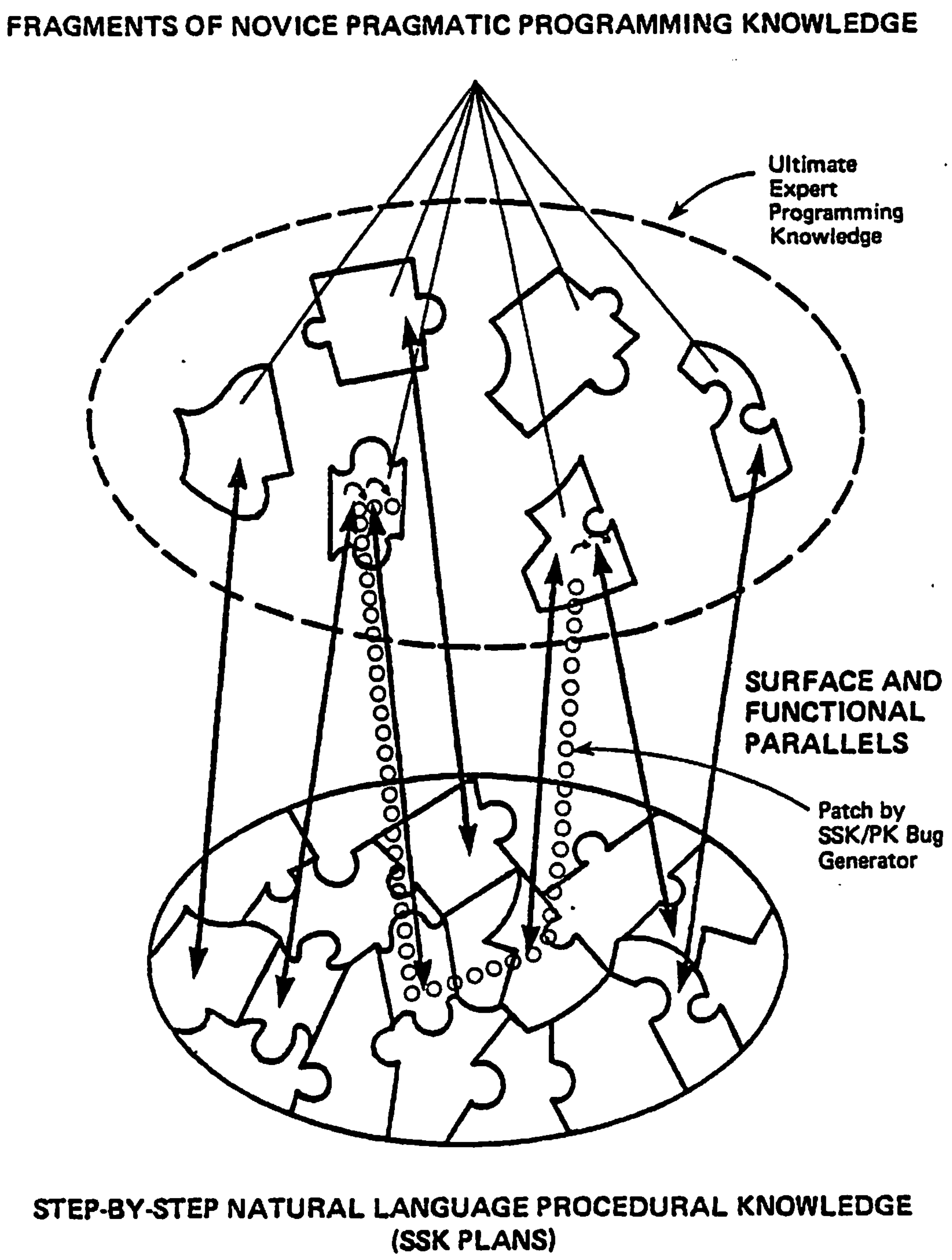
FRAGMENTS OF NOVICE PRAGMATIC PROGRAMMING KNOWLEDGE
Ultimate Expert Programming Knowledge
SURFACE AND FUNCTIONAL PARALLELS
Patch by SSK/PK Bug Generator
STEP-BY-STEP NATURAL LANGUAGE PROCEDURAL KNOWLEDGE (SSK PLANS)

**Figure 2-4:** Bug Generators: *SSK Confounds PK*

with that interpretation for the while in Pascal. This bug can be plausibly explained with the *PL Interpreted as NL* Bug Generator. The bug generator is parameterized by the generalized Loop Plan for both PK and SSK. That is, surface links between the PK Loop Plan and the SSK Loop Plan allowed our subject to develop a patch to his understanding of the while loop.

### 2.2.2. Bug Generators: *Intra-PK*

The next class of bug generators allow novices to resolve an impasse within the PK. The idea of these bug generators is that novices construct a patch by generalizing or specializing based on known PK plans. Unfortunately, the patches so constructed often introduce bugs. Figure 2-5 shows a pictorial representation of the process, showing the various PK plans with links to other PK plans.

The specific bug generators are based on the kinds of inter-plan links possible in the PK. There is an over-generalization bug generator, an over-specialization bug generator, and a bug generator based on incorrect detail. Incorrect detail refers to a situation where the novice is using one PK plan and some aspect of that plan requires more detail, specified in another plan which the novice does not have. A typical example of such a detail plan would be information about appropriate actions within a loop, i.e. summing, counting, test for maximum, etc.

Consider some examples of *Intra-PK* bug generators. Several subjects overgeneralize in unnecessarily initializing to 0 a variable used in a Read statement. One subject asserted that all variables must always be initialized before they are used. A more subtle overgeneralization involves novices using a statement like New := New + 1 and describing this as reading the next value from a user. In this case the subject has reached an impasse about reading the next value. The patch is to overgeneralize from how a counter gets the next value.

### 2.2.3. Bug Generators: *Other Confounds PK*

The last class of bug generators operates much like the *SSK Confounds PK* Bug generators in that they use links to move between the PK and another domain (see figure Figure 2-6). In this case, though, the other domain is not SSK, but some other domain of common knowledge that has some parallels to PK. Two particular other domains that we've identified are Algebra and the programming environment. Algebraic variables, in particular, have parallels with programming variables (see [Soloway et al. 82b]). Novice programmers are also often less than experts in algebra. Bugs originating in the algebra domain often seem to translate into the programming domain. Other work on algebra bugs [Matz 82] [Kaput 79] and specific work with

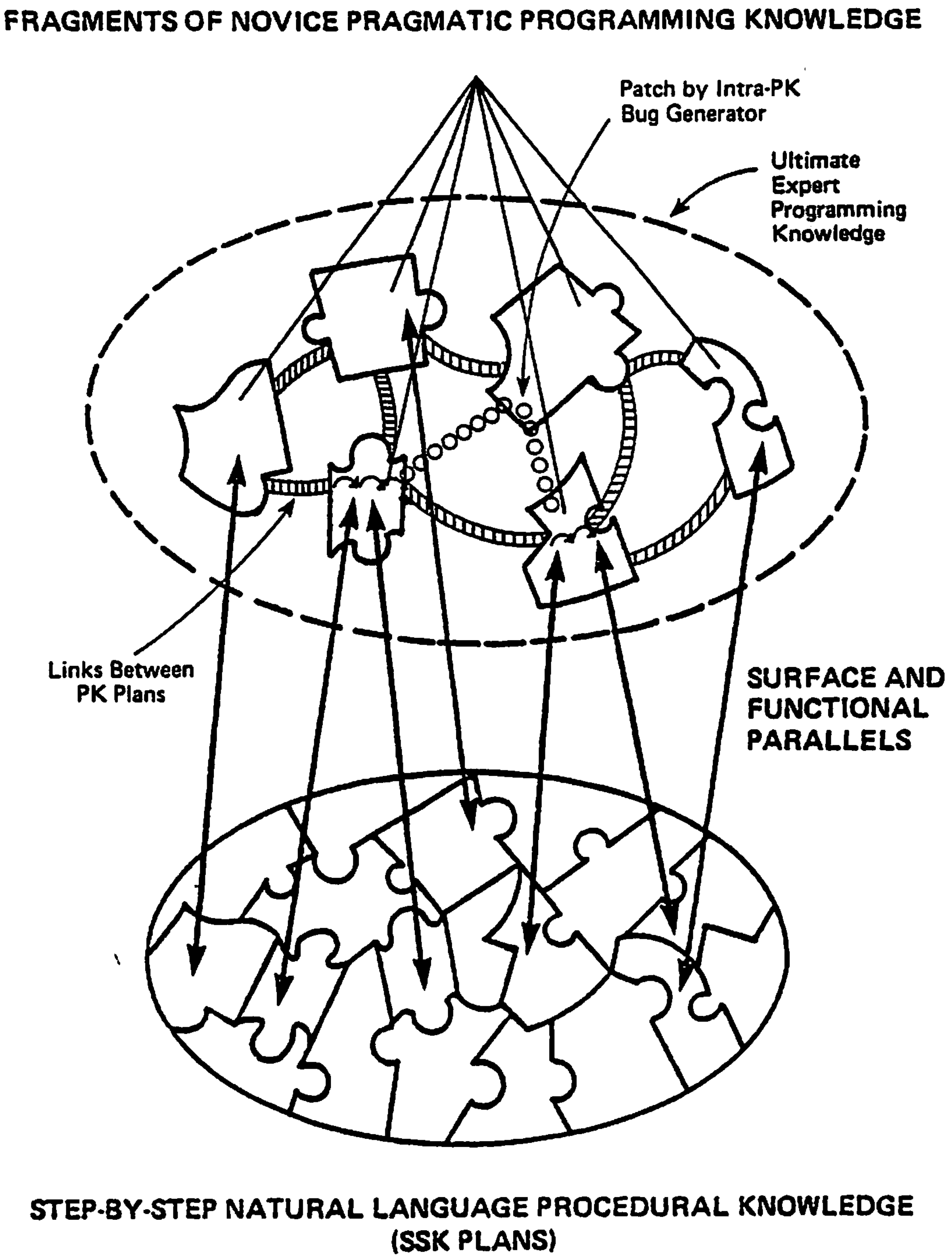
FRAGMENTS OF NOVICE PRAGMATIC PROGRAMMING KNOWLEDGE
Patch by Intra-PK Bug Generator
Ultimate Expert Programming Knowledge
Links Between PK Plans
SURFACE AND FUNCTIONAL PARALLELS
STEP-BY-STEP NATURAL LANGUAGE PROCEDURAL KNOWLEDGE (SSK PLANS)

**Figure 2-5:** Bug Generators: *Intra-PK*

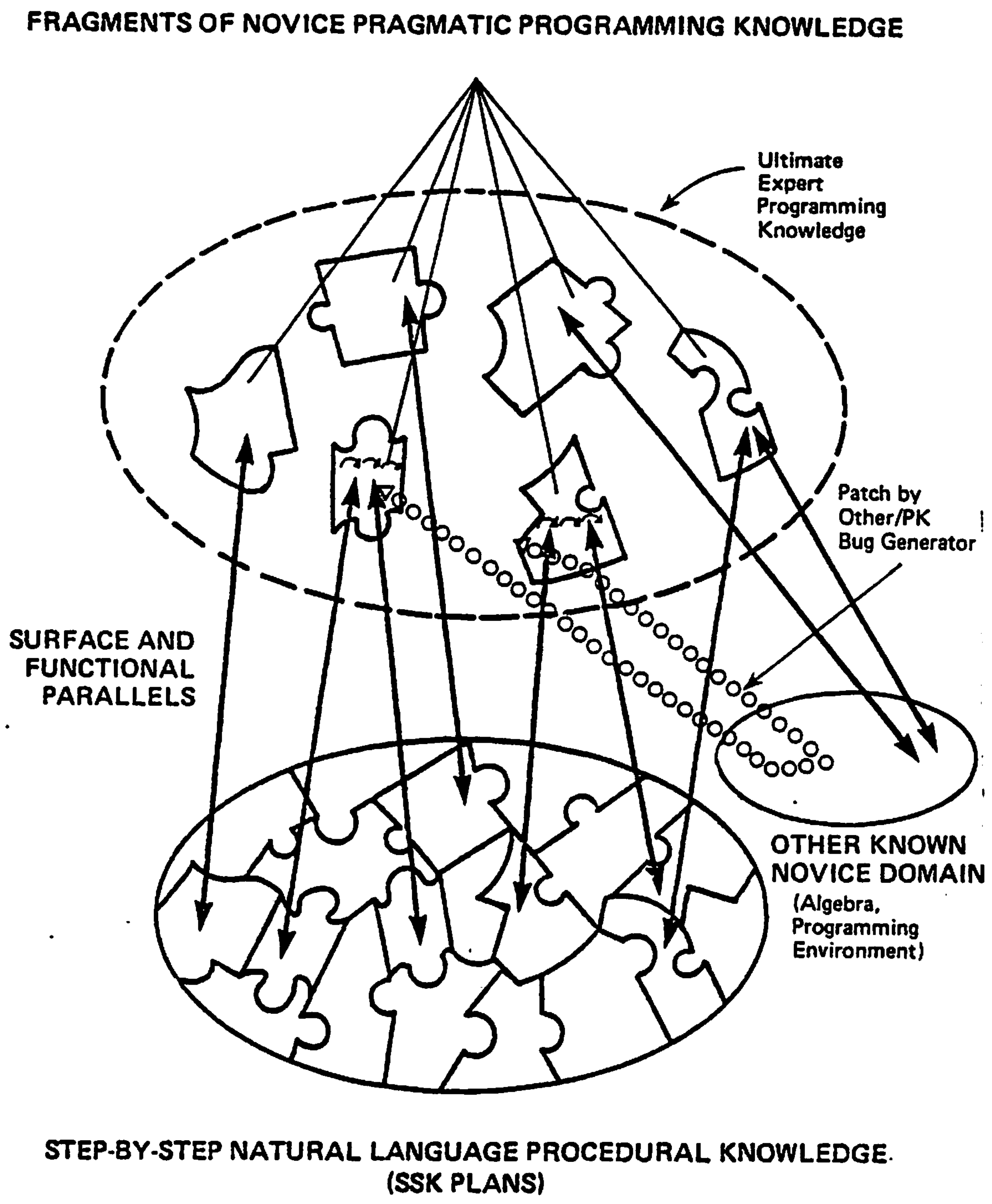
FRAGMENTS OF NOVICE PRAGMATIC PROGRAMMING KNOWLEDGE
Ultimate Expert Programming Knowledge
Patch by Other/PK Bug Generator
SURFACE AND FUNCTIONAL PARALLELS
OTHER KNOWN NOVICE DOMAIN (Algebra, Programming Environment)
STEP-BY-STEP NATURAL LANGUAGE PROCEDURAL KNOWLEDGE (SSK PLANS)

**Figure 2-6:** Bug Generators: *Other Confounds PK*

algebra variable bugs [Rosnick 82] has provided the specific bug generators in this class. In particular, there are many instances where novices collapse several variables into one identifier. For example, in the bug analysis shown in Figure 2-3, one plausible explanation is that Subject 13 collapsed the roles of *Input New Value Variable* and *Arithmetic Sum Variable*,[11] treating the Arithmetic Sum as if it will automatically get a new value added in whenever that new value is read.

The programming environment (command language, editor, etc.) also has parallels to programming that provides opportunities for bugs. A bug generator exists to account for bugs based on these parallels. For example, novice programmers may use an editor "read" command to perform a **Read** from Pascal.

## 3. Plans: Realizing PK and SSK

The theory of novice programming bugs includes two knowledge structures: novice pragmatic fragments of programming knowledge (PK) and step-by-step Natural Language procedure knowledge (SSK). In this section, we describe how those structures are realized for use in understanding protocols of novice programmers. The PK and SSK knowledge structures are represented as **plans**.

In this section we also describe details of the links between the PK and SSK plan sets. The functional links between these PK and SSK plans are represented by closely parallel plan sets in each domain. That is, for each PK plan, there is a parallel SSK plan that expresses the same action in step-by-step Natural Language procedures. Surface parallels, the lexical similarities between components of SSK and PK plans, are not represented directly in the plans. Instead, they are represented as part of the action associated with the Bug Generators that use surface parallels. Our specific realization of the knowledge structures and processes in the theory of novice programmer bugs is summarized in Table 3-1.

For the PK knowledge, this section describes what an **expert** would know. A novice knows only fragments of this knowledge, that is, only some of the plans. For the most part this expert knowledge is **tacit**. Experts are not aware of their use of this knowledge. We also describe what an expert would know about the SSK plans. Remember, though, almost everyone is an expert in SSK, since almost everyone uses Natural Language to produce step-by-step procedures.

[11]The *Input New Value Variable* receives a value from a **Read** statement. The *Arithmetic Sum Variable* holds a running total built up inside a loop. Both of these plans are discussed in Appendix I.

| Theory Component | Realization |
|---|---|
| **PK** | Includes: <br>• Strategic Programming Knowledge (not discussed in this article). <br>• Tactical Programming Plans. A set of tactical plans is summarized in Table 3-2. <br>• Pascal (and other programming language) Implementation Plans. <br>Contains what an expert would know. |
| **SSK** | Includes: <br>• Step-By-Step Natural Language Procedure Implementation Plans <br>Contains what an "expert" would know, but almost everyone is an expert. |
| **Functional Parallels** | Realized by each tactical plan having both a corresponding Pascal implementation plan and Step-By-Step Natural Language Procedure implementation plan. The parallels between the two implementation plans are implicit in their sharing the same tactical plan. Note: In general, PK contains more differentiated and precise plans that may not have functional parallels in SSK. For the PK plans discussed in this article, however, there exist corresponding SSK plans. |
| **Surface Parallels** | Realized by *SSK Confounds PK* Bug Generators which exploit lexical similarities between the two kinds of implementation plans to create a patch. |
| **Bug Generators** | Realized in the specific set of Bug Generators summarized in Table 2-1. |

**Table 3-1:** Realization of Components of the Theory of Novice Programming Bugs

## 3.1. Realising PK

In the theory of novice programming bugs, novice PK is described as fragments of an expert's programming knowledge. The knowledge is organized pragmatically, that is, according to what the statements and variables are used for and their role in the program. We describe pragmatics in a hierarchy of programming knowledge levels: strategic, tactical, and implementation. **Strategic knowledge** is used by expert programmers to manage the complexity of large programs. It will not be discussed in detail here.[12] **Tactical knowledge** is used to recognize and organize standard tasks in programming, but without implementation details specific to any given programming language. Tactical knowledge is information about the standard tasks of programming. These tasks include, for example, ***running total loops, counter loops, successive test and selection***, and ***filter out negative values***. Another way to see tactical knowledge is as the roles or purposes that can be accomplished by different segments of code and variables. Program comments often contain information about the tactics used in the program.

**Implementation knowledge** is used to actually implement a design and set of tactics in a specific programming language. This knowledge is language specific and can be viewed as extensions to the language independent tactical knowledge. Implementation knowledge tells us how to actually implement tactical knowledge. Implementation knowledge is what an experienced programmer needs to acquire when beginning to use a new programming language. For example, experienced Pascal programmers can grasp the ***concepts*** of programming in LISP, but may take a while to learn the ***standard techniques*** used in LISP programming. Those techniques are the implementation knowledge.[13]

As an example, consider the tactical programming plan for a counter. In Pascal a counter is usually implemented with code for a declaration, initialization, and increment (see Figure 3-1). In LISP, however, the counter is more likely to be counting recursive calls to a function as shown in

[12] Strategic knowledge is what makes a skilled programmer. Such knowledge is used in planning the overall strategies of the program, designing data representations, creating appropriate layers of functionality, abstracting and simplifying operations, and dividing a large programming project among a team of programmers. In the software engineering literature use of this kind of knowledge has been called "programming-in-the-large" [De Remer and Kron 76].

We do not discuss strategic knowledge any further. An understanding of strategic knowledge is probably at the heart of skilled programming and very difficult to characterize (but see [Jeffries et al. 81] and [Kant and Newell 83]). Our concern is primarily for the novice programmer, who has enough trouble with the simpler tactical and implementation aspects of a programming language.

[13] Note that for a novice there is probably little distinction between implementation and tactical knowledge. Only with experience does a programmer begin to differentiate these levels. We make the distinction here to provide an expert baseline by which we can examine the extent of novice knowledge.

Figure 3-2. And finally, there are step-by-step Natural Language procedural implementations of the counter (see examples in Figure 3-3).

PK is represented with both tactical plans and Pascal implementation plans. This means that an expert programmer knows both tactical plans describing general programming techniques and implementation plans describing how those techniques are implemented in a specific programming language.[14] Novices have fragments of this complete tactical and implementation plan set. In Table 3-2 we summarize the plans we have developed for use in the Ending Value Averaging Problem (see Figure 3-5 on page 33).

## 3.2. Realising SSK

In the preceeding section we discussed a realization of PK plans for capturing programming knowledge. Here we describe a parallel realization of SSK plans. SSK is represented as implementation plans for Natural Language step-by-step procedures. These SSK plans can be thought of as a representation for Natural Language programming. The idea of Natural Language programming has provoked much discussion. Some have argued that Natural Language is too informal to be useful as a programming tool [Dijkstra 78], while others have suggested that Natural Language would represent the ultimate user interface [Codd 74]. Miller [Miller 81] (also see Miller [Miller 74]) reports on a series of experiments characterizing step-by-step Natural Language procedures. Biermann et al. [Biermann 83] describe work identifying patterns of usage in step-by-step Natural Language. They have implemented a "Natural Language Programming" system called NLC. In an experiment with that system, 74% of the subjects were able to learn the system and produce correct solutions to simple data manipulation problems (solution of a system of 3 linear equations) in less than 2.5 hours.

The SSK presented in our theory of novice programming bugs is a different view of natural language programming then that which motivated the discussion and and studies mentioned above. Where other work has focused on the possibility of automated understanding of Natural Language, our SSK characterizes the kind of Natural Language programming done by humans for other humans. In this context, it is reasonable to represent SSK as an implementation of tactical plans. The theory does not insist on automatic understanding of these Natural Language

[14]Shneiderman and Mayer [Shneiderman and Mayer 79] develop a similar model where they describe a "semantic" programming knowledge base to hold general techniques and a "syntactic" programming knowledge base to hold language specific information.

```
integer Count;          {Declare the variable}
. . .   .
Count := 0;             {Before the loop doing the counting}
. . .
Count := Count + 1;     {Inside a loop doing counting}
```

**Figure 3-1:** Pascal implementation of the tactical programming plan for a counter

```
(defun foo-with-count (x y count)
   . . .
   (foo-with-count . . .  (plus count 1))
   . . . )
```

**Figure 3-2:** LISP implementation of the tactical programming plan for a counter

*"the number of . . . "*

*"frequency of . . . "*

*"get a count of . . . "*

*"count up"*

**Figure 3-3:** Natural Language implementations of the tactical programming plan for a Counter

## Summary of Plans

| | |
|---|---|
| **Sentinel Variable** | Holds an end of data value. |
| **Counter Variable** | Counts activities in a program, particularly in a loop. |
| **Arithmetic Sum Variable** | Holds a running total, especially in a loop. |
| **Result Variable** | Holds a value whose determination satisfied a local or global goal of the program. |
| **Input New Value Variable** | Holds a new value, recently input from the user of the program. |
| **Loop** | The generic looping plan. |
| **New Value Controlled Loop** | A loop whose stopping condition is based on a New Value Variable. |
| **Illegal Filter** | Protects some computation from certain "illegal" values in variables used by the computation. |
| **Input** | Gets a value from the user. |
| **Result Output** | Reports a result to the user. |
| **Instructional Output** | Gives the user instructions. |
| **Prompt Output** | Asks the user to enter some input value. |

**Table 3-2:** Summary of Plans

implementation plans, only on characterizing their relationship to the more general tactical and Pascal implementation plans.

The step-by-step Natural Language procedure plans were developed from a study of non-programmers. Twenty-four subjects were given four problems requiring the development of a step-by-step Natural Language procedure. Consider, for example, the Payroll Problem (shown in Figure 3-4). This problem is analogous to the Ending Value Averaging Problem (shown in Figure 3-5) in that it requires the same tactical plans for a solution. Each problem in the Natural Language procedure study was designed to be analogous to a standard introductory programming problem. This allowed use to identify common tactical plans and contrast the implementation plans.

We coded each step-by-step Natural Language procedure by noting noting how each of the tactical plans underlying the problem were actually realized in the step-by-step procedures. For example, the Payroll Problem (Figure 3-4) uses the *Counter Variable* and the *New Value Controlled Loop* plans (among others). In Figure 3-6 we show an example solution, written by Subject 11 before he began an introductory programming course. There, the *Counter Variable* plan is implemented with the phrase "Add the number of workers you've written down." (step 5). The *New Value Controlled Loop* plan is encoded more subtlely. Steps 1, 2, and 3 all form the body of the loop, describing approximately the first two iterations of the loop. Repetition of this body is indicated with the phrase "... and so on" at the end of step 3. The stopping condition is described in step 4. We used the results of this study of step-by-step Natural Language procedures to compile step-by-step Natural Language implementation plans for each tactical plan previously compiled.

## 3.3. Representing Functional Links Between PK and SSK

For each of the key tactical plans (part of the PK) described in Table 3-2, there is a Pascal implementation plan (also part of the PK) and a corresponding Natural Language implementation plan (part of the SSK). For example, in Figures 3-1 and 3-3 we see the corresponding PK and SSK implementations for a Counter tactical plan. The functional links between the PK and SSK are represented through these corresponding plans. If the full knowledge of an expert programmer is considered, there are many tactical plans with no corresponding Natural Language implementation plans. Evidence from our study of step-by-step Natural Language procedures, however, indicates that every one of the key tactical plans presented below does have a Natural Language implementation plan.

Please write a set of explicit instructions to help a junior clerk collect payroll information for a factory. At the end of the next payday, the clerk will be sitting in front of the factory doors and has permission to look at employee pay checks. The clerk is to produce the average salary for the workers who come out of the door. This average should include only those workers who come out before the first supervisor comes out, and should not include the supervisor's salary.

**Figure 3-4:** The Payroll Problem

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

**Figure 3-5:** The Ending Value Averaging Problem

1. Identify worker, check name on list, check wages
2. Write it down
3. Wait for next worker, identify next, check name, and so on
4. When super comes out, stop
5. Add number of workers you've written down
6. Add all the wages
7. Divide the wages by the number of workers

(from Subject 11)

**Figure 3-6:** Natural Language Implementation of New Value Controlled Loop

In Figure 3-7 we show *Counter Variable* plans as an example of tactical and corresponding Pascal and Natural Language step-by-step procedural implementation plans. The Counter Variable is a kind of Arithmetic Sum (running total) Variable that always adds in 1. The tactical plan consists largely of a description of the plan's functionality. For example, how the counter is initialized, how it is used, and the type of its values. The implementation plans are made up of "criteria" - things that a programmer would say or write in their program that indicate they have used the plan. These criteria are used in novice programmer protocol analysis, and described in detail below. The criteria illustrate how different the two implementations are. In particular, notice that the Natural Language implementation refers to global counting operations over a whole set of items while the Pascal implementation refers to individual counting operations for each item. Also note that in the Natural Language implementation the count operation is specifically after the operation being counted.

## 4. Evaluating The Theory

In Section 2 we developed a theory describing the source of novice errors. Although detailed testing of that theory is beyond the scope of this article, this section presents data to begin evaluation of the theory. The data is derived from a detailed analysis on four protocols selected from a body of interviews conducted to formulate the theory. In the selected protocols we present the same introductory programming problem to four different novice programmers, all in the fourth to sixth week of an introductory programming course. The analysis of those four protocols is described in this section. We begin that analysis by identifying the underlying plan knowledge used in each interview. This plan knowledge is then related to specific bug generators, providing plausible explanations for the errors found. We conclude by presenting summary data about plan and bug generator usage from all four protocols and showing how it supports the theory.

The protocols of novice programmers were originally collected to develop intuitions for the theory of novice programming bugs. As that theory developed, it became clear that these protocols, originally used for **theory formation**, could also be used to provide data for an **initial evaluation**. For example, since the theory predicted the key role of programming and Natural Language plans, an analysis of the protocols should show a pervasive use of plans, as well as an indication of the which plans were most important. Similarly, a detailed analysis should use the bug generators to plausibly explain each bug in the protocol, and indicate the overall utility of each bug generator. In this section, we present an analysis of the protocols used to support and begin evaluation of the theory of novice programming bugs.

## Counter Variable Plan

PARENT ⇒ Arithmetic Sum Variable Plan

- □ *Description:* Counts the occurances of some specific action.
- □ *Initialization:* set to zero
- □ *How used?:* Builds up a value by incrementing into itself at at each step.
- □ *Type:* integer

IMPLEMENT ⇒ Pascal - Counter Variable Plan
IMPLEMENT ⇒ NL - Counter Variable Plan

## NL - Counter Variable Plan

IMPLEMENTS ⇒ Counter Variable Plan

*Criteria:*

1) "the number of . . . "
2) "the total number of . . . "
3) "keep count of . . . "
4) "Count the number of . . . "
5) "Add the number of . . . "
6) "Frequency of . . . "
7) Specifically requires counting to be done after the operation being counted
8) "get a count of"
9) "take a count"
10) "counting the . . . "
11) "count up"

## Pascal - Counter Variable Plan

IMPLEMENTS ⇒ Counter Variable Plan

*Criteria:*

1) Needs two statements, and initialize and an increment. If CV is the Counter Variable, then the form for these is:
```
   CV := 0          (initialize)
   CV := CV + 1     (increment)
```
2) The initialize is above a loop
3) The increment is inside that loop
4) "count each"
5) "increment one-by-one"

**Figure 3-7:** Counter Variable Plans: tactical, Natural Language, and Pascal

The analysis presented in this section is not sufficient to confirm the theory of novice programming bugs. Such a confirmation would involve a study with many subjects, many interviews, and detailed statistical analysis. Given current understanding of the programming process and the level of detail of our model, such a study seemed premature. Instead, this analysis is intended to add a level of rigor and detail to the knowledge structures and processes discussed above. Our analysis is also intended to provide techniques to summarize and compare protocols of novice programmers. In this chapter we examine the following questions: What kinds of plans were used, and how often? How are these plans used when bugs occur? What kinds of bug generators provide plausible explanations for the bugs? How often did each kind of bug generator provide a plausible explanation?

This section begins by presenting three expectations derived from the theory of novice programming bugs. These expectations are used to organize the analysis presented. After briefly looking at methodological questions arising from our use of protocols, we present details of the methods used to perform an analysis of protocols of novice programmers. We conclude this section with the results of that analysis and an evaluation of the expectations developed at the beginning of the chapter.

## 4.1. Expectations

To begin evaluation of the theory of novice programmer bugs we develop a series of **expectations** deriving from the theory. The expectations are not quantitative predictions derived from the theory of novice programmer bugs. As discussed above, current knowledge about novice programming behavior does not justify a quantitative and statistically rigorous analysis. Instead, the expectations represent important trends in the theory. As trends, the expectations are supported by the analysis presented here. This support should not be taken as a confirmation of the theory. Instead, the support provides evidence that the overall approach is correct and further work in focusing and developing the theory is justified.

The first expectation concerns the importance of plans. The theory of novice programming bugs uses plans to represent the programming knowledge used by novices. If plans actually do account for most of the programming knowledge, then plan usage in the protocols should be extensive. This is the first expectation:

> **Expectation 1:** Novices show a regular and extensive use of plans. Novices should make use of *both* SSK and PK implementation plans.

This expectation says that plans are used extensively by novice programmers. It specifies that novices use both SSK and PK implementation plans.[15] Evidence for implementation plans needs to indicate that the novice was reasoning about implementing a programming step in either step-by-step natural language or in Pascal.

Expectation 1 can fail because novices may rely on knowledge not contained in the programming plans we describe. An obvious non-plan source of programming knowledge available to novice programmers are the descriptions of programming constructs presented by programming textbooks and language manuals. These are usually relatively formal descriptions, closely related to the defining semantics of the programming language. Another possible non-plan strategy for novice programmers is reasoning based strictly on syntactic matching from problem statement to available programming constructs. Where a plan formulation of the program requires an understanding of the intended results, this syntactic strategy would consist of simply matching key phrases in the problem statement producing remembered lines of code. Once a novice had a number of lines obtained in that way, he or she would then work on ordering the lines of code. (See Larkin [Larkin et al. 80] for a discussion of such a syntactic strategy used by novice physics students.)

Expectation 1 also can fail because novices may not use *both* SSK and PK implementation plans. We might, for example, find a novice that reasoned with SSK plans, but used another knowledge base (e.g. the semi-formal textbook descriptions or syntactic matching discussed in the last paragraph) to write actual Pascal code. On the other, we might find a novice that used PK plans extensively but no SSK plans at all.

The theory does not specify a particular set of bug generators, only general categories of bug generators. We developed a specific set of bug generators to account for the bugs in protocols 1 and 2 discussed below. Since those were our most errorful protocols, the specific set of bug generators developed should adequately explain most errors seen in novice programs. This is the next expectation:

> **Expectation 2:** The Bug Generator set presented here can plausibly explain most errors found in the protocols.

This expectation is tested in the analysis of protocols 3 and 4. The specific bug generators

[15] We are unlikely to see protocol evidence for novice use of tactical plans. Such evidence would need to indicate that the novice was reasoning about the programming steps in an abstract, language independent way.

presented may be inadequate to explain the bugs found. In particular, since protocols 3 and 4 are less buggy then protocols 1 and 2, it is possible that the less-novice subjects producing protocols 3 and 4 make different kinds of errors.

The first two expectations are basic to the theory of novice programmer bugs. They do not, however, represent the key contribution of the theory. The theory proposes that the intrusion of SSK plays an important role in novice programming bugs. From this, we get the third expectation:

**Expectation 3:** When novices encounter an impasse in a developing programming solution, they usually use *SSK Confounds PK* Bug Generators to patch and continue.

In less formal terms, this expectation says that intrusion of SSK explains most novice programming bugs. Most non-programmers are quite familiar with SSK. It is reasonable that their knowledge of SSK will intrude when they begin to write programs. This is the key intuition that led to this work. Evaluation of this expectation lies at the core of the work reported here.

This expectation can fail because the bugs may not be particularly associated with SSK or *SSK Confounds PK* Bug Generators. For example, we might find that most bugs involve direct misuse of programming constructs, e.g. assuming a **while** loop, like a **repeat** loop, tests at the bottom of the loop - an *Intra-PK* Bug Generator.

## 4.2. Using Protocol Analysis

In our interviews we presented an introductory programming student, the subject, with a problem. The subject worked the problem under our supervision while being video-taped. The interviews were later analyzed, looking for instances indicating use of either SSK or PK implementation plans. Using this information, the protocols and the bugs in those protocols were interpreted in terms of bug generators.

Primarily, a **thinking-aloud** interview format was used for these interviews. Subjects were given problems and instructed to verbalize their thoughts while working on the problems. In addition to thinking-aloud about the program development process subjects were asked to think-aloud while hand simulating their program. At no time were subjects given hints relevant to the solution of the problem, or any evaluation of what they had written. See Ericsson and Simon [Ericsson and Simon 80] for a discussion of methodological issues in using verbal reports as data. The use of protocols to study complex behaviors is addressed in depth with the detailed model of human problem solving developed by Newell and Simon [Newell 77, Newell and Simon 72]. Other protocol work has been done studying complex problem solving behaviors using other models of human cognition [Anderson 83, Clement 81, Konold and Well 81].

In refining our theory of novice programmer bugs, we have chosen protocol analysis for two basic reasons. Firstly, we want to build something more than a descriptive model of novice programmer behavior. The interviews give us a needed detailed view of the novice programmer process. By watching novice programming behavior in a naturalistic setting (i.e., a novice solving a complete programming problem during an interview session), we are better able to understand and capture the novice knowledge and processes by which the knowledge is used.

The second reason for using protocols is that they provide the right level of overview: they do not focus on particular programming components, tasks, or phases. Protocols allow us to study both the components of programming behavior and the relationships between those components. For example, in a protocol we can study how looping strategies influence variable usage. We can do this as those topics are naturally brought up by the subject as part of a full problem solving session. Even more telling, we are able to see, study, and interrelate unexpected phenomena, like those cases where variable usage influences looping strategies. Not only can the individual components be studied, but how the novice connects those components to the rest of the program is indicated by the context in which the discussion of the component arises.

Protocols also provide an overview to the different cognitive tasks that make up programming. These tasks include understanding the problem statement, design, coding, and evaluation (hand-execution) of the code developed. The broader view provided by an interview is necessary to build unifying models of the programming task.

## 4.3. Protocol Analysis Methods

Our analysis of the protocols has two steps: the **plan analysis** and the **bug analysis**. The plan analysis describes the plans that seem to be in use at different points in the transcript. For each segment of the transcript, the plan analysis presents the relevant plans and evidence for those plans. The plan analysis is an attempt to characterize the knowledge in use as the subject solves the problem. The bug analysis presents the subject's bugs and proposes bug generators prepresenting plausible explanations for those bugs. The bug generators act as general, and usually errorful, repair strategies parameterized by the plans the subject has recently used. For each bug, the bug analysis presents relevant bug generators and plans.[16] We begin by describing how protocols were selected for detailed analysis, and then the details of those analyzes.

### 4.3.1. Selecting Protocols for Analysis

In the course of the work reported here we conducted twenty-two interviews with thirteen different subjects. That larger body of protocols was use to develop the theory of novice programmer bugs. In developing the analysis technique presented here, ten of those protocols were analyzed in depth. Of those ten, four with the same programming problems were selected for the analysis presented in this chapter. These four protocols were all taken from college students in the fourth to sixth week of an introductory Pascal programming course. In each protocol, the subjects worked on the Ending Value Averaging Problem (see Figure 4-1).

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

**Figure 4-1:** The Ending Value Averaging Problem

In this problem, the program is to find the average of a series of input numbers terminated by a distinguished value. In Soloway et al. [Soloway et al. 82a], this was found to be the most difficult of three such simple looping and averaging problems. Later work [Soloway et al. 83] examined student performance on this problem in relation to a peculiar construction required by the Pascal **while** loop. (The **while** is the most appropriate looping construct for this problem.) This extensive work on the ending value averaging problem made it a logical problem for detailed protocol analysis.

### 4.3.2. Plan Analysis

The plan analysis attempts to account for the subjects utterances and program code in terms of the plans. In the theory of novice programming bugs, programming is seen as a process of moving from plan to plan, using the knowledge in those plans to write program code or reason about code already written. For a novice, many of the plans used in this process may be SSK plans and not PK plans. The plan analysis attempts to describe which plans are in use by the

[16]In the analyzes presented here, the Plan Analysis and Bug Analysis are folded into the Transcript. In this way, it is easy to see the relations between plans, bugs, and the actual transcript.

subject at any given time.[17] Also, the plan analysis presents evidence that the subject is actually using the plan as claimed.

The key issue in the plan analysis is to relate the relatively abstract implementation plan descriptions (as described in Section 3) to the actual utterances and programs produced by the novice. This relationship is established by the **criteria** fields of the implementation plans. Consider an example to illustrate the use of criteria and the relationship between implementation plans and the protocol. A detailed description of the kinds of criteria and their use is provided after the example.

Figure 3-7 (on page 35) shows the Pascal and Natural Language implementation plans for a Counter Variable. (The reader also may want to refer to the plan summary in Table 3-2 on page 31, and bug generator summary in Table 2-1 on page 21.) The criteria fields correspond to the ways in which each implementation plan can be recognized in a novice protocol. Consider the protocol segment and associated plan analysis annotation in Figure 4-2. In the plan analysis annotation we claim that the subject used a Counter Plan in segment 112. It offers two pieces of evidence for that claim: one evidence item (N4) says that there is evidence for the Natural Language Counter Variable implementation plan, based on criterion 4 (the text after the "N4" repeats the criterion for ease of reading). The second evidence item (P5) says that there is evidence for the Pascal Counter Variable implementation plan, based on criterion 5 (again, the text for "P5" is repeated). There are three things to note about the example:

1. The example provides more than one evidence item for the plan.
2. A novice often says things that indicate they are using both kinds of implementation plans simultaneously.
3. The phrase "incrementing by ones" does not exactly match the criterion target phrase "increment one-by-one", but is considered a match because the criteria for these target phrases are not applied literally.

These features are covered in the following discussion.

The plan analysis is presented as annotations to the transcript. Each plan annotation

[17]The plan analysis is focused on implementation plans and not on tactical plans. Implementation plans contain the knowledge needed to actually implement a plan in a language like Pascal or step-by-step Natural Language. Tactical plans contain more general programming knowledge. The plan analysis focuses on implementation plans because they contain information that appears directly in the novice's code and utterances. Tactical plans cannot be tied to the novice's work in this direct way, and therefore are not discussed as part of the plan analysis.

112 **Subject 11:** . . . I want it simply incrementing by ones, it's counting the number of integers that come through,

**PLAN: Counter**
Evidence: N4 - "Counting the number of integers ..."
Evidence: P5 - "incrementing by ones"

**Figure 4-2:** Protocol Segment With Claimed Instance of Counter Variable Plan Use

mentions specific plans and cites evidence for claiming that those plans are in use. We call each item of evidence for a given plan an **evidence item**. In the quantitative summaries presented later in the chapter, we use counts of evidence items to represent the overall plan activity. This counting scheme means that each evidence item is treated as if it were a unique instance of the plan usage. Although it is acceptable to think that a subject might have several evidence items for a single plan instance, there is no way to make a finer distinction in the protocols. The counting scheme used, though subject to this possible overcounting, is still a reasonable measure of plan activity.

In many places in the transcript we show plan annotations with several evidence items. These are simply a convenient way to present nearby evidence items. For example, the example plan annotation of Figure 4-2 would result in counting two evidence items: as an instance of SSK plan activity and an instance of PK plan activity.

An evidence item for a specific plan is represented by a pair of characters like "N4" and "P5" in Figure 4-2. The first character of the pair says whether that evidence item indicates the use of a Natural Language (SSK) implementation plan (denoted by "N") or a Pascal (PK) implementation plan (denoted by "P"). As is shown in the example, in a brief utterance the subject may provide several pieces of evidence for a given plan. It is not unusual for multiple pieces of evidence to include evidence for both kinds of implementation, i.e. evidence that the subject is reasoning with both kinds of plan implementation knowledge.

The second character in the evidence item pair specifies the specific criteria used to assert that the segment is coded by the specified plan. For example, the entry **P5** in the plan of Figure 4-2 asserts that a Pascal implementation of the Counter Variable Plan is in use and offers criterion 5 as evidence. There are five different kinds of criteria, indicated by the second character of the evidence pair. These criteria are:

1. **N** - The subject used a **N**ame that clearly suggests a certain plan. For example, `Sum` for the Arithmetic Sum, or `Counter` for the Counter.
2. **S** - The subject **S**aid so directly. For example, if the subject said "here I'm using a counter to keep track of the number of values entered".
3. **C** - The subject used a Pascal **C**onstruct that directly suggests a certain Pascal implementation plan. For example, `repeat` and `while` clearly suggest that the subject intends to do looping.
4. **E** - The subject used an **E**xample of some other specific Pascal code. In this case the subject might say "this is going to work just like the statement I wrote here".
5. Function (indicated by a number) - The subject used a phrase or fragments of code that accomplishes the function of the plan. Since most plans have several functional criteria, a specific criterion is indicated in the plan analysis by a number indicating which functional criterion is intended. Note that many functional criteria take the form of **target phrases** - specific phrases to be matched with the wordings in the transcript. "N4" and "P5" from the example in figure 4-2 are both functional criteria.

As discussed in Section 3, the functional criteria for Natural Language step-by-step procedural plans come from a written study of non-programmers writing Natural Language step-by-step procedures. We used representative target phrases from this study to make up the criteria for the Natural Language implementation plans. The functional criteria for Pascal implementation plans represent specific Pascal code, usages, and descriptions associated with that plan. We developed these Pascal implementation criteria by introspecting on our own Pascal programming and teaching experience.

The criteria are not applied strictly. To do so would vastly expand the number of functional criteria. With the Natural Language plans in particular, the target phrase lists would be very long if they did not allow for small variations in lexicon or syntax. For example, when we list the phrase "add up total of . . . ", we also consider "add up all . . . " and "sum up all . . . " to be equivalent. In general, if there was any serious question about whether some item qualified as matching a criterion, it was separated and made into a new, separate criterion.

### 4.3.3. Bug Analysis

The Bug Analysis shows how errors made by a novice can be plausibly explained as Bug Generators operating on currently active plans. Bugs are any error the subject makes that is not immediately corrected. With this definition we exclude immediately corrected slips from our analysis. Within the analysis, however, we still may treat the error as an uncorrected slip. (In the protocols presented below, a number of bugs are plausibly explained as slips.) Typically, written studies of novice programmers can count only those bugs that are left in finished code. The protocol, however, allows us to examine and explain even those bugs that a novice later corrects or mutates. We look for bugs in the programming code the subject writes and in the explanations given by the subject.

The Bug Analysis, like the Plan Analysis, is presented as a series of annotations to the transcript. We analyze each piece of buggy behavior in a separate bug annotation. Where there are plausible connections between nearby bugs, that is noted within the separate annotations. Figure 4-3 shows two example Bug Analysis annotations. The bug annotations have several sections. First the bug is identified (the first line of the bug annotation, in boldface) with a descriptive title.[18] The titles are designed to capture the spirit of the bug. The title is often a quote from the subject, as illustrated in the first bug shown in Figure 4-3. Following the title (the text after the title but before any right pointing hands) is a more detailed description of the bug, often contrasting the subjects actions with a correct answer.

The next section of the bug annotation contains one or more of the bug generators that provide plausible explanations for the bug (each bug generator and explanation is indicated with a right pointing hand). The bug generators (in italics text) are parameterized by plans (in roman text). Bug generators are accompanied by a textual description to explain the details of the bug generator/plan interaction and to provide evidence for that particular explanation of the bug.

For the purpose of the bug analysis we introduce PK plans describing the usage of each Pascal programming construct. In Figure 4-3 the second and fifth right pointing hands illustrate usage of these special PK construct plans. These Pascal construct plans are part of what an expert knows, not necessarily part of what a novice knows. As used in the bug analysis, the

[18]The titles are intended strictly descriptively. We do not have a systematic catalog of bugs (and bug names), since such a catalog would necessarily be based strictly on the features of the buggy program. Instead, we understand bugs as bug generators operating on plans. This allows us to categorize bugs based on the knowledge and processes used by the novice to arrive at a given bug.

---

**BUG: "It reads the Sum": Sum := 0 + I**

The subject is incorrectly tying the read and sum operations together. Something like "it reads in a value which is then added into the sum" would be correct.

☞ ***PL Used as NL*** **on Input New Value - I**

The read operation is done implicitly whenever a value is needed.

☞ ***PL Interpreted as NL*** **on Pascal - Read/Readln**

Here he is treating Read; Readln pair as if they were declaring that reading is to be done, and the results of the read will be used with Sum.

☞ ***Multi-Role Variable*** **on Arithmetic Sum - Sum, Input New Value - I**

The subject is using the variable Sum as if it will automatically get a new value added in whenever that new value is read.

☞ ***Slip***

Just slipped and forgot to say "... a value which is added ... ".

---

**BUG: Counter continues to add when set to 1, then 2**

Subject says that the Counter Variable when set to 1, and then set to 2 inside the loop body, will continue to increment on each iteration.

☞ ***PL Interpreted as NL*** **on Pascal - repeat, Indefinite Loop**

The subject is expecting the loop to work like loops in Natural Language. There, it is common to specify a loop by giving one or two cases of the iteration and assuming that person reading will know how to generalize. Notice he says that the Sum := 0 + I is the "first format of that" refering to the action performed for each value of the New Value Variable.

☞ ***Trace*** **on Counter**

The subject is actually stating the series of values that N will take on through the execution of the program.

---

**Figure 4-3:** Examples of Bug Analysis Annotations

---

novice *does not* know this plan. Instead, the bug generator is operating to make up for missing knowledge. The Pascal construct plans are a special case because it is quite typical for a novice to use Pascal constructs without any pragmatic understanding of the construct.

To illustrate these features of the Bug Analysis annotations, we discuss the first annotation shown in Figure 4-3 in detail. (Note here that we are extending the description provided for Figure 2-3.) The Bug discussed is based on the subject describing the following line of code: `Sum := 0 + I` `Sum` is the Arithmetic Sum Variable and `I` is the Input New Value Variable. In the subject's discussion of `Sum := 0 + I` he says that "it reads the sum". This quote is used in titling the bug. In the description of the bug we discuss how the subject seems to have incorrectly tied the reading and summation operations together. Also in the description we propose a similar phrase that, had the subject used it, would have been correct.

The first plausible bug generator is *PL Used as NL* applied to the Input New Value Variable Plan (the subject's variable `I`). The second plausible bug generator is *PL Interpreted as NL* applied to a special Pascal construct plan that summarizes the usage of the Pascal `Read` and `Readln`. This bug explanation is proposing the bug generator as substituting for the expert knowledge about the Pascal constructs. The third plausible bug generator, *Multi-Role Variable* operates on the Arithmetic Sum Variable Plan and the Input New Value Variable Plan. Finally, it is plausible that the subject just slipped here, and actually did understand that the running sum operation and inputing a new value were separate operations. There is a final thing to note about this example bug annotation. As experts we look at the statement `Sum := 0 + I` and know there is likely a problem with the `0` on the right side of the assignment. In the bug analysis this is handled in a separate bug annotation. This concludes our discussion of the example bug analysis annotation in Figure 4-3.

All plausible bug generators are included in the annotation associated with each bug. Usually there is more than one plausible bug generator. Our preference is indicated by order: the earliest mentioned bug generator is the one we consider most likely. When there is specific data for these preferences, it is indicated in the textual description. Note that these explanations are often *not* mutually exclusive. Even though we treat the explanations as if they don't interact, the best explanation may be a combination of several plausible bug generators. In general, determining the interaction between plausible explanations and choosing the best explanation is still an open problem in this research.

## 4.4. Results of the Analysis

The protocols of four subjects working on the Ending Value Averaging problem were analyzed with the methods discussed above. (This problem appears with a correct solution in Figure 4-1 on page 40.) We begin by discussing the four protocols, focusing on the final program produced by each of the subjects and a summary of the bugs made during the protocol. Next, the expectations discussed in Section 4.1 are evaluated, based on summary data from the protocols.

### 4.4.1. Overview of the Protocols

In this section we detail the performance of each subject on the Ending Value Averaging Problem.[19]

**Interview 1:**

Subject 13's final program is shown in Figure 4-4. There are a number of things wrong with this program, but most critical is his peculiar loop body. Notice that within the loop body, Sum (the Arithmetic Sum Variable) is first set to 0 (see 1), and then to I + Next I (I is the Input New Value Variable) (see 2), while N (the Counter Variable) is set to 1 (see 3) and then 2 (see 4). The subject seems to be implementing the loop with the following Natural Language strategy: *Show an example of the first few steps, and assume that the other iterations will happen correctly* (he actually shows the first two steps). In Appendix III we show an analyzed excerpt from Protocol 1.

**Interview 2:**

Subject 6's The final program is shown in Figure 4-5. It is almost correct. His problem is that there is no READ inside the loop. In the protocol he convinces himself that he needs a READ above the loop (see 1) to make the WHILE test make sense (see 2), but never thinks to also put a second READ inside the loop. At one point, he uneasily states that each test of NEWNUM in the WHILE statement (see 2) will know to read a new value for that iteration.

**Interview 3:**

[19]**Note On Presentation:** Throughout this section, code produced by subjects is reproduced here in as accurately as possible. For example, some subjects used upper-case only in their programs, while others use mixed case. The subject's case preference is preserved here.

Small superscript numbers, e.g. $^1$, $^2$, etc., are used to refer to lines of code discussed in the text.

```
Program Average (Input/. Output);
Var
   N, Sum : Integers
   Average : Real;
Const : Sentinel
Begin (* Average of the integers entered *)
   Writeln ('Enter series of integers to be averaged');
   Writeln (Integers will be Averaged when you
                                         enter the Integer 99999);
   Read;
   Readln;
   Sentinel := 99999;
   N := 0;
   Sum := 0;
Repeat
   Read;
   Readln;
   Sum := 0[1]
   N := 1[3]
   Sum := I + Next I[2]
   N := 2[4]
   until I = 99999
   then Average = Sum/N
Writeln ('Average':= 0);
END.
```

(superscripts mark lines discussed in the text)

**Figure 4-4:** Final program for Subject 13 working on the while problem.

```
PROGRAM AVERAGE (INPUT/. OUTPUT);
COUNT
   SENT = 99999;
VAR
   NEWNUM, COUNT, SUM, AVE : INTEGER;
BEGIN
   COUNT := 0;
   SUM := 0;
   READLN;
   READ (NEWNUM);[1]
   WHILE NEWNUM <> SENT[2]
     DO BEGIN
        SUM := NEWNUM + SUM;
        COUNT := COUNT + 1;
        END;
   IF COUNT <> 0
     THEN AVE := SUM DIV COUNT
   WRITELN ('AVERAGE = '), AVE:8:2;
END.
```

(superscripts mark lines discussed in the text)

**Figure 4-5:** Final program for Subject 6 working on the while problem.

Subject 11's final program is shown in Figure 4-6. Her program is almost completely correct. Like Subject 6, her bugs are related to reading new values inside the loop. She recognized that some sort of Read was required inside the loop, and even realized that it had to come after the Count increment (see 1) and Total update (see 2). The Read (I) below the loop (see 3) was originally, and correctly, put at the bottom of the loop body. In the protocol she argues that leaving it there would cause the program to read a new value before the previous value read is processed. She uneasily moved Read (I) out to it's current position outside the loop (see 3) and settled on Readln (see 4) at the bottom of the loop to express the right amount of reading.

### Interview 4:

Subject 12's final program is shown in Figure 4-7. This program is almost correct. It's bug involves the first value READ (before the start of the loop) (see 1). She reads in a starting value, but then sets NUM (the Input New Value Variable) to 0 (see 2), losing that starting value. From the protocol, it seems that she does this because she wanted to initialize all variables used inside the loop to 0.

For each subject, notice how the run-time behavior of the final program would not clearly reflect the actual student misconceptions. With the protocols we were able to follow each subject and account for many of the details of the final program in terms of the student's knowledge and strategies. This illustrates the importance of the protocol data in formulating the theory of novice programming bugs presented here.

Table 4-1 contains overview statistics for the four protocols analyzed. Interviews lasted between 23 and 45 minutes. It is not clear why subjects 2 and 4 took nearly twice as long as subjects 1 and 3. Notice that time spent on the problem does not appear to have any relationship to the number of bugs. Plan evidence item counts ranged between 104 and 177, with SSK plan evidence item counts ranging between 16 and 50 while PK plan evidence item counts ranged between 88 and 127. The subjects had between 10 and 32 bugs. Notice that in terms of Plans and Bugs, there seem to be two groups in the protocols: Protocols 1 and 2 show greater use of plans and more bugs then protocols 3 and 4. We refer to these as the "buggy group" and the "less-buggy group", respectively. In the rest of the section we present evidence that seems to explain this grouping. It seems that the buggy group shows substantially more use of SSK then

```
Const
   Sentinel = 99999;
Var
   I, Count, total, AVG : Integers;
Begin
   Count := 0;
   Total := 0;
   Writeln ('Enter integer');
   Readln;
   Read (integer);
   While I <> 99999 Do
      Begin
         Count := Count + 1;¹
         Total := Total + I²
         Readln ⁴
      End
   Read (I)³
   Avg := Total Div Count
   Writeln ('Avg is ', Avg:0)
```

(superscripts mark lines discussed in the text)

**Figure 4-6:** Final program for Subject 11

```
PROGRAM SUMUP (INPUT/,OUTPUT);
CONST
   SENTINEL = 99999
VAR
   NUM1, SUM, AVERAGE, COUNT : INTEGER;
BEGIN
   WRITELN ('READ IN AN INTEGER AND CONTINUE UNTIL');
           ('FINISHED THEN ENTER 99999 . . . ');
   READLN;
   READ (NUM);¹
   COUNT := 0;
   SUM := 0;
   NUM := 0;²
   WHILE NUM <> SENTINEL DO
      BEGIN
      SUM := NUM + SUM;
      COUNT := COUNT + 1;
      READLN;
      READ (NUM)
      END;
AVERAGE := SUM DIV COUNT;
WRITELN ('AVERAGE IS, ' AVERAGE : 0);
END.
```

(superscripts mark lines discussed in the text)

**Figure 4-7:** Final program for Subject 12

the less-buggy group. This correlation between buggyness and SSK is the substance of expectation three (and a key finding of the work) and is discussed further below.

| | Protocol Number | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| **Subject Number** | 13 | 6 | 11 | 12 |
| **Duration** (minutes) | 32 | 45 | 23 | 45 |
| **Plan Evidence Items** | 159 | 177 | 104 | 107 |
| SSK Plans | 43 | 50 | 16 | 16 |
| PK Plans | 116 | 127 | 88 | 91 |
| Ratio: PK to SSK | .37 | .39 | .18 | .18 |
| **Plans/Minute** | 5.0 | 3.9 | 4.5 | 2.4 |
| **Bugs**[1] | 32 | 19 | 11 | 10 |

**Notes:**
[1]Bugs (errors) found in the protocol.

**Table 4-1:** Summary Statistics for the Four Protocols Analyzed

In the beginning of this section we developed three expectations based on the theory of novice programming bugs. Using the methodology developed and illustrated in the preceeding few sections, and the data just summarized, we now evaluate those expectations.

### 4.4.2. Expectation 1: Plan Use Is Extensive

The first expectation concerned the use of plans in the protocols:

**Expectation 1:** Novices show a regular and extensive use of plans. Novices should make use of *both* SSK and PK implementation plans.

In Table 4-1 the relevant statistics are presented. There were 104 plan evidence items in the protocol with the fewest plan evidence items (protocol 3). This protocol averaged 2.4 plan evidence items per minute. The protocol with the most evidence items had 177, averaging 3.9 plan evidence items per minute. Our subjects did seem to use plans extensively, though it is difficult to measure what "extensive" means without a baseline for comparison.

There are two ways to break down plan usage: by SSK or PK, and by plan type. Data in Table 4-1 shows that the subjects used both SSK and PK plans. The ratio of PK to SSK ranged between .18 and .39. The buggy group protocols had ratios of .37 and .39 while the less-buggy group both had ratios of .18. That is, the buggy protocols showed more use of SSK plans. This relationship is discussed further in below.

In Table 4-2 we show the the plan evidence item counts broken down by type of plan. Again, there is overall support for the importance of plans. In particular, all the plans discussed above were used by the novices interviewed. All the subjects, including the buggiest novice (protocol 1), used almost every type plan.

| | Protocol Number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | | 4 | |
| **Subject** | 13 | | 6 | | 11 | | 12 | |
| **Plan Evidence Items** | | (%) | | (%) | | (%) | | (%) |
| Sentinel Variable | 9 | (7) | 8 | (6) | 4 | (5) | 10 | (11) |
| Loop | 19 | (15) | 10 | (7) | 7 | (8) | 1 | (1) |
| Counter Variable | 21 | (16) | 15 | (11) | 15 | (18) | 12 | (13) |
| Arithmetic Sum | 18 | (14) | 31 | (22) | 11 | (13) | 14 | (15) |
| Result Variable | 17 | (13) | 10 | (7) | 6 | (7) | 10 | (11) |
| New Value Variable | 11 | (9) | 12 | (9) | 12 | (14) | 14 | (15) |
| Input | 15 | (12) | 19 | (13) | 15 | (18) | 7 | (8) |
| New Value Loop | 8 | (6) | 28 | (20) | 11 | (13) | 16 | (17) |
| Result Output | 5 | (4) | 5 | (3) | 0 | (0) | 1 | (1) |
| Instructional Output | 3 | (2) | 0 | (0) | 2 | (2) | 0 | (0) |
| Prompt Output | 2 | (2) | 1 | (1) | 2 | (2) | 7 | (8) |
| Illegal Filter | 0 | (0) | 2 | (1) | 0 | (0) | 0 | (0) |
| **Total** | 128 | (100) | 141 | (100) | 85 | (100) | 92 | (100) |
| **Bugs** | 32 | | 19 | | 11 | | 10 | |

**Table 4-2:** Break Down of Plan Usage Statistics for the Analyzed Protocols.

Detailed analysis of these data will require finer grained analysis methods then used in this study. To illustrate the problems of drawing simple conclusions from these data, consider several rows from the table:

**New Value Controlled Loop Plan**

In protocol 1, where the subject knew very little about constructing the loop, the count is quite low, accounting for only 6% of the plan evidence items. Other subjects, whose loops were fairly close to correct, had higher counts ranging between 13% and 20% of all plan evidence items.

**Sentinel Variable Plan**

All subjects had little trouble with this plan and had relatively low plan evidence item counts, ranging between 5% and 11% of all plan evidence items.

**Input New Value Variable Plan** and **Input Plan**

These plans gave all subjects a great deal of trouble. Plan evidence items counts (for the two together) range between 18% and 33% of all plan evidence items.

From these and similar observations, a very tentative conclusion can be drawn that plan activity is highest on those plans that a novice *almost* knows. Further work is needed, however, to conclude this with any amount of rigor.

In summary, the novice programmers studied did use plans. The protocols ranged from 104 to 177 plan evidence items, or from 2.4 to 5.0 evidence items per minute on average. All novices used both SSK and PK plans. Plans of all types discussed are used by all the novices here. Even without an established baseline, for saying usage was "extensive", the numbers do support the idea that plans play an important part in the novice programming process.

### 4.4.3. Expectation 2: Bug Generators Plausibly Explain the Bugs

Expectation 2 addresses the adequacy of the bug generator set to plausibly explain all bugs:

**Expectation 2:** The Bug Generator set presented here can plausibly explain most errors found in the protocols.

The data are in the "Bugs Explained" line of Table 4-3. All bugs found can be plausibly explained by one or more of the Bug Generators presented. The bug generator set presented could serve as a concise way to organize and categorize bugs. Particularly useful is that the relatively small set of bug generators is sufficient to describe many different bugs and misconceptions.

Table 4-3 also shows the total number of plausible explanations in each protocol and the number of plausible explanations per bug. These statistics indicate that the number of plausible explanations per bug is relatively constant from protocol to protocol. This gives us some confidence that the bug analysis techniques have been consistently applied across the four

| | Protocol Number | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| **Subject Number** | 13 | 6 | 11 | 12 |
| **Bugs Found** | 32 | 19 | 11 | 10 |
| **Bugs Explained**[1] | 32 | 19 | 11 | 10 |
| **Total Explanations**[2] | 56 | 27 | 15 | 15 |
| **Explanations/Bug**[3] | 1.7 | 1.4 | 1.4 | 1.5 |

**Notes:**
[1]All bugs found were explained.
[2]Number of plausible bug generator explanations for all bugs.
[3]Average number of plausible (Non-Slip) explanations per bug.

**Table 4-3:** Bug Generator Statistics for the Analyzed Protocols

protocols. This data also indicates that while multiple bug generators may be responsible for some bugs, on the average less than two bug generators will be involved.

The plausible bug generator explanations summarized in Table 4-3 do not include cases where the bug was plausibly explained as a slip. These data are summarized in Table 4-4. First, notice that all but two bugs had a plausible non-slip explanation. That is, almost every bug detected in the analysis can be plausibly explained as a bug generator operating on plans. Second, notice that slips can plausibly explain between 34% and 58% of the bugs (depending on the protocol). Most likely a much smaller percentage are actually slips. Though there are many bugs that can plausibly be interpreted as slips, this slip interpretation often becomes less likely when the bug is examined in the context of the whole protocol. Consider, for example, the first bug of Figure 4-3 (on page 45). Although we consider a slip plausible in the bug annotation, the overall context of the bug makes a slip unlikely. In particular, it is unlikely that the subject actually understands how reading is done given that he never includes a **Read** statement in his loop body. This is typical for many of the slips. Notice that in the second example of Figure 4-3 it is not possible to construct a plausible slip. In this case the subject both wrote and spoke things indicating a non-slip explanation.

| | Protocol Number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | | 4 | |
| **Subject Number** | 13 | | 6 | | 11 | | 12 | |
| **Bugs Found** | 32 | | 19 | | 11 | | 10 | |
| **Bugs Explained**[1] | 32 | | 19 | | 11 | | 10 | |
| **Bugs Plausibly Explained by Non-Slip Bug Generators**[2] | | (%) | | (%) | | (%) | | (%) |
| | 32 | (100) | 18 | (95) | 10 | (91) | 10 | (100) |
| **Bugs Plausibly Explained by Slip Bug Generators**[3] | 11 | (34) | 11 | (58) | 5 | (45) | 5 | (50) |

**Notes:**
[1]All bugs found were explained.
[2]The number (percent) of bugs plausibly explained by the Non-Slip Bug Generators.
[3]The number (percent) of bugs plausibly explained by the Slip Bug Generator.

**Table 4-4:** Slip vs. Non-Slip Bug Generator Statistics

### 4.4.4. Expectation 3: SSK/PK Bug Generators Are Critical

We have claimed that novice bugs are closely connected to confounds between SSK and PK. This is expectation 3:

> **Expectation 3:** When novices encounter an impasse in a developing programming solution, they usually use *SSK Confounds PK* Bug Generators to patch and continue.

The analysis presented here contains two kinds of evidence for this expectation. We look both at the overall coverage of *SSK Confounds PK* Bug Generators and the relationship between total number of bugs and the ratio of SSK plan usage to PK plan usage.

*SSK Confounds PK* Bug Generators embody the process whereby novices use SSK knowledge to reason about their programs. As can be seen in the "SSK/PK Total" line of Table 4-5, *SSK Confounds PK* Bug Generators plausibly explained 46%, 63%, 60%, and 27% of the bugs found in protocols 1 to 4 respectively. That is, for protocols 1 to 3, *SSK Confounds PK* Bug Generators can explain almost almost one-half or more of the bugs. In protocol 4, with the lowest coverage by *SSK Confounds PK* Bug Generators, *Intra-PK* Bug Generators plausibly explained 47% of the errors. This evidence, while not conclusive, does support expectation 3.

| | Protocol Number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | | 4 | |
| **Subject Number** | 13 | | 6 | | 11 | | 12 | |
| **Bugs Found** | 32 | | 19 | | 11 | | 10 | |
| **Bugs Explained**[1] | 32 | | 19 | | 11 | | 10 | |
| **Explanations/Bug**[2] | 1.7 | | 1.4 | | 1.4 | | 1.5 | |
| **Plausible Explanations By Non-Slip Bug Generators**[3] | | | | | | | | |
| | | (%) | | (%) | | (%) | | (%) |
| **SSK Confounds PK:** | | | | | | | | |
| PL Used as NL | 10 | (18) | 6 | (22) | 1 | (7) | 1 | (7) |
| NL Interprets PL | 11 | (20) | 10 | (37) | 7 | (47) | 3 | (20) |
| NL Construct | 2 | (4) | 0 | (0) | 0 | (0) | 0 | (0) |
| Generic Name | 2 | (5) | 1 | (4) | 1 | (7) | 0 | (0) |
| **SSK/PK Total** | **26** | **(46)** | **17** | **(63)** | **9** | **(60)** | **4** | **(27)** |
| **Intra-PK:** | | | | | | | | |
| Trace | 10 | (18) | 5 | (19) | 1 | (7) | 1 | (7) |
| Overgen | 4 | (7) | 1 | (4) | 2 | (13) | 3 | (20) |
| Similar | 4 | (7) | 1 | (4) | 2 | (13) | 3 | (20) |
| **Intra-PK Total** | **18** | **(32)** | **7** | **(26)** | **5** | **(33)** | **7** | **(47)** |
| **Other Confounds PK:** | | | | | | | | |
| Multi-role | 8 | (14) | 1 | (4) | 1 | (7) | 3 | (20) |
| Other Domain | 3 | (5) | 2 | (7) | 0 | (0) | 1 | (7) |
| OS Confound | 1 | (2) | 0 | (0) | 0 | (0) | 0 | (0) |
| **Other/PK Total** | **12** | **(21)** | **3** | **(11)** | **1** | **(7)** | **4** | **(27)** |
| **Non-Slip Total** | **56** | **(100)** | **27** | **(100)** | **15** | **(100)** | **15** | **(100)** |

**Notes:**

[1]All Bugs found were explained.

[2]Average number of plausible (Non-Slip) explanations per bug.

[3]The number of bugs plausibly explained by each Non-Slip bug generator.
(%) is the percentage of plausible explanations by the specified bug generator out of all plausible Non-Slip explanations. Each bug can have several explanations. Slip data are presented in Table 4-4.

**Table 4-5:** Bug Generator Statistics by Bug Generator Type

Notice that protocol 4, the least buggy, also had the lowest percentage of plausible *SSK Confounds PK* Bug Generators.

There is a second set of data supporting Expectation 3. Refering back to Table 4-1, we see that the buggy group (protocols 1 and 2) of protocols had higher ratios of SSK to PK plan evidence items then the less-buggy group (protocols 3 and 4). The ratios were .37 and .39 for the buggy group protocols and .18 for both protocols of the less-buggy group. This is consistent with the notion that a more error prone and less advanced novice is doing more reasoning from Natural Language implementation plans.

## 5. Concluding Remarks

In the preceeding section we presented a detailed analysis of four protocols of novice programmers. From that analysis we provide data to support the theory of novice programmer bugs. Specifically, we present evidence that plans, both SSK and PK, are critical and pervasive in the work of novice programmers. We demonstrated that our Bug Generator set can plausibly explain all the novice bugs we have seen in our protocols. Most importantly, the analysis provides support our key assertion: SSK plays an important role in the bugs of novice programmers. Bug generators that describe the confounds between SSK and PK can plausibly explain a substantial number of bugs in each of the protocols. In addition, the SSK to PK implementation plan ratio was higher for those novices with buggier protocols.

There are several next steps for this work. In the rest of this section we explore further development for the theory, implications for teaching, and implications for intelligent tutoring of novice programming.

### 5.1. The Theory of Novice Programming Bugs

Currently, our bug analysis specifies all bug generators which *could* account for a programming bug. We would like to develop the theory so that this is more specific: are there ways to detect exactly which bug generator(s) is (are) responsible? To do this, it will be necessary to further develop the bug generator set. The different bug generators need to be more carefully and formally distinguished. There are situations where bug generators seem closely tied to specific programming constructs. Does this mean that some bug generators are tied to specific sections of the PK or SSK?

In the current theory the application of the bug generators is still largely informal. While constrained by plans, the application of bug generators still is represented by an explanatory paragraph. The theory is now cast in way that makes it easy to modify or extend plans and bug generators to account for a new bug. A more principled theory requires more constrained and formal application of the bug generators to specific bug situations. Modifying a plan or bug generator to create one explanation needs to have consequences for other explanations using that bug generator.

To address these issues, a more constrained set of interviews is planned. In these interviews subjects would be put in specific situations where bugs are very likely. For example, subjects might be given a problem statement and a skeleton program missing a key section. Such interviews would be accompanied by probes to detect the use of certain bug generators and plans.

The theory developed for novice bugs in the domain of programming may be applicable to novice bugs in other expert problem solving domains. In particular, the theory might be applicable to domains where there is a body of proto-knowledge that many students are likely to apply to the new domain. Step-by-step Natural Language procedures are proto-knowledge for programming. Similarly, Aristotelian Physics [DiSessa 82] may serve as analogous proto-knowledge for Classical Newtonian Physics. Matz [Matz 82] has shown how basic arithmetic knowledge serves as proto-knowledge for the learning of high-school algebra.

## 5.2. Teaching Novice Programmers

There are several ways that this work can contribute to the education of programmers. We have shown the usefulness of SSK in understanding many novice programming bugs. Unfortunately, novice programmers are not told about *SSK Confounds PK* bugs and SSK is rarely if ever discussed in our introductory programming courses. We suggest that introductory programming curriculum needs to bridge between SSK and PK. It is not unusual to see introductory programming texts that emphasize careful problem analysis and design before coding. Unfortunately, starting with a typical "pseudo-code program" often has already skipped a key step between the Natural Language implementation techniques and Pascal implementation techniques.

Consider an example of this contrast between the two kinds of implementation techniques. The Ending Value Averaging Problem (see Figure 4-1 on page 40) computes the average of a series values. In a programming implementation, each value is entered, incorporated, and is then no longer needed. A typical pseudo-code program looks like Figure 5-1, quite close to the final Pascal implementation.

```
Read a data item
while more data do
    Sum and Count that data item
    Read the next data item
Compute the average: Sum/Count
Write out the average
```

**Figure 5-1:** Fragment of Pseudo-Code Solution to the Ending Value Averaging Program

In a Natural Language implementation all values are entered and processed as a group. A novice, told to "write a plan of the program, using English or Pascal to express the important steps of the program", produced the Natural Language procedure in Figure 5-2.

*Keep reading data items until the final value is seen.*
*Don't use the final value.*
*Compute the average by dividing the sum of the values*
  *read by how many values there were.*
(Adapted from a response by one of our subjects.)

**Figure 5-2:** Fragment of Typical Novice Natural Language for the Ending Value Averaging Problem

There is a gap between these two approaches. That gap is exactly the knowledge captured in implementation plans. The gap can be bridged by explicitly supporting novice programmers in moving beyond their Natural Language implementation plans. This could be done by discussing the design process in two steps: a preliminary, design based on Natural Language implementation plans, and then a design based on programming language implementation plans. Introducing this distinction allows the novice to explicitly address bridging the gap.

We propose a curriculum organized around common programming plans. In the curriculum, these plans would first be discussed as implemented in SSK, and then discussed as implemented in PK for a programming language like Pascal. These plans would then be used to motivate and introduce the actual programming language constructs. This would be a significant departure from current programming curriculum [Cooper and Clancy 82, Dale and Orchalick 83]. Despite material on program design, most programming courses are still organized around the syntax and semantics of constructs in the programming language being taught: an `if` statement chapter followed by a `while` loop chapter followed by a chapter on `procedures`. Knowledge of programming plans available only implicitly by studying examples.

## 5.3. Conclusion

In this work we propose a theory of novice programmer bugs and a methodology for studying novice programmers. In the theory we characterize the knowledge and processes used by novice programmers. Specifically, we have found that novice knowledge of Natural Language step-by-step procedural knowledge seems to shape the kind of errors novice programmers make when working with a programming language like Pascal.

There is currently an explosion in the number of people owning and using computers. It is still unclear whether these millions of new computer users will be programming their machines, or simply using software developed by others. Currently, many claim that programming is just too hard for the average person. This work indicates that such a conclusion may be premature. Novice difficulty with simple programming may not be inherent in the nature of programming. Instead, the difficulty may result from novice programming languages and introductory programming courses that do not take novices pre-programming knowledge into account.

## Acknowledgements

The author wishes to acknowledge the guidance and patience of Elliot Soloway. His comments and careful reading have contributed significantly this work.

## References

[Anderson 83] Anderson, John R.
*The Architecture of Cognition.*
Harvard University Press, Cambridge, Massachusetts, 1983.

[Biermann 83] Biermann, Alan W., Ballard, Bruce W., Sigmon, Anne H.
An Experimental Study of Natural Language Programming.
*International Journal of Man-Machine Studies* 18:71-87, 1983.

[Bonar 79] Bonar, Jeffrey G.
*Just So Bugs: Stories about Novice Programming Bugs.*
Technical Report, University of Massachusetts, Computer and Information Science Department, 1979.

[Bonar 84] Bonar, Jeffrey G.
*Understanding the Bugs of Novice Programmers.*
PhD thesis, University of Massachusetts, 1984.

[Bonar 85] Bonar, Jeffrey G.
*Personal Programming in BASIC.*
Academic Press, 1985.
In preparation.

[Bonar et al. 82] Bonar, Jeffrey, Ehrlich, Kate, Soloway, Elliot, and Rubin, Eric.
Collecting and Analyzing On-Line Protocols from Novice Programmers.
*Behavioral Research Methods and Instrumentation* , May, 1982.

[Brown and VanLehn 80]
Brown, John Seely, and VanLehn, Kurt.
Repair Theory: A Generative Theory of Bugs in Procedural Skills.
*Cognitive Science* 4:379-426, 1980.

[Chi et al. 81] Chi, Michelene T., Feltovich, P., Glaser, R.
Categorization and Representation of Physics Problems by Experts and Novices.
*Cognitive Science* 5(2):121-152, 1981.

[Chi et al. 82] Chi, Michelene T. H., Glaser, Robert and Rees, Ernest.
Expertise in Problem Solving.
In Sternberg, Robert (editor), *Advances in the Psychology of Human Intelligence.* Lawrence Erlbaum and Associates, Hillsdale, New Jersey, 1982.

[Clement 81] Clement, John.
Cognitive Microanalysis: An Approach to Analyzing Intuitive Mathematical Reasoning Processes.
In Wagner, S. and Geeslin, W. (editors), *Modeling Mathematical Cognitive Development.* ERIC Center for Science, Mathematics, and Envirnmental Education, Ohio State University at Columbus, 1981.

[Clement 82] Clement, John.
Students' Preconceptions in Introductory Mechanics.
*American Journal of Physics* 50(1):66-71, January, 1982.

[Codd 74] Codd, E. F.
*Seven Steps to RENDEVOUS With the Casual User.*
IBM Report J1333 (29842), IBM, 1974.

[Cooper and Clancy 82]
Cooper, Doug and Clancy, Michael.
*Oh! Pascal!*
W.W. Norton and Company, New York, 1982.

[Dale and Orchalick 83]
Dale, Neil, and Orshalick, David.
*Introduction to PASCAL and Structured Design.*
D.C. Heath and Company, Lexington, Massachusetts, 1983.

[De Remer and Kron 76]
De Remer, Frank, and Kron, Hans.
Programming-In-The-Large Versus Programming-In-The-Small.
*IEEE Transactions on Software Engineering* , June, 1976.

[Dijkstra 78] Dijkstra, Edsger W.
On the Foolishness of Natural Language Programming.
1978.
EWD Technical Note.

[DiSessa 82] DiSessa, Andrea A.
Unlearning Aristotelian Physics.
*Cognitive Science* 6(1):37-76, Jan.-Mar., 1982.

[Ehrlich and Soloway 83]
Ehrlich, Kate and Soloway, Elliot M.
An Empirical Investigation of the Tacit Knowledge in Programming.
In Thomas, John and Schneider, Michael L. (editors), *Human Factors in Computer Systems.* Ablex, Inc., 1983.

[Ericsson and Simon 80]
Ericsson, K. Anders and Simon, Herbert.
Verbal Reports as Data.
*Psychological Review* 87(3):215-251, May, 1980.

[Jeffries et al. 81]
Jeffries, Robin, Turner, Althea A., Polson, Peter G., and Atwood, Michael E.
The Processes Involved in Designing Software.
In Anderson, John R. (editor), *Cognitive Skills and Their Acquisition.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981.

[Johnson and Soloway 83]
Johnson, W.Lewis and Soloway, Elliot.
*Proust: Knowledge-Based Program Understanding.*
RR 285, Yale University Computer Science Department, August, 1983.

[Johnson et al. 82]
Johnson, W. Lewis, Draper, Stephen, and Soloway, Elliot.
Classifying Bugs is a Tricky Business.
In *Proceedings of the Seventh Annual NASA/Goddard Workshop on Software Engineering.* NASA, Baltimore, 1982.

[Kant and Newell 83]
Kant, Elaine and Newell, Allen.
Problem Solving Techniques for the Design of Algorithms.
*Information Processing and Management* , 1983.

[Kaput 79] Kaput, James J.
Mathematics and Learning: Roots of Epistemological Status.
In Lochhead, Jack and Clement, John (editors), *Cognitive Process Instruction.* Franklin Institute Press, Philadelphia, 1979.

[Konold and Well 81]
Konold, Clifford E. and Well, Arnold D.
Analysis and Reporting of Interview Data.
In *Proceedings of the Annual Meeting of the American Educational Research Association.* American Educational Research Association, 1981.

[Larkin et al. 80] Larkin, Jill H., McDermott, John, Simon, Dorothea P., Simon, Herbert A.
Expert and Novice Performance in Solving Physics Problems.
*Science* 208:1335-1342, 1980.

[Matz 82] Matz, Marilyn.
Towards a Process Model for High School Algebra Errors.
In Sleeman, Derek and Brown, John Seely (editors), *Intelligent Tutoring Systems.* Academic Press, London, 1982.

[Miller 74] Miller,Lance A.
Programming by Non-Programmers.
*International Journal of Man-Machine Studies* (6):237-260, 1974.

[Miller 81] Miller, Lance A.
Natural Language Programming: Styles, Strategies, and Contrasts.
*IBM Systems Journal* 20(2):184-215, 1981.

[Minsky 75] Minsky, Marvin.
A Framework for Representing Knowledge.
In Winston, Patrick H. (editor), *The Psychology of Computer Vision.* McGraw-Hill, New York, 1975.

[Newell 77] Newell, Allen.
On the Analysis of Human Problem Solving Protocols.
In Johnson-Laird, P. N. and Wason, P. C. (editors), *Thinking*, pages 46-61. Cambridge University Press, Cambridge, 1977.

[Newell and Simon 72]
Newell, Alan and Simon, Herbert.
*Human Problem Solving.*
Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[Pollatsek et al. 81]
Pollatsek, A., Lima, S., and Well, A. D.
Concept or Computation: Students' Understanding of the Mean.
*Educational Studies in Mathematics* 12:191-203, 1981.

[Resnick 82] Resnick, Lauren B.
Syntax and Semantics in Learning to Subtract.
In Carpenter, T., Moser J., and Romberg T. (editors), *Addition and Subtraction: A Cognitive Perspective*. Erlbaum, Hillsdale, NJ, 1982.

[Resnick 83] Resnick, Lauren B.
A New Conception of Mathematics and Science Learning.
*Science* 220:477-478, April, 1983.

[Rich 81] Rich, Charles.
*Inspection Methods in Programming.*
Technical Report AI-TR-604, MIT, 1981.
MIT AI Lab.

[Rich and Shrobe 78]
Rich, Charles and Shrobe, Howard.
Initial Report on a LISP Programmer's Apprentice.
*IEEE Transactions on Software Engineering* 4(6), November, 1978.

[Rissland 78] Rissland, Edwina.
Understanding Understanding Mathematics.
*Cognitive Science* 2(4):361-383, 1978.

[Rosnick 82] Rosnick, Peter.
*Student Conceptions of Semantically Laden Letters in Algebra.*
Technical Report, University of Massachusetts, Amherst, Cognitive Development Project, 1982.

[Shneiderman and Mayer 79]
Shneiderman, Ben, and Mayer, Richard.
Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results.
*International Journal of Computer and Information Sciences* 8(3):219-238, 1979.

[Sime et al. 77] Sime, M. E., Green, T. R. G., and Guest, D. J.
Scope Marking in Computer Conditionals -- A Psychological Evaluation.
*International Journal of Man-Machine Studies* 9:107-118, 1977.

[Soloway et al. 81]
Soloway, Elliot, Woolf, Beverly, Rubin, Eric, and Barth, Paul.
MENO-II: An Intelligent Tutoring System for Novice Programmers.
In *Proceedings of International Joint Conference in Artificial Intelligence.* IJCAI, Vancouver, British Columbia, 1981.

[Soloway et al. 82a]
Soloway, Elliot, Ehrlich, Kate, Bonar, Jeffrey, and Greenspan, Judith.
What Do Novices Know About Programming?
In Shneiderman, Ben, and Badre, Albert (editors), *Directions in Human-Computer Interactions.* Ablex Publishing Company, 1982.

[Soloway et al. 82b]
Soloway, Elliot M., Lochhead, Jack, and Clement, John.
Does Computer Programming Enhance Problem Solving Ability?
In Seidel, R., Anderson, R., and Hunter, B. (editors), *Computer Literacy*.
Academic Press, New York, 1982.

[Soloway et al. 83]
Soloway, Elliot, Bonar, Jeffrey, and Ehrlich, Kate.
Cognitive Strategies and Looping Constructs: An Empirical Study.
*Communications of the Association For Computing Machinery* , November, 1983.

[Webster 75]
G. & C. Merriam Co.
*Webster's New Collegiate Dictionary*.
G. & C. Merriam Co., 1975.

[Young and O'Shea 81]
Young, Richard M. and O'Shea, Tim.
Errors in Children's Subtraction.
*Cognitive Science* 5(2):153-177, April-June, 1981.

[Zaks 80]
Zaks, Rodney.
*Introduction to Pascal*.
Sybex Press, California, 1980.

## Appendix I. The Plans

In this section we present a set of key plans necessary for novice programming. Each of the plans is presented in three forms: a tactical plan, a Pascal implementation plan, and a Natural Language implementation plan. For each set of plans, we particularly emphasize the distinctions between the Pascal and Natural Language implementations.

There has been a great deal of work with programming plans. This includes formal definition [Rich 81], empirical investigations [Soloway et al. 82a], and implementation in AI systems [Johnson and Soloway 83, Rich and Shrobe 78, Soloway et al. 81]. Plans are flexible and adaptable as a knowledge representation scheme. Rissland [Rissland 78], for example, uses a plan-like representation as a pedagogical tool in the domain of linear algebra. In this dissertation we focus on plans as a tool for coding and analyzing protocols of novice programmers working on programming problems. The plan set presented here reflects only one possible interpretation of the underlying knowledge of novice programmers. As more work is done with these plans the details are likely to evolve. These plans are not a definitive set, but rather a useful characterization of knowledge needed for a novice programmer to acquire expertise.

Only the plans actually seen in the Ending Value Averaging Problem are discussed in detail, since only those plans were used in a detailed analysis of novice programming protocols. In that use, these plans were developed, refined, and specified in detail.

## Sentinel Variable Plan

The Sentinel Variable is a used to hold or represent a value that marks the end of data in an input stream or data structure. Typically a sentinel is some distinguished value which is recognized by the program.

## Arithmetic Sum Variable Plan

The Arithmetic Sum Variable holds the sum of a series of values encountered during program execution. Notice that the Natural Language implementation criteria refer to adding a body of numbers while Pascal implementation refer to repeatly adding in an additional single number. That is, Pascal implements a running total while Natural Language implements a long sum.

## Counter Variable Plan

The Counter Variable is a kind of Arithmetic Sum Variable that always adds in 1 (see Figure 3-7). Notice that, like the Arithmetic Sum Variable, the Natural Language implementation refers to global counting operations over a whole set of items while the Pascal implementation refers to individual counting operations for each item. Also note that in the Natural Language implementation the count operation is specifically after the operation being counted.

## Result Variable Plan

The Result Variable Plan is used to save the result of some expression. It is used in cases where one does not want to repeat the computation of an expression, or wants to label some result. Note that the Pascal implementation plan requires a Pascal specification of the computation needed to produce the result, where the Natural Language implementation allows more flexibility. Concern about division by zero is considered an indication of a Pascal implementation plan.

## Input New Value Variable Plan

The Input New Value Variable holds a newly generated value that has been input by the user. The Pascal implementation plan is described with phrases where the subject refers to moving items "in". Values are discussed as if there is some object that they are moved into. In the Natural Language implementation, on the other hand, one sees the subject use phrases that are synonymous with "examine". Also in the Natural Language implementation, an Input New Value will often be refered to positionally in a sequence of values, i.e. "next . . . " or "previous . . . " (criteria 6 and 7).

## Illegal Value Filter Plan

The Illegal Value Filter Plan protects a piece of computation that would fail if some particular value were used in that computation. The Natural Language implementation usually adds this test as a special case after the computation being protected. The Pascal implementation, however, surrounds the computation being protected in a branch of an `if` statement.

## Successive Case Conditional Plan

The Successive Case Conditional Plan chooses from a succession of related cases. The conditions that apply to each case are not necessarily dependent from the others. That is, failing test 1 and 2 may not say anything about success on case 3. In the Pascal implementation, successive cases are usually nested, enforcing an ordering of testing and exploiting related alternatives to save redundant tests. In the Natural Language implementation the cases are usually arranged flat - without nesting: there is no assumption about order, and alternatives, no matter how related, are treated completely separately.

## General Looping Plan

Several subjects explicitly mentioned that they knew they needed a loop, but didn't know what kind. They later became more specific about the loop. The General Looping Plan was used to indicate the knowledge in use before they became specific about the loop.

## New Value Controlled Loop Plan

This is the loop plan used with the Ending Value Averaging Problem.

## Results Output Plan

This plan is used to print out the results of a computation. The NL implementation often leaves this operation implicit, assuming that a description of the result value is sufficient. The Pascal implementation uses a `write` statement.

### Instruction Information Output Plan

This plan is used to give the program user instructions on how to use the program. In this case there is no Natural Language implementation since the Natural Language procedure is itself an instance of "Instruction Information Output Plan".

### Prompt Output Plan

The Prompt Output Plan is designed to request information from the program user. In the Natural Language implementation, this involves verbs like "tell" or "ask". The Pascal implementation involves use of a Pascal `Write` statement immediately followed by a Pascal `Read`.

## Appendix II. The Bug Generators

In this appendix we present the specific **bug generators** developed to account for the bugs of novice programmers solving programming problems. In the theory presented in section 2, we describe how novices solving a programming problem encounter **impasses**: places where they get stuck in their developing solution. These impasses are caused when the novice encounters gaps or inconsistencies in the PK. Bug generators allow a novice to patch an impasse and continue solving the problem. Bug generators work by exploiting parallels between PK, SSK, and other knowledge possessed by the novice. Simply, the bug generators bridge the gaps in PK with other knowledge. Bug Generators are so named because in patching an impasse they almost always introduce a bug.

The bug generators described here are used to analyze bugs in protocols of novice programmers. Many details about bug generators are still under development. The details in this appendix represent a certain level of maturity in the bug generator set, but the work is not finished. In the bug generator descriptions shown below, each bug generator is described and illustrated with examples. The "criteria" refered to appear in the Pascal language (PL) or Natural Language (NL) implementation plan descriptions that appear in [Bonar 84].

### *SSK Confounds PK* Bug Generators: Confounding the Two Kinds of Knowledge

The first class of bug generators is based on the similarities between the SSK and PK domains. The bug generators exploit these similarities to reason about gaps in the PK with knowledge from SSK.

## PL Used as NL

The novice uses programming constructs as if they had their Natural Language meanings or omits programming constructs that would not be needed in Natural Language.

**Example**: A subject uses, as if it were legitimate Pascal, the `repeat. . . until. . . then` construct. He justifies this by saying:

> ". . . if then, while then, repeat until then, well it makes sense . . . " (Subject 13, protocol 1, 146)

In effect, he is reasoning from what sounds good in English (see Criteria 6 and 7, NL - New Value Controlled Loop Plan).

**Example**: A subject uses `if then` to control a process he has clearly identified as happening repeatedly (see Criteria 4 and 7, NL - New Value Controlled Loop Plan):

> "I only want to do that process as long as the sentinel's not been read. So uh, just for now [pause] let me see now, a while loop. No, I don't want a while [pause] I think just an if/then. . . " (Subject 6, protocol 2, 75)

(This example has been described in detail above.)

## PL INTERPRETED AS NL

The novice uses a programming language construct to implement a Natural Language plan.

**Example**: The word "repeat" in Natural Language suggests a series of specific steps to be repeated, e.g. "repeat that procedure with the other 3 cylinders" (see Criteria 6, NL- New Value Controlled Loop Plan). On the other hand, in English the word "while" suggests a global observer waiting for a condition to happen, e.g. "continue while the street is still two lanes". Subjects will apply this English language distinction to choose between the `while` and `repeat` loops in Pascal: "the repeat until will just repeat [hand motions indicating looping] what is entered and add it to, so that would be simpler than a while loop going through the whole, thing. We just want an average at the end." (Subject 13, protocol 1, 8).

**Example**: A subject puts a Read of the New Value Variable above the loop, but no read inside the loop. Nonetheless, s-he discusses the loop as if reading is happening when needed:

> "If the sentinel isn't the value read in, it's going to take the sum, it's going to take the count, it's going to go back and be ready to add in a new number. . . " (Subject 6, 187).

One interpretation of this bug is that the subject is reasoning from Natural Language where values are all entered at once and usage of the individual values is implicit (see Criteria 1, NL - Input Plan).

## NEW PL CONSTRUCT FROM NL

The novice invents a new programming construct based on a Natural Language implementation of the relevant plan.

**Example**: A subject writes the following:

```
New := next New
```

(Subject 13, protocol 1, 235). The subject is using "next" as if it were a programming keyword that returns the next value of the New Value Variable (see Criteria 6, NL - Input New Value Variable Plan).

## *Intra-PK* Bug Generators: Errors In Programming Knowledge

The next set of bug generators stem directly from missing knowledge in the novice's version of the expert knowledge base:

## EXECUTION ORDER

The novice bases programming language statement order on the program execution order. That is, the subject tries to order the program as if it was an execution trace.

**Example**: A subject is working on the Arithmetic Sum Variable, writing the update assignment for inside the loop body:

> "And then integer [the way the subject refers to the New Value Variable, I], or rather, sum equals integer, ahm, equals zero plus integer, [WRITES Sum := 0 + I] . . . [lower in the loop body] and then Sum equals integer plus integer,[WRITES Sum := I + I]" (Subject 13, protocol 1, 125 and 131).

It seems he has (more or less) written the execution trace of the first two assignments to the Sum variable.

**Example**: A subject is working on the update of the Arithmetic Sum Variable:

> "It'll be taking the sum of that plus the value of all the previous ones. Which I suppose should be another variable. Uh, total, yes, that'll be another, I'll assign that another variable value for now." (Subject 6, protocol 2, 18).

There is no need for another variable, but it seems that he has made this intermediate value sufficiently concrete that he gives it a variable.

## GENERIC NAMES

The novice names parts of the program or variables generically, based on common programming language implementation strategies.

**Example**: A subject names the Input New Value Variable integer, because it will hold integers. (Subject 13, protocol 1 and Subject 12, protocol 5).

## PL OVERGENERALIZATION

The subject over-generalizes from one Pascal implementation plan to another.

**Example**: A subject wants to initialize the Result Variable (this variable holds the result of some computation) while initializing the Counter and Arithmetic Sum Variable:

> "I have to initialize the count, I have to initialize total, do I have to initialize the average?" (Subject 11, protocol 4, 45).

This initialization is not necessary.

**Example**: A subject assumes that since the while and for loops require a begin end around a multi-line loop body, so does the the repeat until loop (Subject 1, protocol 1, 199).

## TACTICAL SIMILARITY

The novice fails to distinguish between things that are similar on a tactical level, but implemented differently.

**Example**: A subject considers putting all assignment statements inside a loop body inside of Readln statements:

> ". . . so it reads each piece separately, it takes in the new . . ." (Subject 1, protocol 1, 233).

It seems that he is focusing on the variables acquiring a new value, and assuming that an input statement is needed to do this.

**Example:** A subject uses the assignment operator to give a declared constant a value. He does this while initializing the other variables (Subject 1, protocol 1, 66).

## *Other Confounds PK* Bug Generators: Errors From Other Knowledge

The final set of bug generators represent patches performed in a domain separate from the step-by-step Natural Language or Programming knowledge bases:

### MULTIPLE VARIABLE ROLES

The novice uses a single variable but gives multiple roles for that variable. This bug generator is drawn from the work of Rosnick [Rosnick 82]. He found that students often combine variable roles in a algebraic word problems. For example, in a problem involving the price and quantity of books, students would use one variable, B, to stand for both price and quantity. Novice programmers do something similar as a patch.

**Example:** A subject uses the same variable to stand for both the Input New Value and a Count of the input values that have been read (In this example $x$ is the Input New Value Variable and $n$ is the Counter Variable):

"I want to get a statement that is going to be clear that we're going to add the numbers, each number entered, we'll have the tally of the, number of integers entered . . . ahhh, N equals, ahmmm integer. [WRITES: $n$ := $x$]" (Subject 13, protocol 1, 114-116).

It seems that the subject is using this assignment both to count the number of elements entered and get these elements.

### OTHER DOMAIN

The novice uses an understanding of the problem from some non-programming (not PK or SSK) domain to produce an incorrect answer.

**Example:** A subject assumes that the variable $x$ will automatically increment, based on experience with the common mathematical and programming use of $i$ as a Counter (Subject 11, protocol 4, 75).

**Example**: A problem involves searching for the sentinel 99999. A subject thinks that there would never be legitimate values greater than 99999, so he uses `< 99999` as the test for the `while` loop, instead of the correct `<> 99999` (Subject 1, protocol 1, 83).

**OPERATING SYSTEM CONFOUND**

The novice confuses a programming language construct with some command or operation from the operating system.

**Example**: The subject reasons based on what he expects will be done when the Pascal compiler *reads* the code, not what happens when his `Read` statement is actually executed.

**Example**: The subject uses an editor command inside a program.

Finally, there is the Slip bug generator, which represents "slips of the tongue", misspeakings, etc.

**SLIP** The novice produced a random error out of being distracted, making a speech slip, or making a typographical error.

## Appendix III. A Sample of an Analyzed Protocol

This appendix contains a short segment from a fully analyzed protocol. This segment is taken from protocol one in the set of four protocols analyzed in depth and discussed in Section 4.

The subject of this segment is working on the Ending Value Averaging Problem. (See Figure III-1 for the problem statement and a typical correct solution.) At the point we pick up the protocol, he is coding the loop. The transcript below is annotated with the plan analysis and the bug analysis. In the course of the protocol he will write the code shown in Figure III-2. The subject's basic problem is that he does not understand how variables are used to process values inside the loop. As we will see in the protocol, the subject clearly understands that the loop body must accumulate a running total in the variable `Sum` and a count in the variable `N`. He also understands that the variable `I` will hold new values read from the user. Explanations for the peculiar code in the loop body are presented in the bug analysis annotations with the transcript segment shown below.

## The Problem

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

## Typical Correct Solution

```
program While_Average_Solution (Input, Output);
    const Sentinel = 99999;
    var Count, Sum, New : integer;
begin
Count := 0;
Sum := 0;
Readln (New);
while New <> 99999 do
    begin
    Sum := Sum + New;
    Count := Count + 1;
    Readln (New)
    end;
if Count > 0
    then Writeln ('The average is ', Sum/Count)
    else Writeln ('There were no numbers to average.')
end.
```

**Figure III-1:** A Correct Solution to the Ending Value Averaging Problem

There are several things the reader should note while reading the protocol segment. Generally, notice the richness of the reasoning used by Subject 13. Although he has a number of bugs, he is not working at random. Also, notice that the bug generators proposed as plausible explanations for each bug are parameterized by the plans recently used by the subject.

```
1 )  N := 0;
2 )  Sum := 0;
3 )  repeat
4 )      Sum := 0 + I
5 )      N := 1
6 )      Sum := I + I
7 )      N := 2
8 )  until
```

(Line numbers are refered to by the protocol on pages 75 to 78.)

| | |
|---|---|
| N | is the Counter Variable, |
| Sum | is the Arithmetic Sum (Running Total) Variable |
| I | is the Input New Value Variable |

**Figure III-2:** Code Written In the Short Example Protocol Analysis

Here is the protocol segment:

124 **Subject 13:** Ummm, ummm, well I think its, ahh, its [N := I, eventually changed to N := 1, eventually put on line 5] not right I don't think, but I I'm gonna leave it that way for the moment.

125 **Interviewer:** OK, Fine

126 **Subject 13:** And then integer [the way the subject refers to the variable I], or rather, sum equals integer, ahm, equals zero plus integer, [**WRITES:** Sum := 0 + I, eventually put on line 4] and the number equals the integer, ahhh

**PLAN**: **Arithmetic Sum Variable**

Evidence: P3 - Running total assignment to variable Sum in loop.

---

**BUG: Arithmetic Sum set to current New Value Variable**
There will be no running sum operation, since the Arithmetic Sum, Sum, always gets the value of the current New Value Variable, I.

☞ ***Trace* on Arithmetic Sum Variable**
The subject is reasoning about the execution behavior of the Arithmetic Sum Variable. He recognizes that on the loops first iteration, Sum will be given the value 0 + I and then writes that.

---

127 **Interviewer:** Why don't you tell me what you are thinking

128 **Subject 13:** Well I'm thinking that [points to N := I, eventually changed to N := 1 and eventually put on line 5] should go after [points to Sum := 0 + I, eventually put on line 4] because the Sum is going to be zero plus the integer and then the Number is going be, ahh, Number equals 1.

129 **Interviewer:** Ah huh

130 **Subject 13:** [**WRITES:** [Crosses off N := I] And then Number equals one [**WRITES:** N := 1 on line 5]

---

**PLAN: Counter Variable**
Evidence: N7 - Counting to variable N done after operation.

---

**BUG: Counter needs to go after the Arithmetic Sum**
The increment of the Counter Variable, N, needs to be done after the update of the Arithmetic Sum Variable, Sum.

☞ ***PL Interpreted as NL* on Counter Variable**
In Natural Language counting always occurs after the operation being counted. Here the subject is applying that convention to Pascal.

---

131 **Interviewer:** OK

132 **Subject 13:** And then, and then Sum equals integer plus integer, [**WRITES:** Sum := I + I on line 6] and number equals 1. Ahhh

---

**PLAN: Arithmetic Sum Variable**
Evidence: P3 - Running total assignment to variable Sum in loop.

---

---

**BUG: Arithmetic Sum set to current New Value Variable**
There will be no running sum operation, since the Arithmetic Sum variable sum always gets twice the value of the current New Value Variable, x.

☞ ***Trace*** **on Arithmetic Sum Variable**
The subject is reasoning about the execution behavior of the Arithmetic Sum Variable. He recognizes that on the loops second iteration, sum will be given the value x + x and then writes that.

☞ ***Multi-Role Variable*** **on New Value Variable**
The subject allowing the New Value Variable to take two roles: its "sum so far" and its next value. Note that later[20] he will add "next" as a keyword in front of the second x.

---

133 **Interviewer:** What are you thinking now?

134 **Subject 13:** Number equals 2 [**WRITES:** n := 2 on line 7] and it would go on, it would repeat, that, if [the loop body] continues to repeat [sweeping motions] this [points to the 2 on line 7] will increase.

---

**PLAN: Counter Variable**
Evidence: P3 - Increment counter variable n inside the loop

---

**PLAN: Indefinite Loop**
Evidence: N1 - "Continues"
Evidence: N2 - "this will repeat until . . . "

---

**BUG: Says Counter Variable will increase, but it won't**
Says that the Counter Variable, n, will increase based on the n := 1 and n := 2 inside the loop.

☞ ***PL Interpreted as NL*** **on Pascal - repeat, Indefinite Loop**
The subject is expecting the loop to work like loops in Natural Language. There, it is common to specify a loop by giving one or two cases of the iteration and assuming that person reading will know how to generalize. Notice below (137) he says that sum := 0 + x (line 4) is the "first format of that" refering to the action performed for each value of the New Value Variable.

---

135 I'm assuming for the moment that this is sufficient input.

136 **Interviewer:** OK, "sufficient input"?

[20] Beyond the section excerpted here

137 **Subject 13:** Input to [pause] so that the computer will know that, for each [pause] for each integer entered, you add 1, you add the integer to the sum [points to `Sum := 0 + X` on line 4], and that this is the first format of that, zero plus integer, `N` equals 1, sum equals integer plus integer, number = 2, ["next" motion with hand] until [pause] [**WRITES:** `until`]

---

**PLAN: Indefinite Loop**
Evidence: N3 - "for each . . . "

---

**PLAN: Counter Variable**
Evidence: P3 - Increment Counter `N` inside the loop

---

**PLAN: Arithmetic Sum Variable**
Evidence: P3 - Update Arithmetic Sum, `Sum`, inside the loop

---

This concludes the example protocol analysis.