Alternatives to Construct-Based Programming Misconceptions

James C. Spohrer
Elliot Soloway

Department of Computer Science
Cognition and Programming Project
Yale University
New Haven, Connecticut 06520

ABSTRACT

In this paper, we investigate whether or not most novice programming bugs arise because students have misconceptions about the semantics of particular language constructs. Three high frequency bugs are examined in detail -- one that clearly arises from a construct-based misconception, one that does not, and one that is less cut and dry. Based on our empirical study of 101 bug types from three programming problems, we will argue that most bugs are not due to misconceptions about the semantics of language constructs.

1. Introduction: Motivation and Goals

Recently, there has been a great deal of interest in understanding bugs in programs (Anderson and Jeffries, 1985; Bonar and Soloway, 1985; Domingue, 1985; Johnson and Soloway, 1983; Pea, 1985; Sleeman, Putnam, Baxter and Kuspa, 1985; Spohrer, Soloway and Pope, 1985). Interest in program bugs usually derives from one of three perspectives:

- Engineers: Improve programmer productivity and program reliability.
- Educators: Improve novices' understanding of programming, and novices' ability to rapidly learn to generate correct, maintainable programs.
- Cognitive scientists: Improve our understanding of the acquistion and performance of complex problem solving skills and determine how misconceptions, problem solving strategies, and cognitive constraints on processing can lead to errors.

In this paper we are primarily concerned with the educator's perspective. Our data and theory of non-syntactic programming bugs can provide educational leverage for instructors in two ways:

- By providing a vocabulary for discussing programs and bugs.
- By giving a sense of why some of the bugs may have occurred.

One commonly used vocabulary for discussing programs and bugs is based on the syntactic constructs of the language. This construct-based perspective has fostered the view that:

- *Most bugs arise because students have misconceptions about language constructs.*

We will argue, based on our empirical study, that: Misconceptions about language constructs do not seem to be as widespread as typically believed.

2. Data Collection

The descriptions of bugs that we report in this paper are based on actual student generated programs. We augmented the operating system of the VAX 750 that the students were using and, with their permission, obtained a copy of each syntactically correct program they submitted for compilation. We call such data

This work was sponsored in part by the National Science Foundation, under NSF Grant DPE-8470014, and by the Army Research Institute, contract MDA903-85-K-0188.

 ACM 0-89791-180-6/86/0400 - 0183 $00.75

on-line protocols. We have collected this type of data from students for a number of introductory Pascal programming courses at various universities. The data reported in this paper were collected during the Spring 1984 semester at Yale University in an introductory Pascal programming course specifically designed for non-science, humanities oriented students.

```
Assignments   - Some Critical Issues

Cricket Chirp - Input Reals and Integers, Arith., Output
Electric Bill - Nested If-Then-Else, Single Boundary Guard
Reformatting  - Loop, Interval & Set Guard, Read Characters
Averaging     - Average, Counters, Running Accumulators
Fibonacci     - Nested Loops, Saving Previous Value In Loop
Rainfall      - Maximum, Average, Division-by-zero Guard
Bank          - Command Selection Loop
Tax           - Dispatch Loop, Procedures
License Plate - Functions, Random Numbers
Decoder       - Input from File
```

Figure 2-1: Programming assignments for CS110.

During this course, students were assigned ten programming problems (see Figure 2-1). Three of the programming problems were selected for further study:

- *Electric Bill Problem*: First problem requiring IF-THEN-ELSE constructs.
- *Reformatting Problem*: First problem requiring a WHILE-DO construct.
- *Tax Problem*: First problem requiring PROCEDURE constructs.

Sixty-one students' first syntactically correct programs for the three problems were selected for detailed analysis. We selected the first syntactically correct version because we are primarily interested in non-syntactic errors, and the first version contains more bugs than later versions (i.e., the novices had not yet debugged their programs). Of the 183 programs chosen in this manner, twenty-five programs were excluded from the analysis because the students had not tried to solve enough of the problem in their first attempt (e.g., several students' first syntactically correct program merely printed a message, which explained what their program was intended to do). In addition to these data, we have analyzed novice programs for other problems and from later versions; see Johnson et al. (1983), Joni and Soloway (1985), and Spohrer et al. (1985) for a more complete analysis.

3. Identifying Bugs in Terms of Goals and Plans

If a program is syntactically well-formed, then the program can be executed, but functional errors or *bugs* may prevent the program from behaving in an appropriate manner or from producing the correct output. Run-time error messages such as "attempted division by zero", "attempted access of uninitialized variable", or "array index out of bounds" indicate that a program has functional errors. In addition, deviant program behavior such as "never returning" (an infinite loop) or producing the wrong output on test input also indicate functional errors. To understand how we identify bugs in novice programs, we must first describe the *goal and plan* structure of a program.

We have been developing a theory of the knowledge that programmers use in generating and interpreting programs. Two key types of knowledge for which we have found empirical support are *programming goals and programming plans* (Soloway and Ehrlich, 1984; Soloway, Ehrlich, Bonar, and Greenspan, 1982; Bonar and Soloway, 1985). Programming plans are stereotypic sequences of code that accomplish some programming goal. In short, goals are *what* must be accomplished to solve a problem, and plans are *how* the goals can be achieved. For example, consider the summary specification of the Reformatting Problem in Figure 3-2; the four top-level goals that must be achieved are:

- *G:VDE*: This is the G:VALID-DATA-ENTRY goal of inputting experiment data, and in case of errors, giving the user a second chance to enter correct data.
- *G:CALC*: This is the G:CALCULATION goal of computing the elapsed time of the experiment in seconds.
- *G:OUTPUT*: Write out the reformatted data.
- *G:LOOP*: Process multiple records of information.

Each of these goals must be achieved using a particular programming plan. For example, consider the sample solution to the Reformatting Problem shown in Figure 3-1:

- P:INPUT-ALL/GUARD-ALL/RETRY-ALL is used to achieve G:VDE (lines 19-39).
- P:STANDARDIZE-WRAPAROUND-DIFFERENCE is used to achieve G:CALC (lines 42-46).
- P:OUTPUT-ALL is used to achieve G:OUTPUT (line 49).
- P:SEPARATE-LOOP-CONTROL-CHARACTER is used to achieve G:LOOP (lines 8-16, 52-59).

Many alternative plans exists for achieving these four goals (Spohrer et al, 1985), and certain plans have characteristic bugs (Spohrer, Soloway, Pope, 1985).

We use the goal and plan structure of a program to

1. Input experiment data; in case of typos, give the user a second chance.
2. Calculate the Elapsed_Time of the experiment based on start and end times.
3. Output reformatted experiment data with Elapsed_Time.
4. Process multiple experiments.

Figure 3-2: Summary of the Reformatting Problem.

```
01 PROGRAM Reformatting(INPUT,OUTPUT);
02 VAR Id,Start_Hour,Start_Minute,Start_Second,End_Hour,End_Minute,
03     End_Second,Elapsed_Time,Start_Time,End_Time:INTEGER;
04     Problem_Type,Accuracy,Sentinel,Whitespace1,Whitespace2:CHAR;
05
06 BEGIN
07 +-------[ P:SEPARATE-LOOP-CONTROL-CHARACTER ]----------+
08 |WRITELN('Type "y" to process data ("n" to stop)');    |
09 |READLN(Sentinel);                                     |
10 |IF ((Sentinel <> 'y') and (Sentinel <> 'n'))          |
11 |   THEN BEGIN                                         |
12 |        WRITELN('Type "y" or "n"');                   |-- G:LOOP
13 |        READLN(Sentinel);                             |
14 |        END;                                          |
15 |WHILE (Sentinel = 'y') DO                             |
16 | BEGIN                                                |
17 +------------------------------------------------------+
18 | +--------[ P:INPUT-ALL/GUARD-ALL/RETRY-ALL ]-------+
19 | |WRITELN('Enter: Id Type Start&End(h m s) Acc.');   |
20 | |READLN(Id, Whitespace1, Problem_Type,              |
21 | |       Start_Hour, Start_Minute, Start_Second,     |
22 | |       End_Hour, End_Minute, End_Second,           |
23 | |       Whitespace2, Accuracy);                     |
24 | |IF ((Id < 0)  OR  ((Problem_Type <> 'a') AND       |
25 | | (Problem_Type<>'b' AND(Problem_Type<>'c')) OR     |
26 | | ((Start_Hour   < 1) OR (Start_Hour   > 12)) OR    |
27 | | ((Start_Minute < 0) OR (Start_Minute > 59)) OR    |
28 | | ((Start_Second < 0) OR (Start_Second > 59)) OR    |-- G:VDE
29 | | ((End_Hour   < 1) OR (End_Hour   > 12)) OR        |
30 | | ((End_Minute < 0) OR (End_Minute > 59)) OR        |
31 | | ((End_Second < 0) OR (End_Second > 59)) OR        |
32 | | ((Accuracy <> '+') AND (Accuracy <> '-')))        |
33 | |    THEN BEGIN                                     |
34 | |         WRITELN('BAD DATA: Try again.');          |
35 | |         READLN(Id, Whitespace1, Problem_Type,     |
36 | |          Start_Hour,Start_Minute,Start_Second,    |
37 | |          End_Hour, End_Minute, End_Second,        |
38 | |          Whitespace2, Accuracy);                  |
39 | |         END;                                      |
40 | +---------------------------------------------------+
41 | +--------[ P:STANDARDIZE-WRAPAROUND-DIFFERENCE ]----------+
42 | |Start_Time:=Start_Hour*3600+Start_Minute*60+Start_Second;|
43 | |End_Time:=End_Hour*3600+End_Minute*60+End_Second;        |
44 | |IF (Start_Time >= End_Time)          +----------------+
45 | |   THEN End_Time := End_Time + 12*3600; |
46 | |Elapsed_Time := End_Time - Start_Time;  |-- G:CALC
47 | +---------------------------------------+
48 | +-------------[ P:OUTPUT-ALL ]-------------------------+
49 | |WRITELN('New=',Id,Problem_Type,Elapsed_Time,Accuracy);|
50 | +------------------------------------------------------+
51 +----------[ P:SEPARATE-LOOP-CONTROL-CHARACTER ]-----+  |
52 |  WRITELN('Type "y" to process data ("n" to stop)');|  G:OUTPUT
53 |  READLN(Sentinel);                                |
54 |  IF ((Sentinel <> 'y') and (Sentinel <> 'n'))     |
55 |     THEN BEGIN                                    |
56 |          WRITELN('Type "y" or "n"');              |-- G:LOOP
57 |          READLN(Sentinel);                        |
58 |          END;                                     |
59 |  END;                                             |
60 +---------------------------------------------------+
61 END.
```

Figure 3-1: A Reformatting Problem solution.

identify bugs. Our position is: *Given a goal and plan vocabulary for describing the structure of a program, we can identify bugs in novice programs as the differences between one of the correct plans for achieving a goal and the observed plan for achieving a goal.* For example, Figure 3-3 shows an incorrect implementation of the P:INPUT-ALL/GUARD-ALL/RETRY-ALL. Because we have the correct plan (left side of Figure 3-3), we can compare it to the observed implementation used by the novices and note the differences. The buggy plan is missing certain variables that are contained in the correct plan. The problem with the buggy plan is that the whitespace (that is used to separate the input values) will be read in instead of the intended character values. In line 3 of the buggy plan, the Problem_Type variable will read the space just before the character a, which is the intended input for Problem_Type. However, note that in line 3 of the correct plan an extra variable, Whitespace1, has been added to capture the whitespace character before the character a. The problem also exist in line 18, when the G:RETRY subgoal is achieved, and in line 6 and 21 for the Accuracy variable. All together there are four plan differences, as indicated by the arrows in Figure 3-3. Because all four of these differences result from the same underlying misconception about reading characters embedded in other input data, we identify the following bug:

- G:VDE[P:INPUT-ALL/GUARD-ALL/RETRY-ALL]/G:TYPE&G:ACCURACY
  :: MALFORMED G:INPUT&G:RETRY

The bug name serves to locate where the bug occurred: In the top-level G:VDE goal, when the P:INPUT-ALL/GUARD-ALL/RETRY-ALL plan was employed, the G:TYPE&G:ACCURACY subgoals were not successfully achieved due to malformed plans to achieve the G:INPUT&G:RETRY subgoals.

4. Analysis of the Data

Figure 4-1 provides an overview of our analysis of the first syntatically correct programs for the three problems.

Type of Statistic	Elec.B.	Reform.	Tax
Number of Subjects Analyzed	55	46	57
Average Size of Program (Lines)	28	73	69
Number of Observed Bugs(TOKENS)	85	140	59
Number of Unique Bugs (TYPES)	28	46	27

Figure 4-1: Statistical overview of the three problems.

Programs that solved the Reformatting Problem were the longest (on average 73 lines of Pascal), and also had the most types of bugs (46 bug types). An important distinction to be aware of when counting bugs is the type-token distinction. A *bug token* is an instance of a bug in some program. A *bug type* is a name for a group of identical bug tokens. The total number of bug tokens found in all the novice programs was 284 (85+140+59), while the total number of bug types was 101 (28+46+27).

SAMPLE INPUT: *23 a 10 45 30 10 57 02 +* ;Note separators

```
01 +-[ CORRECT  P:INPUT-ALL/GUARD-ALL/RETRY-ALL ]-------------+      +------------[ OBSERVED BUGGY PLAN FOR G:VDE ]-------------+
02 |WRITELN('Enter: Id Type Start(h m s) End(h m s) Accuracy');|      |WRITELN('Enter: Id Type Start(h m s) End(h m s) Accuracy');|
03 |READLN(Id, Whitespace1, Problem_Type, ------------------------->|READLN(Id, Problem_Type,                                  |
04 |       Start_Hour, Start_Minute, Start_Second,               |      |       Start_Hour, Start_Minute, Start_Second,            |
05 |       End_Hour, End_Minute, End_Second,                     |      |       End_Hour, End_Minute, End_Second,                  |
06 |       Whitespace2, Accuracy); ------------------------------------->  Accuracy);                                          |
07 |IF ((Id < 0)  OR  ((Problem_Type <> 's') AND                 |      |IF ((Id < 0)  OR  ((Problem_Type <> 's') AND              |
08 |    (Problem_Type <> 'b) AND (Problem_Type <> 'c)) OR        |      |    (Problem_Type <> 'b') AND (Problem_Type <> 'c')) OR   |
09 |    ((Start_Hour   < 1) OR (Start_Hour   > 12)) OR           |      |    ((Start_Hour   < 1) OR (Start_Hour   > 12)) OR        |
10 |    ((Start_Minute < 0) OR (Start_Minute > 59)) OR           |      |    ((Start_Minute < 0) OR (Start_Minute > 59)) OR        |
11 |    ((Start_Second < 0) OR (Start_Second > 59)) OR           |      |    ((Start_Second < 0) OR (Start_Second > 59)) OR        |
12 |    ((End_Hour   < 1) OR (End_Hour   > 12)) OR               |      |    ((End_Hour     < 1) OR (Start_Hour   > 12)) OR        |
13 |    ((End_Minute < 0) OR (End_Minute > 59)) OR               |      |    ((End_Minute   < 0) OR (Start_Minute > 59)) OR        |
14 |    ((End_Second < 0) OR (End_Second > 59)) OR               |      |    ((End_Second   < 1) OR (Start_Second > 59)) OR        |
15 |    ((Accuracy <> '+') AND (Accuracy <> '-')))               |      |    ((Accuracy <> '+') AND (Accuracy <> '-')))            |
16 |    THEN BEGIN                                               |      |   THEN BEGIN                                             |
17 |         WRITELN('BAD DATA: Try again.');                    |      |        WRITELN('BAD DATA: Try again.');                  |
18 |         READLN(Id, Whitespace1, Problem_Type, ------------------------->| READLN(Id, Problem_Type,                          |
19 |                Start_Hour, Start_Minute, Start_Second,      |      |                Start_Hour, Start_Minute, Start_Second,   |
20 |                End_Hour, End_Minute, End_Second,            |      |                End_Hour, End_Minute, End_Second,         |
21 |                Whitespace2, Accuracy); ------------------------------------->  Accuracy);                                 |
22 |         END;                                                |      |         END;                                             |
23 +-------------------------------------------------------------+      +----------------------------------------------------------+
```

Figure 3-3: Identifying bugs as plan differences.

4.1. Semantics of Constructs: The Dominant Source of Novice Bugs?

In most introductory textbooks each successive section covers a new language construct. Instructors teach the course by describing successive language constructs and then showing how to use the construct in a program. Because of this emphasis on language constructs, one might suspect that:

- *Most bugs arise because the novice does not fully understand the semantics of particular programming language constructs.*

Certainly, extensive evidence does exist that novices have many misconceptions about constructs. For instance, Bayman & Mayer (1983) showed that novices possess a wide range of misconceptions concerning individual statements or constructs from the programming language they had learned. These misconceptions reflected differences between novices' mental models of individual BASIC statements and an expert's framework for describing the function of the statements (Mayer, 1979). However, these studies were designed to test novices ability to interpret single statements after taking a three-session self-instruction course. To evaluate the notion that most bugs arise from misconceptions about a construct, more representative data is required. Our data which was collected in a more natural setting during a semester-long introductory programming course can be analyzed to provide a more complete picture of novice programmers and the types of bugs they make.

In order to evaluate the commonsense notion that most novice programming bugs are due to a misunderstanding of the semantics of some particular language construct, we must try to identify the underlying source of each bug. In effect, we are trying to guess what the student was thinking about that lead to the bug. We term explanations of the origins of bugs *plausible accounts*. For example, the bug described in section 3 (see Figure 3-3) might have arisen because the student thought that since whitespace between numeric input is ignored, the whitespace between character input would be ignored as well.

In what follows, we will examine plausible accounts for three high frequency Reformatting Problem bugs. One of the bugs is clearly *not* the result of a misconception about programming constructs, one of the bugs *does* result from a misunderstanding of a programming construct, and one of the bugs *may or may not* result from misunderstanding a construct. We have classified the plausible accounts for all 101 bug types into one of three categories (i.e., NO, YES, MAYBE). After describing three of these plausible accounts in detail (one illustrating each category), we will present the proportion of bug types that fell into each of the three categories in order to evaluate whether or not most bugs do arise from misconceptions about programming language constructs.

One final point about plausible accounts should be made: Developing and evaluating the veracity of such

plausible accounts is a difficult activity. We have drawn on the work of Bonar and Soloway (1985), Sleeman et al. (1985), and Pea (1985), who interviewed students as they were attempting to write programs, in developing and evaluting the plausible accounts presented in this section. The verbal reports they gleaned often shed light on why the students were writing the programs in the way they were. Nonetheless, extreme care must be exercised in using such verbal reports to both develop and evaluate plausible accounts. For example, simply asking a student what he/she was thinking about as they were creating the buggy program (or after the fact), may or may not provide particularly useful data; oftentimes when someone is confused, they do not know why they are confused, and their verbal report may not be accurate. We have also drawn on our not inconsiderable experience in analyzing and tutoring students in programming (Johnson et al., 1983; Littman, Pinto and Soloway, 1985; Spohrer et al., 1985) in developing and evaluating the plausible accounts in this section. Thus, while these plausible accounts must be subjected to further scientific inquiry, they were by no means pulled out of thin air; we feel that they have considerable grounding in psychological reality. In what follows, then, we will use these plausible accounts to explore the question -- are most bugs the result of misconceptions about language constructs?

NO: Boundary Problem

In Figure 4-2 the most frequent Reformatting Problem bug is shown. The condition that is used to guard against invalid times is not quite correct. When the value of the hour variable is 12, then the input will be considered invalid, even though it is in fact a legal value. If either the upper bound for hours was changed from 12 to 13 (see line 4 in Figure 4-2) or the relational operator was changed from >= to > (and the constant was left 12), then the code would be correct.

```
G:VDE/G:START-HOUR/G:GUARD :: MALFORMED G:COND (12 for 13)
01 WRITELN('Enter start hour, minutes, and seconds');
02 READLN(Start_Hour,Start_Minute,Start_Second);
03+-------------------------------------------------+
04|IF((Start_Hour   <=  0) OR (Start_Hour   >= 12) OR|
05|   (Start_Minute <= -1) OR (Start_Minute >= 60) OR|
06|   (Start_Second <= -1) OR (Start_Second >= 60))  |
07+-------------------------------------------------+
08   THEN BEGIN
09        WRITELN('Error: Rtype hour minute second');
10        READLN(Start_Hour,Start_Minute,Start_Second);
11        END;
```

Figure 4-2: An off-by-one bug.

The students who generated this bug probably understood the semantics of the relational operators. In Figure 4-2 all of the other relational operators are correct; only the one for the upper hour boundary is inappropriate. This *off-by-one bug* may have arisen because of an inconsistency between the various times units. Hours range from 1 to 12, while minutes and seconds range from 0 to 59. Note that the lowest value in each range is different, so while 12 is both the number of allowed values and the highest value for hours, 60 is the number of allowed values and 59 is the highest value for minutes and seconds. The number of allowed values in a range (12 and 60) are important constants because they are used in conversion rules which relate quantities measured in one time unit to quantities measured in another time unit. The problem is that if the novice wants to use these two constants in the upper bound guard, then a different relational operator is needed for hours than was used for minutes and seconds. So the novices might generate this bug because they tend to use seemingly related constants (i.e., 12 and 60, rather than 13 and 60), while at the same time trying to use the same relational operator to make the terms more parallel. Alternatively, the novices might be confusing the number of values in a range with the maximum value in a range; the former is needed to convert between units and the latter is needed to construct a valid data guard.

In sum, the source of the bug seems to be a problem that novices experience while reasoning about a group of related boundary conditions. The novices may want the constants to be semantically related and at the same time they are trying to exploit certain forms of parallelism in their code. However, it seems unlikely that they misunderstoodd relational operators -- misusing only the relational operator for the upper hour boundary. For this reason, we have classified this bug as most likely not arising from a misunderstanding of the semantics of any language construct.

YES: Type Inconsistency Problem

Figure 4-3 shows the bug described in section 3 in Figure 3-3. Apparently, some novices did not understand that READLN can read a space, and not ignore it as merely a separator. Possibly, some novices thought that READLN ignored all whitespace when parsing an input line and assigning values to a sequence of variables. When a READLN is used to input a sequence of numeric values, this is certainly the case. For this reason, we have classified this bug as probably

INPUT:
23 a 10 45 30 10 57 02 +

Buggy Code to achieve G:INPUT:

```
G:VDE/G:TYPE&G:ACCURACY :: MALFORMED INPUT
01 WRITELN('Enter Id,Type,Start&End(h m s),Accuracy');
02+-------------------------------------------+
03|READLN(Id,                                 |
04|       Problem_Type,                       |
05|       Start_Hour, Start_Minute, Start_Second, |
06|       End_Hour, End_Minute, End_Second,   |
07|       Accuracy);                          |
08+-------------------------------------------+
```

Corrected Code for G:INPUT: (note whitespace variables)

```
09 WRITELN('Enter Id,Type,Start&End(h m s),Accuracy');
10+-------------------------------------------+
11|READLN(Id,                                 |
12|       Whitespace1, Problem_Type,          |
13|       Start_Hour, Start_Minute, Start_Second, |
14|       End_Hour, End_Minute, End_Second,   |
15|       Whitespace2, Accuracy);             |
16+-------------------------------------------+
```

Figure 4-3: A space-as-separator-not-character bug.

resulting from a misunderstanding of the semantics of the READLN language construct.

The novices had previously used the READLN construct successfully in other problems on different types of input data. In predicting where bugs will occur, knowing that a particular construct has been used successfully before does not imply that it will be used correctly in future problems. The key issue is not the particular construct that was being used, but the programming plan in which the construct played a role. In the case of the whitespace bug in Figure 4-3, the new plan involves reading character values on the same line with numeric values. Previously, either character values were read alone on a line or just numeric values were read on a line.

MAYBE: Negation and Whole-Part Problem

Figure 4-4 shows an *OR-for-AND* bug that occurred in novices' Reformatting programs. The boolean connective that is used in the guard (lines 4 and 5) should be AND, rather than OR. The bug occurred during the composition of a P:SET-MEMBERSHIP-EXCLUSION plan for the Problem_Type variable. The data item in question could take on one of a small set of character values that were members of a set of valid inputs. For instance, the valid values for a problem type are 'a',

```
G:VDE/G:TYPE/G:GUARD :: MALFORMED G:COND (OR for AND)
01  WRITELN('Enter problem type');
02  READLN(Problem_Type);
03+-------------------------------+
04| IF ((Problem_Type) <> 'a') OR  |
05|     (Problem_Type) <> 'b') OR  |
06|     (Problem_Type) <> 'c'))    |
07+-------------------------------+
08      THEN BEGIN
09           WRITELN('Type "a", "b", or "c"');
10           READLN(Problem_Type);
11           END;
```

Figure 4-4: An OR-for-AND bug.

'b', or 'c'. The data guard was intended to determine if the input value was outside the valid set, and if so give the user a second chance to enter the data correctly.

While the OR-for-AND bug seems to indicate that novices did not understand the semantics of OR, two pieces of evidence contradict this conclusion. First, novices usually used the OR boolean connective correctly in the P:INTERVAL-EXCLUSION plans required to guard the hour, minute, and second data items in the Reformatting Problem (as shown in Figure 3-1 lines 26-31). Secondly, some of the novices used a P:SET-MEMBERSHIP-INCLUSION plan to guard the data items as shown below:

```
01  IF ((Problem_Type = 'a') OR
02      (Problem_Type = 'b') OR
03      (Problem_Type = 'c'))
04  THEN ...g:data_ok...
05  ELSE ...g:retry...
```

The novices who chose the P:SET-MEMEBERSHIP-INCLUSION plan used the OR boolean connective correctly. In order to explain why we think that many of the novices may actually understand the semantics of OR, we introduce the following two terms:

- *good-thing:* a term in a boolean expression, which either partially or totally implies that the value of the data item is valid.
- *bad-thing:* a term in a boolean expression, which either partially or totally implies that the value of the data item is invalid.

From our perspective, it appears that the novices may have the following notion concerning the use of OR: If each term corresponds to a whole *good-thing* or a whole *bad-thing*, then connect the terms with the boolean connective OR. In the case of P:INTERVAL-EXCLUSION (see Figure 3-1 lines 26-31), since (Start_Hour > 12) is a whole *bad-thing*, OR should be used. In the case of P:SET-MEMBERSHIP-INCLUSION,

since (Problem_Type = 'a') is a whole *good-thing*, OR should be used. If the novices mistakenly reasoned that since (Problem_Type = 'a') is a whole *good-thing*, by simple negation (Problem_Type <> 'a') is a whole *bad-thing*, then OR would have been the appropriate connective to use. In this case, the student's problem would not be in misunderstanding when to use the OR boolean connective, but in thinking that something was a whole *bad-thing* when in fact it was just one piece of a *bad-thing*. (Problem_Type <> 'a') is just a part of a *bad-thing* because, even if the problem type is not equal to 'a', it still may be valid because it could still be equal to 'b' or 'c' instead. Therefore, what on the surface appears to be a misunderstanding of the semantics of a construct may in fact be the result of some other error.

We have classified this bug as maybe arising from a misunderstanding of the semantics of the language. While we feel some students did in fact understand the semantics of OR, but had the misconception that Problem_Type <> 'a' was a whole *bad-thing* due to faulty reasoning about the effects of negating a term, some students may not have understood the semantics of OR. Those students who did not understand the semantics of OR may have always used the boolean connective OR to connect terms in a boolean expression, or have assumed, based on their linguistic intuitions, that if they could understand the intent of the boolean expression, then so could the computer.

4.2. All Bugs

While the previous section looked at just 3 out of the 101 bug types, in this section we ask the same question for all 101 bug types (see Figure 4-1; 101=28+46+27). With this larger sample, we will be in a better position to evaluate the hypothesis: Most bugs arise from a misunderstanding of the semantics of a particular programming language construct. Figure 4-5 shows the proportion of all bugs in each of the three categories:

- NO: Probably did not result from misunderstanding a construct's semantics.
- MAYBE: Might have resulted from misunderstanding a construct's semantics.
- YES: Probably did result from misunderstanding a construct' semantics.

Three judges, trained for two months in generating plausible accounts for a separate programming problem, independently classified the 101 bugs into the three categories with over 90% agreement. The results of the judge who had produced the most conservative classification (i.e., most MAYBE and YES decisions) are presented in Figure 4-5.

Problem		NO	MAYBE	YES
BILL	(N = 28)	.68	.32	.00
REF	(N = 46)	.43	.46	.11
TAX	(N = 27)	.48	.37	.15
OVERALL	(N = 101)	.52	.39	.09

Figure 4-5: Proportion of bugs in the three categories.

One of the most useful statistics for evaluating proportion data (samples divided among categories) are confidence intervals (Hays, 1965). Given an estimate of the proportion of individuals in a population having a certain characteristic, we can qualify this estimate to indicate the general magnitude of the sampling error. In general, the larger the population sampled the smaller the sampling error, and hence the closer the estimated proportion is to the true proportion. Given the sample size (N) and the estimated proportion, we can estimate a 99% confidence interval such that the true proportion will be contained in this interval 99 times out of 100. Using this very conservative confidence interval, we will look at the data in Figure 4-5 in three ways:

- *Just the clear-cut cases:* Ignoring the bugs that fell in the MAYBE category for a moment, the NO category (non-construct-based plausible accounts) has a 99% confidence interval on the proportion .52 that ranges between .39 and .65, and the YES category (construct-based plausible accounts) has a 99% confidence interval on the proportion .09 that ranges between .03 and .20. For the clear-cut cases then, non-construct-based problems underlie a far greater proportion of the bugs (at least 2-to-1 as shown in Figure 4-6A).
- *Combine the MAYBE and YES categories:* Although it is extremely unlikely that further research will reveal that all the bugs in the MAYBE category are actually due to misconceptions about constructs, the most conservative position we can adopt in evaluating the hypothesis that most bugs do result from misconception about constructs is to combine all bugs in the MAYBE category with those in the YES category. The 99% confidence interval on the proportion .48 for the combined MAYBE-YES category ranges between .36 and .63. Recall that for the NO category the confidence interval ranges between .39 and .65. While the NO category has a slightly higher range, these intervals

overlap on over 90% of their ranges (see Figure 4-6B). So even by making the most favorable assumption for the hypothesis that most bugs arise from misconceptions about constructs, the hypothesis is not supported.

- *Dividing the MAYBE category between the YES and NO categories*: While further research is required to assign bugs in the MAYBE category to either the YES or NO category for certain, we can provide the reader with a graph of the resulting 99% confidence intervals for the entire range of possible outcomes from splitting the MAYBE category. Figure 4-7 summarizes the results of all possible ways of dividing up the MAYBE category between the YES and NO categories. To understand the graph consider the following three cases:
  - The left-hand-side of the x-axis corresponds to all the MAYBEs being assigned to the YES category (100% MAYBEs to YES, and 0% MAYBEs to NO). This situation is illustrated in Figure 4-6B as well.
  - The right-hand-side of the x-axis corresponds to all the MAYBEs being assigned to the NO category (0% MAYBEs to YES, and 100% MAYBEs to NO)
  - The intermediate points on the x-axis indicate points between these two extremes. For example, if further analysis of the MAYBE bugs were to show that 25% of those bugs were actually accounted for by non-construct-based plausible accounts (75% accounted for by construct-based plausible accounts), then the confidence bands would indicate that the non-construct-based plausible accounts are almost certainly going to be more useful for explaining bugs than the construct-based plausible accounts.

On the basis of the observed data, we conclude that there is no reason to believe that construct-based plausible accounts are any more useful in explaining students' bugs than non-construct-based plausible accounts. In fact, our data suggest just the opposite view from the standard programming folk wisdom – most bugs seem to arise from problems other than misunderstanding the semantics of language constructs.

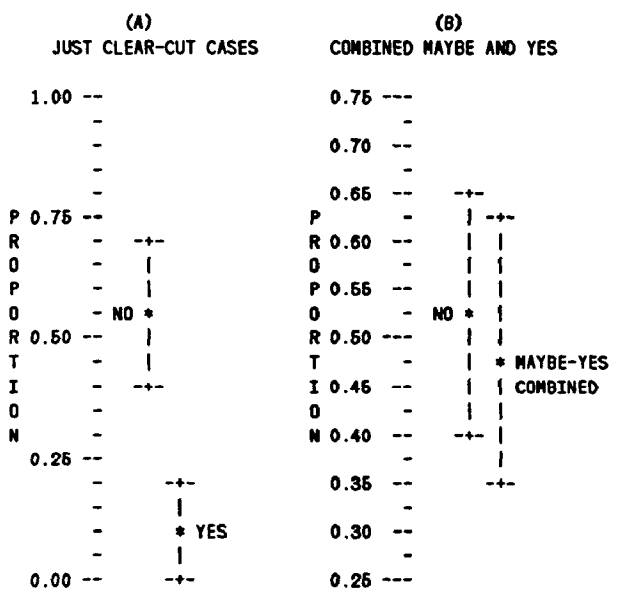

Figure 4-6: 99% confidence intervals on categories.

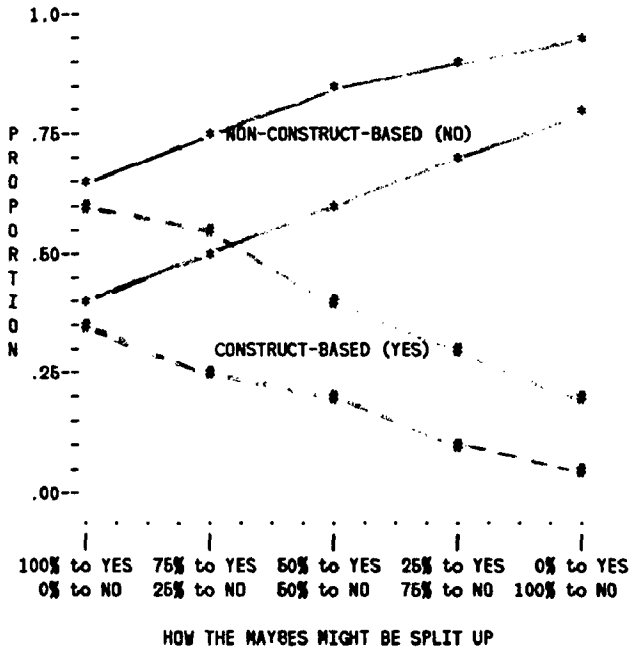

Figure 4-7: Dividing MAYBE between YES and NO.

5. Concluding Remarks

In short, our data does not support one commonsense notion about bugs in novice programs:

- *NOT SUPPORTED: Most bugs result because novices misunderstand the semantics of some particular programming language construct.*

Two of the common non-construct-based problems that novices have are:

- *Boundary Problem*: Novices have difficulties deciding what the appropriate boundary points are. Consider the off-by-one bug.
- *Negation and Whole-Part Problem*: Novices mistakenly infer that the negation of a whole *good-thing* is a whole *bad-thing*. Consider the OR-for-AND bug.

In addition to these problems, our evaluation of plausible accounts for all 101 bug types identified two other common non-construct-based problems:

- *Interpretation Problem*: When novices read a programming assignment, they do not always infer the correct interpretation from the specifications.
- *Composition Problem*: Novices may not detect negative interactions between sections of code that are locally correct, but globally incorrect. For example, the code to perform the output may be correct, but in the wrong place in the program.

The main implication these findings have for educators is that: Simply making the semantics of constructs clearer will not address many of the problems novices are having.

REFERENCES

Anderson, J.R. and Jeffries, R. (1985) Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction, 1.* (In press).

Bayman, P. and Mayer, R.E. (1983, September) A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM, 26,* 677-679.

Bonar, J. and Soloway, E. (1985) Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction, 1.* (In press).

Domingue, J. (1985) *Towards an automated programming advisor.* (Technical Report No. 16). Milton Keynes, MK7 6AA, England: Human Cognition Research Laboratory.

Hays, William L. (1965) *Statistics for psychologists.* New York, NY: Holt, Rinehart and Winston. Pg. 290.

Johnson, L., and Soloway, E. (1983, August) *PROUST: Knowledge-based program understanding.* (Technical Report 285). New Haven, CT: Department of Computer Science, Yale University.

Johnson, L., Soloway, E., Cutler, B. and Draper, S. (1983, October) *BUG CATALOGUE: I.* (Technical Report 286). New Haven, CT: Dept. of Computer Science, Yale University.

Joni, S. and Soloway, E. (1985) But My Program Runs! *Journal of Educational Computing Research,* Fall. (In press).

Littman, D., Pinto, J., Soloway, E. (1985) Observations on tutorial expertise. *Proceedings of Conference on Expert Systems in Government.* MacClean, VA.

Mayer, R.E. (1979, November) A psychology of learning BASIC. *Communications of the ACM, 22,* 589-593.

Pea, R.D., (1985) *Language-independent conceptual "bugs" in novice programming.* (Manuscript). New York, NY: Bank Street School.

Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. (1982) What do novices know about programming? In B. Shneiderman and A. Badre (Eds.), *Directions in Human-Computer Interactions,* Norwood, NJ: Ablex. 27-54.

Soloway, E., Ehrlich, K. (1984) Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, 5,* 595-609.

Sleeman, D., Putnam, R.T., Baxter, J.A., Kuspa, L.K. (1985) *Pascal and high-school students: A study of misconceptions.* (Manuscript). Stanford, CA: School of Education, Stanford University.

Spohrer, J.C., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., Johnson, L., and Soloway, E. (1985, May) *BUG CATALOGUE: II,III,IV.* (Technical Report 386). New Haven, CT: Dept. of Computer Science, Yale University.

Spohrer, J.C., Soloway, E., and Pope, E. (1985) A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction, 1.* (In press).