In J. Hiebert (Ed.) Conceptual and Procedural Knowledge: The Case of Mathematics. Hillsdale, NJ: Erlbaum. 1986

6 Arithmetic Procedures are Induced from Examples

Kurt VanLehn
Carnegie-Mellon University

Suppose one asked some concerned adults, for example, parents, how multi-column subtraction is learned in school. Their explanation would probably run something like this: The teacher tells the students how to perform the algorithm, then sets them to solving practice exercises. Perhaps the practice causes some students to realize that they hadn't quite understood what the teacher meant. Or perhaps the teacher notices that certain students are following the wrong procedure. In either case, the teacher helps the students by telling them in more detail about the algorithm, probably by adapting the explanation to the particular exercises that the students are working on. *Telling* dominates both the initial instruction and the subsequent teacher-student interactions, according to this folk model of arithmetic acquisition. Winston (1978) dubbed this model "learning by being told."

The evidence to be presented below suggests that learning-by-being-told is an inaccurate model of the kind of arithmetic learning that actually occurs in classrooms. Rather, arithmetic is learned by induction—the generalization and integration of examples. An "example" of a procedure is an execution of it. When teachers work a subtraction exercise on the blackboard, their writing actions constitute an example of the subtraction procedure. Although some inductive learning may occur while passively observing the teacher work problems, most inductive learning probably occurs in the midst of problem solving. For instance, a student may try to solve a practice exercise, get stuck, and seek help from the textbook, the teacher, or a classmate. With this help, the student determines what writing actions to perform next and thereby continues toward a solution of the exercise. Learning occurs when the student generalizes these actions and incorporates them into his or her procedure. The writing actions are an example of a

subprocedure that the student needs to learn. The student's generalization of that example yields a subprocedure, although it may not be the correct subprocedure. This hypothetical incident illustrates how inductive learning can occur in the midst of problem solving. The defining characteristic of induction of arithmetic procedure is that learning is based on *generalization of writing actions*, regardless of whether those actions are acquired by actively soliciting help or by passively observing an exercise being solved.

The folk model of arithmetic learning holds that verbal explanations provide information from which students learn procedures. Although verbal explanations certainly dominate the instruction, the evidence presented below suggests that the examples that inevitably accompany such explanations are doing the pedagogical work. However, the examples don't do all the work. The verbal explanations are crucial for indicating to the students the particular kind of induction to perform on the examples. The verbal explanations indicate what aspects of the examples to generalize and how to integrate them. That is, verbal explanations have an indirect effect. They function as if they tell the student *how to induce the arithmetic procedure* despite the fact that their literal content is about how to perform the procedure. This view of arithmetic learning will be called the induction hypothesis.

Clearly, the induction hypothesis would be false if it were tested in a fine-grained way. During the hundreds of classroom hours that a student spends learning the multicolumn arithmetic algorithms, there are, no doubt, many episodes where the student suddenly grasps a new aspect of an algorithm, and yet there is no example in sight. Because examples are a prerequisite for induction, such episodes would be outright contradictions of the induction hypothesis, if the hypothesis were to be taken as a statement about the second-by-second learning process. Such a fine-grained interpretation is not the intended one. An explanation of the intended interpretation requires that a little background on the research project be presented first.

The research project began with the "buggy" studies of Brown and Burton (1978). It is well known that arithmetic students make a large variety of systematic errors (Buswell, 1926; Brueckner, 1930; Brownell, 1941; Roberts, 1968; Lankford, 1972; Cox, 1975; Ashlock, 1976). Brown and Burton used the metaphor of bugs in computer programs in developing a precise, detailed formalism for describing systematic errors. The basic idea is that a student's errors can be accurately reproduced by taking a formal representation of the correct algorithm and making one or more small perturbations to it, such as deleting a rule. The perturbations are called bugs. A systematic error is represented by a correct procedure for the skill plus a list of one or more bugs. Bugs describe systematic errors with unprecedented precision. If a student makes no unintentional mistakes (e.g., $7 - 2 = 4$), then the student's answers will exactly match the buggy algorithm's answers, digit for digit.

Burton (1982) developed an automated data analysis program, called Debuggy. Using it, data from thousands of students learning subtraction were analyzed, and 76 different kinds of bugs were observed (VanLehn, 1982). Similar studies discovered 68 bugs in addition of fractions (Shaw et al., 1982), several dozen bugs in linear equation solving (Sleeman, 1984), and 57 bugs in addition and subtraction of signed numbers (Tatsuoka & Baillie, 1982).

It is important to stress that bugs are only a notation for systematic errors and not an explanation. The connotations of "bugs" in the computer-programming sense do not necessarily apply. In particular, bugs in human procedures are unstable. They appear and disappear over short periods of time, often with no intervening instruction, and sometimes even in the middle of a testing session (VanLehn, 1982). Often, one bug is replaced by another, a phenomenon called bug migration.

Collecting bugs leads inevitably to wondering why those bugs exist. There are an infinite number of possible bugs; why do students only acquire certain of these? One way to answer such questions is to develop a generative theory of bugs. Such a theory should generate (predict) exactly which bugs will occur and which bugs won't. The way that it generates a bug constitutes an explanation for the bugs' existence.

Repair theory (Brown & VanLehn, 1980) was our first version of a generative theory of bugs. The basic idea of repair theory is that students don't simply halt when they reach an impasse while following a procedure, as a computer would. Rather, they apply certain metalevel problem solving operations, called repairs, that change their interpretation of the procedure in such a way that they can continue. As an illustration, suppose that a student who has not yet learned about borrowing from zero encounters the problem $305 - 109$. When the student trys to decrement the top digit in the tens column, as his incomplete procedure says he should, he finds that it is a zero and can't be decremented. He is at an impasse (see a below).

a. $\begin{array}{r} 30^{1}5 \\ -10\,9 \\ \hline \end{array}$ b. $\begin{array}{r} 30^{1}5 \\ -10\,9 \\ \hline 20\,6 \end{array}$ c. $\begin{array}{r} {}^{2} \\ \not{3}0^{1}5 \\ -10\,9 \\ \hline 10\,6 \end{array}$

(The small numbers represent the student's scratch marks.) Several repairs could potentially be applied here. A simple one is just to skip an action when the preconditions are violated. In this case, the repair would result in omitting the decrement half of borrowing (see *b* above). If the student does this on every problem that requires borrowing from zero, then he or she will appear to have a systematic error, a bug called Stops-Borrow-At-Zero. (The appendix lists the observed subtraction bugs, with a short description of each.) If the student chooses a different repair, such as relocating the stuck action, then a different bug would be generated. Problem *c* above exhibits Borrow-Across-Zero, a bug where the decrement has been moved leftward. The impasse/repair mechanism can explain many bugs as coming from the same underlying incomplete procedure. Such underlying procedures are called *core procedures*.

Repair theory can also explain bug migrations. Suppose a student has the same core procedure throughout a testing session, but instead of repairing every occurrence of an impasse with the same repair, he or she makes different repairs. This would make it appear as if the student were exhibiting different bugs on different problems, or maybe even on different columns within the same problem. For instance, if the hypothetical student mentioned above chooses the first repair for some impasses and the second repair for others, then it will appear that there is a bug migration between Stops-Borrow-At-Zero and Borrow-Across-Zero. Even though the core procedure is stable, there is instability in what appears on the surface to be the student's procedure. The *surface procedure*, which exists only in the eye of the observer and not in the mind of the student, changes from one buggy procedure to another.

If one stipulates just the right set of core procedures, then repair theory can generate a large set of observed bugs. The set of bugs is larger than the set of core procedures, so repair theory is not a vaccuous theory. Originally, the set of core procedures was discovered by trial and error. In a sense, we "observed" those core procedures in the subject population. Something is needed to explain why exactly that set of core procedures is observed in the subject population. A generative theory of core procedures is needed. Such a theory has been developed (VanLehn, 1983). It is the topic of this chapter.

As the example above makes clear, some core procedures are a direct result of the fact that diagnostic tests are administered to students who have not yet completed the subtraction curriculum, and therefore have not yet been taught the entire algorithm. The incompleteness in their training causes their core procedure to lack some of the subprocedures that a correct, complete procedure would have. This incompleteness shows up as bugs on the diagnostic test. Testing beyond training explains why some core procedures have missing subprocedures.

Other core procedures, however, are not easily explained as missing subprocedures. Instead, some of the subprocedures have wrong information in them. The following is a simple example. In the correct subtraction procedure, the student should borrow when $T < B$ in a column, where T and B are the top and bottom digits, respectively, in the column. The bug N-N-Causes-Borrow performs a borrow when $T \leq B$ (see below).

```
   6
  7'5
 -2 5
 ----
  410
```

[illegible]ly, students with this bug have overgeneralized the condition for when [illegible]ow. Their core procedure is complete, but incorrect.

To sum up, there are a variety of core procedures. Some seem quite naturally to be the result of testing beyond training, whereas others seem to be the result of learning processes that have gone awry. The research project is to find a learning theory that generates (predicts) the core procedures that are found in the subject population.

Because the learning theory should actually construct the core procedures, a computational learning model is needed. It should take in something that represents the classroom experiences that the students have. Using that input, it should construct some knowledge structures that represent the core procedures that students acquire from those experiences. Researchers in Artificial Intelligence (AI) have built such computational models of learning (Cohen & Feigenbaum, 1983). There are models of skill acquisition based on induction (Biermann, 1972), analogy (Carbonell, 1983), learning-by-being-told (Badre, 1972), planning and debugging (Sussman, 1976), practice (Mitchell, Utgoff, & Banerji, 1983), and other techniques.

In general, these researchers make no attempt to empirically test their model's psychological validity, but there are exceptions. Perhaps the most thorough validation is Anderson's study of the acquisition of skill in geometry theorem proving and Lisp programming (Anderson 1983; Anderson, Greeno, Kline, & Neves, 1981; Anderson, Farrell, & Saurers; 1984). In Anderson's studies, the main data are protocols of students solving problems, with a textbook and a tutor beside them. Their comments and actions are coded and presented to the learning model. When the model is successful, it simulates the students' learning behavior accurately at a second-by-second level of detail. Despite the fact that learning such complex skills requires hundreds of hours of learning and practice, the protocols cover only two or three hours of an individual student's education. That is, of course, inevitable. It is impossible to record, analyze, and simulate the whole of a student's education. Consequently, significant extrapolation beyond the data is needed in order to claim that the observed samples of the learning process characterize the whole of the students' education.

Rather than taking a small sample of the students' education and analyzing it in great detail, the present research takes the whole of the students' education and analyzes it at a higher level of detail. The essential information in the students' experiences is abstracted and presented in an ideal form to the learning model. The key question is, what should this essential information be? This question is intimately related to the question of what the learning process is. If the learning process is inductive, as claimed earlier, then examples are the essential information to abstract from the curriculum and present to the learning model. If the learning process is learning-by-being-told, then the teacher's verbal explanations are the essential information. Either way, the objective is to find some learning process and its associated abstraction of the curriculum such that (1) the whole curriculum can be presented to the learning model, and (2) the learning model accurately predicts the core procedures that the students acquire. In short, the desired learning theory is a generative theory of core procedures (and hence bugs) that models learning over the whole curriculum.

This, then, is the interpretation under which the induction hypothesis seems true. Induction is more accurate than learning-by-being-told as a whole-curriculum, generative theory of bugs. This claim creates an interesting tension. How can induction be such a good model when the curriculum is viewed as a whole, and such a mediocre model when instruction is viewed on a second-by-second level? The research required to resolve the tension is just beginning. Some speculative explanations are offered in the last section of the chapter.

The body of the chapter is devoted to explicating and supporting the induction hypothesis. Three hypotheses will be defined and contrasted: the induction hypothesis, learning-by-being-told, and a third hypothesis—learning from analogies to familiar procedures. The familiar procedures used in arithmetic classrooms are usually ones for manipulating concrete numerals (e.g., coins, Dienes blocks, poker chips, Montessori rods, etc.). Analogy is included as a third hypothesis even though it is not particularly plausible as a stand-alone learning process. Much goes on in the classroom that does not involve drawing analogies to familiar procedures. However, it is plausible that analogy might go on in combination with induction or learning-by-being-told. That is, we might find that some bugs can be explained by analogy, and the rest can be explained by either induction or learning-by-being-told. Analogy may be of special interest in the context of this book, for it seems to engender (more so than either induction or learning-by-being-told) conceptual knowledge of arithmetic, as opposed to procedural knowledge.

At any rate, the first task is to clarify these three hypotheses. In the process, the role of the conceptual/procedural distinction, as it applies in this context, will also become clear. To this end, it is helpful to begin by making an assumption about the kind of knowledge that students acquire from the arithmetic curriculum.

SCHEMATIC vs. TELEOLOGICAL KNOWLEDGE

The assumption is that student's knowledge about procedures is schematic but not teleological. To define these terms, *schematic* and *teleological*, it is helpful to relate them to more familiar terms. (Fig. 6.1 is a road map of the terms to be discussed.) Computer programmers generally describe a procedure in three ways (N.B., the term "procedure" is being used temporarily to mean some very abstract, neutral idea about systematic actions):

- *Program:* A program is a schematic description of actions. It is schematic, because one must say what its inputs are before one can tell exactly what actions it will perform. That is, a program must be instantiated, by giving it inputs, before it becomes a complete description of a chronological sequence of actions.

- *Action sequences:* Executing a program produces an action sequence. In principle, one could describe a procedure as a set of action sequences. This is analogous to specifying a mathematical function as a set of tuples (e.g., n! as {< 0,1 >, < 1,1 >, < 2,2 >, < 3,6 >, < 4,24 >, . . .}).
- *Specifications:* Specifications say what a program ought to do. Often they are informally presented in documents that circulate among the programmers and market researchers on a product development team. Sometimes specifications are written in a formal language so that one can prove that a program meets them.

There are names for the processes of transforming information about the procedure from one level to another. *Programming* is the transformation of a specification into a program. *Execution, interpretation* and *running* are names for the transformation of a program into an action sequence. There are also names for static, structural representations of these transformations. A *trace* is a structural representation of the relationship between a program and a particular execution of it. A *procedural net* (Sacerdoti, 1977), a *derivation* (Carbonell, 1983) and a *planning net* (VanLehn & Brown, 1980) are all formal representations of the relationship between a specification and a program. Actually, these three terms are just a few of the formalisms being used in an ongoing area of investigation. Rich (1981) has concentrated almost exclusively on developing a formalism describing the relationship between a specification and a program. In his representation system, both the specification and the program are *plans*—the surface plan (program) is just a structural refinement of the other. Rather than seeming to commit to one or another of these various formalisms, the neutral term *teleology* will be used. Thus, the teleology of a certain program is information relating the program and its parts to their intended purposes (i.e., to the specification).

Because teleology may be an unfamiliar term, it is worth a moment to sketch its meaning. The teleology of a procedure relates the schematic structure (pro-

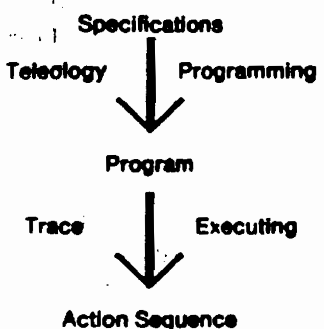

FIG. 6.1. Three levels of description for a "procedure." Names for the processes of converting from higher levels to lower levels are on the right. Names for conversion structures are on the left.

gram) of the procedure to its goals and other design considerations. The teleology might include, for instance, a goal-subgoal hierarchy. It might indicate which goals serve multiple purposes, and what those purposes are. It might indicate which goals are crucially ordered, and which goals can be executed in parallel. If the program has iterations or recursions, it indicates the relationship between the goals of the iteration body (or recursion step) and the goal of the iteration (recursion) as a whole. In general, the procedure's teleology explicates the *design* behind the procedure.

A procedure for making gravy serves well as an illustration of the difference between teleological and schematic knowledge. A novice cook often has the only schematic knowledge (a program) for the gravy recipe—which ingredients to add in which order. The expert cook will realize that the order is crucial in some cases, but arbitrary in others. The expert also knows the purposes of various parts of the recipe. For instance, the expert understands a certain sequence of steps as making a flour-based thickener. Knowing the goal, the expert can substitute a cornstarch-based thickener for the flour-based one. More generally, knowing the teleology of a procedure allows its user to adapt the procedure to special circumstances (e.g., running out of flour). It also allows the user to debug the procedure. For instance, if the gravy comes out lumpy, the expert cook can infer that something went wrong with the thickener. Knowing which steps of the recipe make the thickener, the cook can discover that the bug is that the flour-fat mixture (the roux) wasn't cooked long enough. The purpose of cooking the roux is to emulsify the flour. Because the sauce was lumpy, this purpose wasn't achieved. By knowing the purposes of the parts of the procedure, people are able to debug, extend, optimize, and adapt their procedures. These added capabilities, beyond merely following (executing) a procedure, can be used to test for a teleological understanding.

It is an empirical question whether the students' knowledge level corresponds to the schematic level (i.e., a program for the procedure) or the teleological level. The assumption made here is that their knowledge is schematic. This is a rather uncontroversial assumption. In fact, much research in arithmetic begins with the assumption that current instruction gives students only a schematic (or procedural) knowledge of arithmetic, then seeks some new instructional methods that will give them a teleological (conceptual) knowledge of the skill. Although the assumption of schematic knowledge is uncontroversial, it might be worth discussing it a little further in order to illustrate how one might detect teleological knowledge if one succeeded in teaching it.

One hallmark of expert cooks, and others who have a teleological knowledge of procedures, is their ability to debug and extend the procedures when necessary. Gelman and her colleagues (Gelman & Gallistel, 1978; Greeno, Riley, & Gelman, 1984) used tests based on debugging and extending procedures in order to determine whether children possess the teleology for counting. Adapting their techniques, I tested five adults for possession of teleology for addition and

subtraction. All subjects were competent at arithmetic. None were computer programmers. The subjects were given nine tasks. Each task added some extra constraint to the ordinary procedure, thereby forcing the subject to redesign part of the procedure in order to bring it back into conformance with its goals. A simple task, for example, was adding left to right. A more complex task was inventing the equal additions method of borrowing (i.e., the borrow of 53 − 26 is performed by adding one to the 2 rather than decrementing the 5).

The results were equivocal. One subject was unable to do any of the tasks. The rest were able to do some but not all of the tasks. The experiment served only to eliminate the extremes: Adults don't seem to possess a complete, easily used teleology, but neither are they totally incapable of constructing it (or perhaps recalling it). Further experiments of this kind may provide more definitive results. In particular, it would be interesting to find out if adults were constructing the teleology of the procedure, or whether they already knew it. At any rate, it's clear that not all adults possess operative teleology for their arithmetic procedures, and moreover, some adults seem to possess only schematic knowledge of arithmetic.

Adults found the teleology test so difficult that I was unwilling to subject young children to it. However, there is some indirect evidence that students acquire very little teleology. It concerns the way students react to impasses. Consider the decrement-zero impasse discussed earlier. The hypothetical student hasn't yet learned how to borrow from zero although borrowing from non-zero numbers is quite familiar. Given the problem

$$\begin{array}{r} 604 \\ -217 \\ \hline \end{array}$$

the student starts to borrow, attempts to decrement the zero, and reaches an impasse. If the student understands the teleology of borrowing, then the student understands that borrowing from the hundreds would be an appropriate way to fix the impasse. That is, the teleology of non-zero borrowing allows it to be easily extended to cover borrowing from zero. Although some students may react to the decrement-zero impasse this way, many do not. They repair instead. Because students do not make teleologically appropriate responses to impasses, it appears that they did not acquire much teleology (or if they did, they are unwilling to use it—in which case it's a moot point whether they have it or not).

THREE WAYS THAT ARITHMETIC COULD BE LEARNED

Given the assumption that students' knowledge of arithmetic procedures is schematic, we can more accurately address the issue of how they acquire that knowledge. The tripartite distinction between specifications, programs, and actions

sequences will be used again. If the goal is to construct a description of the procedure at a schematic (program) level, there are four possible routes (see Fig. 6.2):

1. From specification to program: A kind of learning by discovery.
2. From examples (action sequences) to programs: induction.
3. From some other schematic description, either
   a. another familiar program: learning by analogy, or
   b. a natural language presentation of the program: learning-by-being-told.

The first possibility is not particularly plausible in the domain of arithmetic. The teleology of arithmetic is very complex, and the curriculum would have to modified radically in order to teach it. VanLehn and Brown (1980) present a complete teleology for addition, and discuss how it could be taught. This form of learning will not be considered further here. However, the remaining three forms of learning—induction, analogy, and learning-by-being-told—are exactly the three hypotheses to be discussed.

The best way to compare these three hypotheses would be to develop three complete generative theories, one for each hypothesis, then see which theory is better according to the usual scientific criteria. In fact, only the induction hypothesis has such a theory behind it (VanLehn, 1983). The theory is quite rigorously formulated. There are 31 main hypotheses. The induction hypothesis is one of them. The other 30 hypotheses concern the form of the knowledge representation, the mechanisms for impasses and repairs, and the details of the inductive learning process. From the standpoint of supporting the induction hypothesis, this degree of rigor presents some problems. First, the empirical adequacy of the theory depends on all the hypotheses and not just the induction hypothesis. Thus, if the theory fails to generate a certain bug, this does not necessarily mean that the bug can not be acquired by induction. It could be that some other hypotheses in the theory are wrong, and they should shoulder the blame for the theory's

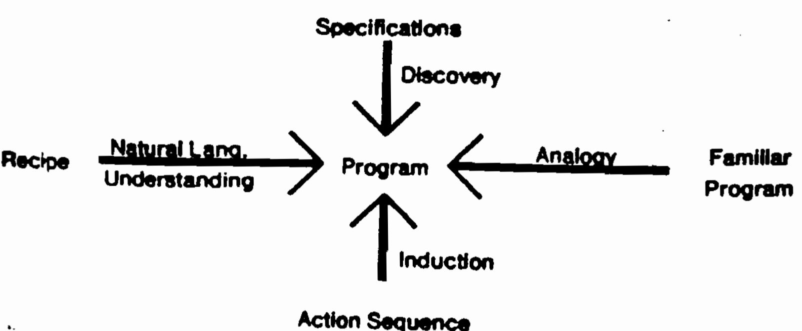

FIG. 6.2. Ways to acquire a program level of description.

inability to predict the bug. This problem of assigning blame to hypotheses can be solved, but it takes a very careful analysis of the relationships among the hypotheses and the data. That analysis has been undertaken, but it is too lengthy to present here (see (VanLehn, 1983)). Instead, two informal analyses of the the theory's empirical adequacy will be presented.

The first analysis will be a conservative evaluation of the induction hypothesis. It will present exactly the bugs the theory can generate in the task domain of subtraction. It will turn out that the theory can generate 33% of the observed bugs. This conservative evaluation confounds the effects of the induction hypothesis with the other 30 hypotheses. In order to pick their effects apart somewhat, the derivations of a few of the bugs will be presented. The relationship of the induction hypothesis to these bugs is typical of its relationship to other, similar bugs.

The second analysis will be a liberal estimate of the generative power of the induction hypothesis. It is meant to indicate how many bugs the induction hypothesis could generate if the other hypotheses in the theory were relaxed or discarded. Essentially, it is an estimate of the generative power of the best possible inductive learning theory. It will be shown that 85% of the observed bugs can be generated.

Following these two analyses, there will be a discussion of the two competing hypotheses, learning by analogy and learning-by-being-told. Because no generative theories have been developed for these hypotheses, their empirical predictions will be derived informally. It will be shown that they are not as productive as the induction hypothesis.

THE CONSERVATIVE EVALUATION OF THE INDUCTION HYPOTHESIS

First, some background. The bug data to be presented were collected from 1147 subtraction students in grades 2 through 5. The collection and analyses of these data are detailed in VanLehn (1982). The learning model used to generate the theory's predictions is the one documented in VanLehn (1983). The model has changed since then, and its predictions have improved. However, the figures from that publication are used because it presents a detailed account of how they were generated.

The overall adequacy of the theory is displayed in Fig. 6.3. There are 76 observed bugs. The theory generates 49 bugs, 25 of which are observed. These 25 bugs are confirmed predictions. Seventeen of the predicted bugs are plausible, but have not yet been observed. Perhaps if another thousand students were examined, some of these would be found. However, 7 of the predicted bugs are so strange that it is extremely doubtful that they would ever be observed. These bugs should not be generated by the theory. Of the observed bugs, 51 are not

generated by the theory. This is not as damning an inditement of the theory as the 7 implausible bugs. It could be that other bug-generating processes are at work, and they are responsible for some of the 51 bugs. When those processes are discovered, they can be added to the theory, converting some of the 51 bugs to confirmed predictions. However, the generation of the 7 implausible bugs can't be fixed by adding another bug-generating process to the theory. They indicate problems with the present theory that need rectification.

In developing the theory, it was often the case that one could increase the number of confirmed predictions but only at the expense of increasing the number of implausible predictions. In the case of this theory, such choices were always made in favor of reducing the number of implausible predictions. If these choices had been made the other way, then many of the 51 unaccounted for bugs would be accounted for. The liberal evaluation, which will be presented in the next section, indicates roughly how many of the 51 bugs could be converted to confirmed predictions if the theory were liberalized.

The numbers presented above are difficult to understand without some point of reference. Two such points are provided by earlier generative theories of subtraction bugs. An early version of repair theory is documented in Brown and VanLehn (1980). Its empirical adequacy can be compared with the present theory's. Clearly, this theory will do better since it includes the ideas of its predecessor. Another generative theory of subtraction bugs was developed by Richard Young and Tim O'Shea (1981). They constructed a production system for subtraction such that deleting certain of its rules (or adding other rules, in some cases) would generate observed bugs. They showed that these mutations of the production system could generate many of the bugs described in the original Buggy report (Brown & Burton, 1978).

A chart comparing the results of the three theories is presented as Table 6.1. Observed bugs that no theory generates are not listed, nor are bugs that have not

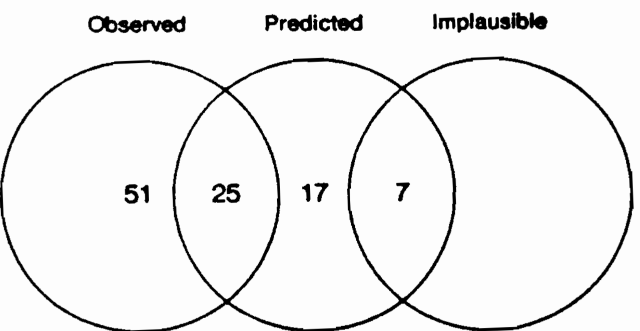

FIG. 6.3. A Venn Diagram showing relationships and size of the sets of observed and predicted bugs.

TABLE 6.1
Comparison of Observed Bugs Generated by Three Theories:
Y&O = Young and O'Shea; B&V = Brown & VanLehn;
Cur. = current theory

Y&O	*B&V*	*Cur.*	*Occurs*	*Bug*
		✓	6	Always-Borrow-Left
		✓	1	Blank-Instead-of-Borrow
		✓	7	Borrow-Across-Second-Zero
✓	✓	✓	41	Borrow-Across-Zero
		✓	4	Borrow-Don't-Decrement-Unless-Bottom-Smaller
		✓	2	Borrow-From-One-Is-Nine
		✓	1	Borrow-From-One-Is-Ten
✓	✓	✓	14	Borrow-From-Zero
✓			1	Borrow-From-All-Zero
	✓	✓	2	Borrow-From-Zero-Is-Ten
	✓	✓	18	Borrow-No-Decrement
		✓	6	Borrow-No-Decrement-Except-Last
		✓	1	Borrow-Treat-One-As-Zero
	✓		1	Can't-Subtract
		✓	1	Doesn't-Borrow-Except-Last
✓			15	Diff-0−N=0
✓			43	Diff-0−N=N
✓			1	Diff-N−N=N
✓			6	Diff-N−0=0
		✓	7	Don't-Decrement-Zero
		✓	4	Forget-Borrow-Over-Blanks
		✓	2	N−N-Causes-Borrow
		✓	1	Only-Do-Units
	✓	✓	5	Quit-When-Bottom-Blank
	✓		4	Stutter-Subtract
✓	✓	✓	115	Smaller-From-Larger
		✓	3	Smaller-From-Larger-Except-Last
	✓	✓	5	Smaller-From-Larger-Instead-of-Borrow-From-Zero
		✓	7	Smaller-From-Larger-Instead-of-Borrow-Unless-Bottom-Smaller
		✓	3	Stops-Borrow-At-Multiple-Zero
✓	✓	✓	64	Stops-Borrow-At-Zero
	✓	✓	1	Top-Instead-of-Borrow-From-Zero
✓	✓		5	Zero-Instead-of-Borrow
10	12	25		totals

been observed. Brown and VanLehn (1980) count bugs differently than they are counted here. (See VanLehn 1983, pg. 68, for details). The chart shows that the present theory generates more bugs, which is not surprising since it embeds many of the earlier theories' ideas. What is perhaps a little surprising is that there are a few bugs that they generate and it does not. These bugs deserve a closer look.

Young and O'Shea's model generates a class of bugs that they call *pattern errors*. Four bugs were included in this class:

Diff-0 − N = N If the top of a column is 0, write the bottom as the answer.

Diff-0 − N = 0 If the top of a column is 0, write zero as the answer.

Diff-N − 0 = 0 If the bottom of a column is 0, write the zero as the answer.

Diff-N − N = N If the top and bottom are equal, write one of them as the answer.

Young and O'Shea derive all four bugs the same way. Each bug is represented by a production rule, and the rule is simply added to the production system that models the student's behavior. Phrased differently, they derive the bugs formally by stipulating them, then explain the stipulation informally. Their explanations are:

> From his earlier work on addition, the child may well have learned two rules sensitive to zero, NZN and ZNN [two rules that mean N + 0 = N and 0 + N = N]. Included in a production system for subtraction, the first, NZN, will do no harm, but rule ZNN will give rise to errors of the "0 − N = N" type. Similar rules would account for the other zero-pattern errors. If the child remembers from addition just that zero is a special case, and that if a zero is present then one copies down as the answer one of the numbers given, then he may well have rules such as NZZ or ZNZ [the rules for the bugs Diff-N − 0 = 0 and Diff-0 − N = 0]. . . . .Rule NNN [the rule for the bug Diff-N − N = N] covers the cases where a child asked for the difference between a digit and itself writes down that same digit. It is clearly another instance of a "pattern" rule. (Young & O'Shea, 1981, pg. 163).

The informal explanations, especially the one for Diff-0 − N = N, are plausible. (Note, by the way, that some of these explanations crucially involve induction.) To treat them fully, one would have to explain why *only* the zero rules are transferred from additions, and not the other addition rules.

The point is that one can have as much empirical adequacy as one wishes if the theory is not required to explain its stipulations in a rigorous, formal manner. The present theory could generate the same pattern bugs as Young and O'Shea's model simply by making the appropriate modifications to the core procedures

and reiterating their informal derivation (or tell any other story that seems right intuitively). This would not be an explanation of the bugs, but only a restatement of the data embroidered by interesting speculation. This approach does not yield a theory with explanatory value. In short, there is a tradeoff between empirical adequacy and explanatory adequacy. If the model does not itself construct the appropriate knowledge representations, then it is the theorist and not the theory that is doing the explaining. The theory per se has little explanatory value, even though it might cover the data quite well. The present theory aims first for explanatory adequacy, even if that means sacrificing empirical adequacy.

With the foregoing background information in hand, we can return to discussing the induction hypothesis. Induction is generalization from examples. Almost always, there are many possible concepts consistent with a given set of examples. Some will be more general than the target concept, and some will be more specific. If the learner is not somehow given extra information about which of the consistent concepts is the target concept, then there is a strong chance that the learner will guess wrong, and pick either an overgeneralized or overspecialized concept instead. If human students are learning inductively, then there should be many bugs that are the result of overgeneralization or overspecialization.

In order to demonstrate the influence of the induction hypothesis, the bugs generated by the theory will be divided into several groups, and characteristic bugs from each group will be discussed.

The first group consists of bugs generated by overgeneralization of the conditions that determine whether or not to perform a subprocedure. A prototypical case is the bug N-N-Causes-Borrow, which was mentioned earlier. The proper test for when to borrow is $T < B$, where T is the top digit of a column and B is the bottom digit. The bug borrows when $T \leq B$. This makes sense given the induction hypothesis. The student sees many columns, some with $T > B$ and some with $T < B$. The student induces that the $T < B$ columns trigger borrowing, and the $T > B$ columns don't. However, the student must see a $T = B$ column in order to induce that $T = B$ columns don't trigger borrowing. Such problem types are rare in the textbooks used by the students in our subtraction study, and they never occupy a lesson of their own. Nowhere in the teacher's guides are $T = B$ columns pointed out as deserving special mention. So it is plausible to assume that sudents never notice that $T = B$ columns do not require borrowing. Unless a $T = B$ column is examined, the student doesn't know whether the borrowing predicate is $T \leq B$ or $T < B$. The students who guess $T \leq B$ show up in the data as having the bug N − N-Causes-Borrow, and the students who guess $T < B$ will show up as having the correct test for borrowing. Both kinds of performance have been observed.

A similar explanation works for the bug Borrow-Treat-One-As-Zero. The proper test for when to borrow across zero is $T = 0$ (see problem A, below). However, students with this bug will perform the subprocedure when $T = 1$ or $T = 0$ (see problem *b*).

```
      29          29
a.   30'5    b.  31'5
    -10 9       -10 9
     19 6        19 6
```

Perhaps the student thinks that 1 and 0 are special numbers (which they are, for they are the identity elements of the field), and that "*T* is a Special Number" is the appropriate test condition for when to perform the borrow-from-zero sub-procedure. The bug can be explained by the induction hypothesis and the fact that problems like *b* are rare among the textbooks' examples. Two other bugs in this class of overgeneralized test condition bugs are Borrow-From-One-Is-Nine and Borrow-From-One-Is-Ten.

The preceding group of bugs illustrated that the test conditions can be over-generalized. One would also expect overspecialization, if the induction hypothesis is correct. The next group of bugs result from overspecialization of test conditions. (These bugs were not generated by the version of the theory discussed earlier, and do not appear in Table 6.1. They are generated by the learning model in its present form.) The following is a classic case of a bug generated by overspecialization of a test condition. In the textbooks in our study, borrowing is introduced in two-column problems. This means that the borrow originates in the units column. Some students who are tested at this point in the curriculum believe that borrowing can only be triggered in the units column (see problem *a*, below). This is inductively correct, because they have not yet seen examples that contradict their overspecialized idea about when to borrow.

```
       5            5 2           6 5           2
a.   656'5    b.  6'53'5    c.   7 6'5   d.   53'5
    -191 9       -1 98 9        -1 9 8       -11 9
     544 6        4 66 6         5 6 7        41 6
```

A similar story explains the bug shown in problem *b*. This bug, Smaller-From-Larger-When-Borrowed-From, borrows in any $T < B$ column, except those where the top digit in the column has been scratched out already, as in the tens column of problem *b*. The textbooks delay teaching about adjacent borrowing (e.g., problem *c*) until well after isolated borrowing has been taught (e.g., problem *d*). Consequently, a student who is tested between these two lessons cannot yet have seen a $T < B$ column with the top digit scratched out, and therefore cannot know whether to borrow or not. In short, for both these bugs, the students have an overspecialized concept of when to borrow because certain examples haven't yet been presented to them. Two other bugs, X − N = 0-After-Borrow and X − N = N-After-Borrow, have nearly identical derivations.

Another four bugs are generated by overspecialized tests of when to perform borrowing across zero. They are: Borrow-Across-Zero-Over-Zero, Borrow-Across-Zero-Over-Blank, Don't-Decrement-Zero-Over-Zero, and Don't-Decrement-Zero-Over-Blank.

So far, the bugs discussed have concerned ***when*** to borrow. The next group of bugs concerns a different kind of procedural knowledge, ***how*** to borrow. Students with the bug Always-Borrow-Left borrow from the leftmost column in the problem no matter which column triggers the borrowing. Problem *a* below shows the correct placement of borrow's decrement. Problem *b* shows the bug's placement.

```
       5           2          5
a.   36'5    b.  36'5   c.   6'5
    -10 9       -10 9       -1 9
     25 6        16 6        ·4 6
```

The explanation begins with the fact that borrowing is introduced using only two-column problems, such as problem *c* above. Multi-column problems, such as *a*, are not used. Consequently, the student has insufficient information to unambiguously induce where to place borrow's decrement. The correct placement is in the left-adjacent column, as in *a*. However, two-column examples are also consistent with decrementing the leftmost column, as in *b*. Once again, the induction hypothesis provides the key for explaining how a bug is acquired. Other bugs in this group are Forget-Borrow-Over-Blanks, Only-Do-Units, Borrow-Don't-Decrement-Unless-Bottom-Smaller, Smaller-From-Larger-Unless-Bottom-Smaller. Doesn't-Borrow-Except-Last, and Smaller-From-Larger-Except-Last.

The other bugs generated by the theory stem from core procedures that are incomplete, rather than misinduced. Examples of bugs in this group are Stops-Borrow-At-Zero and Borrow-Across-Zero, which were described in the introduction. The bugs in this group are consistent with the induction hypothesis. Indeed, the inductive learning model constructs their core procedures from the initial few lessons of the lesson sequence. However, these bugs also would be consistent with any form of learning that could derive a correct (albeit incomplete) procedure from an initial segment of the lesson sequence. So the bugs in this group confirm the induction hypothesis, but don't differentiate it from other hypotheses.

This completes the conservative analysis of the empirical adequacy of the induction hypothesis. It was shown that overgeneralization and overspecialization could account for many bugs. On a deeper level, there are three ultimate causes for the bugs. (1) Many bugs are generated by testing subskills that haven't been taught yet. Stops-Borrow-At Zero is a good illustration of this. (2) Certain critical examples are missing from the curriculum or under-emphasized. The bug N − N-Causes-Borrow is a clear case of this. If the critical examples were added to the curriculum and emphasized, such bugs might not occur. (3) Certain examples that should be grouped into the same lesson are separated into two lessons, and a diagnostic test is administered in between them. The bug Always-Borrow-Left is a good illustration of this. If the two lessons were merged or placed close together, this bug may not occur. In short, all three causes for bugs can be cured

by modifying the curriculum and the testing policies—or so it seems. These are suspiciously crisp predictions. I suspect that there will be some surprises if one tries to eliminate bugs by changing instruction in the indicated ways.

A LIBERAL EVALUATION OF THE INDUCTION HYPOTHESIS

As mentioned earlier, the theory generates only 33% of the observed subtraction bugs. It does not, for instance, generate the bug Diff-0 − N = N. This bug could be explained as an overgeneralization of the correct subtraction rule that N − 0 = N. The current theory could in fact generate the bug by overgeneralization, but only if the rule N − 0 = N is taught in a separate lesson, which it is not in the textbook series used by the subjects. Similarly, other bugs could be induced if the theory were tested less rigorously than it was, or if the various ancillary hypotheses of the theory were relaxed. This section estimates the best possible empirical adequacy that a theory based on induction could have. It will be, of course, be a rough estimate.

The estimate is based on the following, admittedly weak, line of reasoning. (The weakness of this kind of reasoning, by the way, is exactly why one must build theories.) If a bug is induced from examples, then there ought to be some way to describe it in terms of the visual and numerical features that examples have. For instance, the bug Diff-0 − N = N can be described by a rule "If T = 0, then write B in the answer," where T and B are the top and bottom digits in a column. The feature $T = 0$ is a visual-numerical feature. On the other hand, if a bug is not acquired by induction, then it might be difficult to describe it with simple rules composed of visual-numerical features. For instance, the bug Borrow-Unit-Difference seems to be acquired noninductively. In a correct borrow, the student adds ten to T and decrements the next digit over by one. Borrow-Unit-Difference calculates how much needs to be added to T in order to make T equal B, then it decrements the next digit over by that amount:

```
 49
 85
-19
---
 21
```

Teachers who use coins, Dienes blocks, or other concrete numerals to teach subtraction will recognize this bug almost immediately. In a monetary representation, the problem above is "You have 8 dimes and 5 pennies. What do you have left if you give me one dime and 9 pennies?" There is a bank, consisting of piles of dimes and pennies, that the student may use to make change. Many students, seeing that they need to hand out 9 pennies, will take exactly four pennies from the bank. That's all they need to make 9. However, they know that

they must make change fairly, so they hand in four dimes. They have got almost everything right and they are doing a fine job of means-ends analysis problem solving, except that they have one constraint wrong: They think that dimes and pennies are worth the same amount in this context. So their concrete procedure has a small bug in it. To put it differently, when Borrow-Unit-Difference is represented as problem-solving with concrete numerals, it has a succinct, accurate representation as the substitution of an incorrect constraint for the correct one. However, there is no such succinct representation when the bug is represented as a procedure for manipulating written numerals. In particular, it can not easily be represented as a condition-action rule using simple visual-numerical features.

When a bug is learned by processes other than induction, such as the analogy process that evidently underlies this bug's acquisition, it will be difficult to represent as condition-action rules over a vocabulary of simple visual-numerical features. Consequently, the ease of description of a bug in these terms can be used as a weak test for whether it can be generated by an inductive theory.

At the end of this chapter all 78 currently observed subtraction bugs are listed. (Two bugs have been discovered since the empirical results of Fig. 6.3 were calculated.) The list demonstrates that 66 of these (i.e., 85%) can be easily represented as condition-action perturbations to a correct procedure. The set of visual-numerical features used in these perturbations are listed in Table 6.2.

Of the observed subtraction bugs, 12 could not be represented in a simple way with visual-numerical features. These bugs are potentially disconfirming to the induction hypothesis. Let's examine them. Of the 12 bugs, 5 have relatively simple explanations. One of them, Borrow-Unit-Difference, has been discussed already. It seems to have been acquired by analogy from concrete manipulative procedures. One bug, Add-Instead-Of-Sub, seems to result from confusing addition with subtraction. Simple-Problem-Stutter-Subtract seems to be a confusion between multiplication and subtraction. When the bottom row of the subtraction problem has a single digit in it, the student uses a pattern of calculation similar to multiplication: the single digit is subtracted from each digit in the top row. Stutter-Subtract is generated by the theory from an incomplete procedure, one that doesn't know how to handle columns with a blank in the bottom. Don't-Write-Zero seems to be due to overgeneralization of a special kind. Somewhere in the curriculum (there is no lesson for it), teachers instruct students to omit writing a zero if the zero will be the leftmost digit in the answer. That is, younger students answer as in *a*, below, and older students answer as in *b*:

```
     2         2
a.  3¹5   b.  3¹5   c.  339
   -2 9      -2 9      -237
   ----      ----      ----
    0 6         6       1 2
```

Apparently, the bug Don't-Write-Zero, whose work is shown in *c*, is an overgeneralization of the prohibition against writing leading zeros. So this bug actu-

TABLE 6.2
The Visual-Numerical Features

B=#	The bottom digit in the column is blank.
B=0	The bottom digit in the column is zero.
BORROWED	A borrow has taken place already.
DECR'ED	The top digit in the column has been decremented.
DECR'ED/BOT	The bottom digit in the column has been decremented.
IN/LAST/COLUMN	The current column is the leftmost one.
IN/NEXT/TO/LAST/COLUMN	The current column is the penultimate one.
NEXT/B=#	The bottom digit in the next column to the left is blank.
NEXT/T=0	The top digit in the next column left is zero.
NEXT/T=00	The top two digits in the next two columns to the left are zero.
NEXT/T=1	The top digit in the next column left is one.
T≤B	The top digit is less than or equal to the bottom digit.
T<B	The top digit is strictly less than the bottom digit.
T>B	The top digit is strictly greater than the bottom digit.
T=B	The top digit is equal to the bottom digit.
T=B/ORGINALLY	T=B in the original column, before decrementing occurred.
T=#	The top digit in the column is blank.
T=0	The top digit in the column is zero.
T=00	The top digit in the column and the next one to its left are zero.
T=1	The top digit in the column is one.
T=1V0	The top digit is a one or a zero.
T=9	The top digit is nine.

ally supports the induction hypothesis, even though it can't be simply expressed in the given feature vocabulary.

Of the 12 bugs, the remaining 7 baffle me. If the reader has explanations for any of the these bugs, I would appreciate hearing about them.

At any rate, if appears that, whatever the noninductive learning processes are, they generate only a few bugs each. Induction, on the other hand, seems able to handle, in principle, 85% of the bugs. This 85% includes all the most commonly occurring bugs, as the appendix documents.

This completes the informal, liberal analysis of the empirical adequacy of the induction hypothesis. Perhaps the evidential relationships are a little weak, but the data side strongly with the induction hypothesis. The remaining two sections discuss the competing hypotheses, learning by analogy and learning by being told.

LEARNING BY ANALOGY

Learning by analogy is the mapping of knowledge from one domain over to the target domain, where it is applied to solve problems. Analogies are used in the early grades to teach base-10 numeration. Students are drilled on the mapping

between written numerals and various concrete representations of numbers, such as collections of coins, Diennes blocks, Montessori rods, and so forth. This is a mapping between two kinds of numerals, and not two procedures. Later, this internumeral mapping is drawn on in teaching carrying and borrowing. For example, a known procedure for making change—trading a dime for ten pennies—is mapped into the borrowing procedure of written subtraction. Many textbooks and teachers' manuals advocate this method of teaching by analogy. Although it is not clear how much this technique is actually used in the classroom, it warrants our attention as one possible hypothesis about how students learn procedures. Indeed, one piece of evidence for the analogy hypothesis, the bug Borrow-Unit-Difference, has been discussed already. However, it is only one bug, and only one subject in our 1147-student sample had it.

Presumably, once an analogy has transferred some knowledge, it is still available for use later to transfer more knowledge about the procedure. In some cases, this predicts significant student competence. For instance, if the students learned simple borrowing via the analogy, then it's quite plausible that when confronted with more complex borrowing problems, such as

$$\begin{array}{r} 607 \\ -238 \\ \hline \end{array}$$

(assuming the student hasn't yet been taught how to solve such borrow across zero problems), the student could solve the problem in the concrete domain by trading a dollar for nine dimes and ten pennies, then map back into the written domain, thus producing the correct solution. Indeed, the analogies used in instruction may have been designed so that these productive extensions of the base analogy are encouraged.

But this is a much more productive understanding of borrowing than most students achieve. As discussed earlier, when most students discover that it is impossible to decrement the zero, they repair. These students do not use analogies to familiar procedures (e.g., making change). If the students had learned their procedures via analogy, one would have to make ad hoc stipulations to explain why they no longer use that analogy after they have learned the procedure. It's more plausible that they simply didn't utilize the analogy in the first place. Loosely speaking, learning by analogy is too good. It predicts that students would fix impasses by constructing a correct extension to their current procedure. That is, they would *debug* instead of repair. Because many students have repair-generated bugs, another explanation would be needed for how these students acquired their procedures. At the very least, analogy cannot be the only kind of learning going on, if it happens much at all.

Carbonell (1983) makes a telling argument about analogies between procedures. His ARIES program was unable to form analogies between certain procedures when all it had was the program (schematic) representations. However, Carbonell found that analogies could be forged when the procedures were described teleologically (i.e., in Carbonell's terminology, the analogy is between

derivations of procedures). Suppose one stretches Carbonell's results a little and claims that knowing the teleology (derivation) of procedures is *necessary* for procedural analogy. (Carbonnell claims only sufficiency, if that.) Because most math students are ignorant of the teleology of their procedures, as discussed earlier, one can conclude that students did not acquire their procedures via analogy.

How is it that teachers can present material that is specifically designed to encourage learning procedures by analogy, and yet their students show few signs of doing so? Winston's research (Winston, 1979) yields a speculative answer. It indicates that the most computationally expensive part of analogy can be discovering how best to match the parts of the two sides of the analogy. To solve electrical problems given hydraulic knowledge, one must match voltage, electrical current, and resistance to one each of water pressure, current, and pipe size. There are six possible matchings, and only one matching is correct. The number of possible matchings rises exponentially with the number of parts. For a similar analogy, a best match must be selected from 11! or 40-million possible matches. The matching problem of analogy is a version of a NP-complete problem: finding the maximal common subgraph of two digraphs (Hayes-Roth & McDermott, 1978). Hence, it is doubtful that a faster solution than an exponential one exists. Even if the matching algorithm were implemented on a connection machine (i.e., a computer that is like a neural net in that it uses millions of small processors arranged in a network instead of a single powerful processor, as conventional computers use), it seems that the combinatorics would not decrease radically (D. Christman, personal communication).

If computational complexity can be equated with cognitive difficulty, Winston's work predicts that students may find it difficult to draw an analogy unless either it is a very simple one (i.e., few parts) or they are given some help in finding the matching. Resnick (1982) has produced some experimental evidence supporting this prediction. Resnick interviewed students who were taught addition and subtraction in school, using the usual analogies between concrete and written numerals. She discovered that some students had mastered both the numeral analogy and the arithmetic procedures in the concrete domain, and yet they could not make a connection between the concrete procedures and the written ones. Resnick went on to demonstrate that students could easily make the mapping between the two procedures provided that the steps of the two procedures were explicitly paired. The students were walked through the concrete procedure in parallel with the written one. A step in one was immediately followed by the corresponding step in the other.[1] If we assume the conjecture from above, that combinatorial explosions in mapping equates with difficulty for humans making analogies, and we assume that "parts" of procedures roughly

[1]Although Resnick's technique makes learning by analogy easier, it does not guarantee that such learning will occur. In recent work, Resnick found that the instruction was effective only half the time (Resnick & Omanson, in press).

correspond to steps, then Resnick's finding makes perfect sense. The procedures are currently presented in school in a nonparallel mode. This forces students to solve the matching problem, and most seem unable to do so. Consequently, the analogy does little good. Only when the instruction helps the students make the matching, as it did in Resnick's experiment, does the analogy actually succeed in transferring knowledge about one procedure to the other. In short, analogy could become a major learning technique, but current instructional practices must be changed to do so.

There is anecdotal evidence that analogy is common, but it is analogy of a very different kind. In tutoring, I have watched students flip through the textbook to locate a worked problem that is similar to the one they are currently trying to solve. They then draw a mapping of some kind between the worked problem and their problem that enables them to solve their problem. Anderson et al. report the same behavior for students solving geometry problems (Anderson, Greeno, Kline, Neves, 1981) and Lisp problems (Anderson, Farrell, & Saurers, 1984). Although the usage could be disputed, Anderson et al. use "analogy" to refer to this kind of example-exercise mapping. It differs significantly from the kind of analogy discussed earlier. The abstraction that is common to the two problem solutions is exactly the schematic knowledge (program) of the procedure. In the analogy between making change and borrowing, the common abstraction lay much deeper, somewhere in the teleology (conceptual basis) of the procedure. To state it differently, the example-exercise analogy maps two action sequences of a procedure together, thus illustrating the procedure's schematic structure (program). The other analogy maps two distinct procedures together in order to illustrate a common teleology.

The former mapping, between two instances of a schematic object, is nearly identical to induction. In both cases, the most specific common generalization of the two instances is calculated. Winston also points out the equivalence of generalization and analogy in such circumstances (Winston, 1979). Although I have not investigated example-exercise analogy in detail, I expect it to behave indistinguishably from learning by generalizing examples.

To summarize, one form of analogy (if it could be called that) is indistinguishable from induction. The other form of analogy seems necessarily to involve the teleology of procedures. Because students show little evidence of teleology, it is safe to assume that analogic learning is not common in classrooms, perhaps because current instructional practices aren't encouraging it in quite the right way.

LEARNING BY BEING TOLD

One framework for acquiring a procedure involves following a set of natural language instructions until the procedure is committed to memory. This framework for explaining learning is called *learning by being told* (Winston, 1978). It

views the central problem of learning as one of natural language understanding. There are possible several sources for the natural language "recipes." One is the teacher, who explains the algorithm while presenting it. Another source is the textbook. Because students spend most of their time doing seatwork, when their major source of recipes is the textbook, let's begin by examining what the textbooks say. The key issue is whether the textbooks describe the procedure in enough detail. If they do, then all the students need to do is understand the language, and they will be able to perform the procedure.

Manuals of procedures are ubiquitous in adult life. Examples are cookbooks, user guides, repair shop manuals, and office procedure manuals. In using procedure manuals, adults sometimes learn the procedures described therein, and cease to use the manuals. So learning by being told is probably quite common among adults. The content of procedure manuals can be taken as a model for how good a natural language description has to be if it is to be effective in teaching the procedure.

Open any arithmetic text, and one immediately sees that it is not much like a cookbook or an auto repair manual. There is very little text; the books are mostly practice exercises and worked examples. The reason is obvious: Because students in the primary grades are just beginning to read, they could make little use of an elaborate written procedure.

Badre (1972) built an AI program that reads the prose and examples of a fourth grade arithmetic textbook in order to learn procedures for multicolumn addition and subtraction. Badre sought in vain for simple, concise statements of arithmetic procedures that he could use as input to his natural language understanding program. He comments:

> During the preliminary work of problem definition, we looked for a textbook that would explain arithmetic operations as a clearly stated set of rules. The extensive efforts in this search led to the following, somewhat surprising result: nowadays, young American grade-school children are never told how to perform addition or subtraction in a general way. They are supposed to infer the general algorithms from examples. Thus actual texts are usually composed of a series of short illustrated stories.' Each story describes an example of the execution of the addition or the subtraction algorithms. (Badre, 1972, pg. 1–2)

Despite the fact that Badre's program "reads" the textbook's "stories" in order to obtain a description of the examples, the role of reading in its learning is minimal. The heart of the program is generalization of examples. In particular, the program employs only a few heuristics that use the book's prose to disambiguate choices left open by generalization.

Although textbooks don't seem to have the right sort of language to make learning-by-being-told work, perhaps the teachers supply it. It is infeasible to find out everything that teachers say in classrooms over the years that arithmetic is taught. However, one way to test this hypothesis is to make a plausible assumption about what teachers might give as an explanation, and see whether that makes any useful predictions. Suppose that the teacher is describing borrowing for the first time. As mentioned earlier, borrowing is invariably introduced with two-column problems, such as problem *a*.

```
        5             2
a.     6̸¹5     b.   3̸6¹5
      - 1 9         -10 9
      -----         -----
        4 6          46 6
```

Under the induction hypothesis, this causes Always-Borrow-Left, as in *b*. Let's see what kind of prediction is produced by assuming that learning is dominated by natural language understanding.

In problems like *a*, "tens column" is probably the most common noun-phase used to describe the place to decrement from. Under the natural language hypothesis, "tens column" would be how students would describe to themselves where to do the decrement. This predicts that if they are given borrowing problems with more than two-columns, then they would always decrement the ten column, as in *c* and *d* below:

```
          5              15
c.    1¹56̸5     d.    3¹6̸ 5
    -   910          -1 9 0
    -------          ------
      1 655           2 6 5
```

This kind of problem solving has never been observed, and in the opinion of the project's diagnosticians and teachers, it never will. The natural language hypothesis is making an implausible prediction. Perhaps the hypothesis can be salvaged. In problem *c*, the student decrements a column that has already been answered. Perhaps the student would somehow appreciate that this won't have any effect on the answer, and thus not do it. However, this salvage attempt won't work for problem *d*. The decremented column is not yet answered at the time it is decremented.

These brief examples illustrate the kinds of trouble that a naive approach to natural language understanding as the source of procedure knowledge falls into. The basic problem is that natural language is terribly imprecise. The examples add the precision that the language lacks. But attending to the examples brings us back to the induction hypothesis.

SUMMARY

This chapter presented two hypotheses. The first hypothesis, a rather uncontroversial one, is that the knowledge that students acquire is schematic/procedural (at the level of a program) rather than teleologic/conceptual (at the level of the design for

a program). Both descriptive levels are logically sufficient to describe a procedure. However, if students possessed the teleology of their procedures, most impasses could be fixed by deriving a correct procedure (i.e., students would debug instead of repair). At least some students, the ones with bugs, must be lacking such teleological knowledge. Also, there is experimental evidence that some adults have no teleology for their arithmetic procedures. They either never learned it or they forgot it while somehow retaining the schematic level (program) for the procedure. All in all, it is more parsimonious to assume that students learn just the schematic level descriptions for their procedures. This implies that students' knowledge can be formalized by something like Lisp procedure or production systems. It is not necessary to use more powerful formalisms such as planning nets (VanLehn & Brown, 1980), planning calculi (Rich, 1981) or procedural nets (Sacerdoti, 1977).

The second hypothesis is that students learn inductively. They generalize examples. There are several less plausible ways that procedures could be learned: (1) Learning-by-being-told explains procedure acquisition as the conversion of an external natural language information source, e.g., from a procedural manual, into a cognitive representation of the procedure. Learning from written procedures is implausible in this domain because young students don't read well. If spoken natural language were the source of procedure descriptions, some bugs would be predicted that should not be, for they have never occurred. (2) Learning-by-analogy is used in current mathematical curricula, but in ways that would produce an overly teleological understanding of the procedural skills. If students really understood the analogies, they wouldn't develop the bugs that they do.

Of the various ways to learn procedures, only induction seems both to fit the facts of classroom life and to account for the schematic (program) level of knowledge that students appear to employ. Most importantly, the induction hypothesis can account for many bugs, ranging from 33% to 85% of the observed bugs, depending on how rigorously one tests the hypothesis.

Most of the evidence supporting the induction hypothesis came from bug data. This invites a counter-hypothesis that runs as follows: Students who learn by induction acquire bugs, whereas students who learn by being told acquire a correct procedure. However, induction is perfectly adequate for acquiring a correct procedure. In all cases where induction leads to bugs, there is an alternative path that leads to a correct procedure. Because the examples aren't rich enough to tell the student which path is correct, some will guess wrong, and end up with bugs. The others will guess correctly, and end up with a correct procedure. The induction hypothesis can't help predicting that some students acquire correct procedures. It would require extremely ad hoc stipulations to block that prediction.

An advocate of the counter-hypothesis might suggest a modified version of it: Students who learn by induction acquire either bugs or a correct procedure, whereas students who learn by being told acquire a correct procedure. The implausibility of this hypothesis should be obvious. Why is learning-by-being-told so perfect that no bugs are ever acquired? It was shown in the preceding section that when one assumes that learning-by-being-told is less than perfect, then the kinds of bugs that are produced are quite implausible. To block these bugs would require, I suppose, some rather ad hoc stipulations.

In short, we are left in a classic Occam's razor situation. One mechanism, induction, suffices to account for most of the data. Another mechanism, although intuitively plausible, accounts for only one datum: the acquisition of the correct procedure, and it must be constrained in ad hoc ways in order to do so. We can choose to believe that one empirically adequate mechanism, induction, is present. Or we can choose to believe that induction is accompanied by a second mechanism that adds nothing to the empirical coverage and may even hurt it. Occam's razor counsels us to choose the former, simpler theory.

CONCLUDING REMARKS

Induction works well as the foundation of a whole-curriculum, generative theory of bugs. However, it seems inconsistent with a fine-grained, second-by-second account of classroom learning. There is just too much natural language being used in the classroom for induction per se to accurately characterize the student's learning processes. This section contains a few speculations about what all that natural language is doing, and what the relationship of the induction hypothesis is to fine-grained, daily classroom life.

As a form of inference, induction is incomplete. In order to be useful, induction must be constrained by predilections or biases. For instance, a common bias is to prefer the *simplest* concept that is consistent with the examples. Several of the 31 hypotheses of the present theory concern the inductive biases that arithmetic students seem to have. Another prerequisite for induction is a vocabulary of primitive features with which to describe the examples. In arithmetic, the primitives used by students seem to be mostly visual and numerical (see Table 6.2). The present theory has no account for why students use exactly those primitives. It also has no account for why they hold the inductive biases that they do. There are certainly other primitives and biases that they could use, and don't. For instance, $T < B$, $T = B$ and $T > B$ are all salient features to arithmetic students. Yet they don't appear to use $T = B + 1$ or $T = 2 \times B$, relationships that are crucial in counting (Groen & Parkman, 1972; Groen & Resnick, 1977). They use the notions of leftmost column and rightmost column, but not the notion of second-column-from-the-right (i.e., the tens column). They segment problems into columns and rows, but not into diagonals or 2×2 blocks. They use primitive operations for decrementing by one and incrementing by ten, but they

don't use tens complement (7 − 3) or doubling. There are infinitely many primitives and biases that students could use, yet they seem to employ only a remarkably small set. Why?

One possibility is that the teacher's explanations somehow indicate which primitives and biases are appropriate for arithmetic. Suppose the teacher says, "You can't take 6 from 3 because it's too large." The phrase "too large" tips the student off that some relationship of relative magnitudes, e.g., $T < B$, is involved, and not, say $T = 2 \times B$. It doesn't say specifically what relationship to use, but it does narrow the set of relevant primitives down. On this view, the role of verbal explanations is to give the students a rough idea of what the procedure is. Induction fills in the details. On this view, the language is absolutely crucial. There are infinitely many numerical relationships that can hold, say, between 3 and 6. The words "too large" narrows it down to a small set. Without the language to indicate the kinds of primitives that induction should use, induction would be impossible.

Some students, when interviewed, seem to have an excellent grasp of subtraction in that they can explain, in words quite similar to those that a teacher might use, how borrowing is done and why. Yet, when they start solving problems, they have borrowing bugs (Resnick, 1982). There is a dramatic decoupling of their verbal competence and their written competence. Children aren't the only ones that show this decoupling. When adults play the Buggy game (Brown & Burton, 1978), they are required to infer a buggy procedure from examples. When they feel that they have discovered the bug, they first type in a verbal description of it, and then take a diagnostic test where they solve the problems using the procedure that they have induced. Quite frequently, the players would pass the test with flying colors, indicating that they really had induced the target procedure. However, their verbal explanations would have little recognizable relationship to the procedure. Brown and Burton (1978, pg. 169) comment on this phenomena, concentrating on the use of language during remediation:

> Another important issue concerns the relationship between the language used to describe a student's errors and its effect on what a teacher should do to remediate it. Is the language able to convey to the student what he is doing wrong? Should we expect teachers to be able to use language as the tool for correcting the buggy algorithms of students? Or should we expect teachers only to be able to understand what the bug is and attempt remediation with the student with things like manipulative math tools? The following descriptions of hypotheses given by student teachers, taken from protocols of [the Buggy game], give a good idea of how difficult it is to express procedural ideas in English. The descriptions in parentheses are [the Buggy game's] prestored explanations of the bugs.
>
> "Random errors in carryover." (Carries only when the next column in the top number is blank.)

> "If there are less digits on the top than on the bottom she adds columns diagonally." (When the top number has fewer digits than the bottom number, the numbers are left-justified and then added.)
>
> "Does not like zero in the bottom." (Zero subtracted from any number is zero.)
>
> "Child adds first two numbers correctly, then when you need to carry in the second set of digits, child adds numbers carried to bottom row then adds third set of digits diagonally finally carrying over extra digits." (The carry is written in the top number to the left of the column being carried from and is mistaken for another digit in the top number.)
>
> "Sum and carry all columns correctly until get to last column. Then takes furthest left digit in both columns and adds with digit of last carried amount. This is in the sum." (When there are an unequal number of digits in the two numbers, the columns that have a blank are filled with the left-most digit of that number.)
>
> Even when one knows what the bug is in terms of being able to mimic it, how is one going to explain it to the student having problems? Considering the above examples, it is clear that anyone asked to solve a set of problems using these explanations would, no doubt, have real trouble. One can imagine a student's frustration when the teacher offers an explanation of why he is getting problems marked wrong, and the explanation is as confused and unclear as these are.
>
> For that matter, when the correct procedure is described for the first time, could it, too, be coming across so unclearly!

For both children and players of the Buggy game, there is often a huge difference between the procedure that is performed and its verbal description. This is not, I believe, the fault of the individuals, but rather a property of natural language. Natural language is just not well suited for describing procedures. It takes considerable work to generate a good description of a procedure, and even then, ambiguities remain. This is evident not only in the student teachers' descriptions quoted above, but also in the Buggy game's descriptions of bugs, which, although painstakingly fine-tuned, are still easily misunderstood.

On the other hand, the verbal descriptions produced by the players of the Buggy game (and its authors!) probably seem quite clear to the people writing them. Most teachers probably believe that their descriptions of the arithmetic algorithms are quite lucid. Moreover, anyone else *who already knows the algorithm* would probably agree that the teacher's verbal descriptions are clear. Yet, for those who don't know the algorithm yet, viz. the students, the verbal descriptions are vague, muddied, and useless. If this is the true situation, one can easily see how the folk model of arithmetic learning stays alive. It accurately characterizes what the teacher, the parent, and anyone else who already knows arithmetic would "hear" if they visited a classroom. But from the students'

point of view, the verbal descriptions are, at worst, just noise, and at best, a hint about what kind of inductive primitives and biases to employ.

APPENDIX

Can Bugs be Expressed Using Only Visual-Numerical Features?

The demonstration consists of presenting, for each bug, a formal representation of the bug that employs only visual-numerical features. This appendix is intended to show that 85% of the observed subtraction bugs can be represented using a only visual and numerical features. This result supports the hypothesis that the bugs are learned from generalizing examples, as described in the text.

The most precise demonstration of the point would employ the bug representation used by Debuggy (Burton, 1982), because Debuggy is the final arbiter of bug existence, as the data are normally analyzed (VanLehn (1982) describes the analysis technique and the particular 1147-student sample used herein). Debuggy is used as the judge of existence because it is more uniform and reliable than human judges (VanLehn, 1982). However, Debuggy uses a rather complicated representation for bugs. The point could be made with Debuggy's representation, but it would be difficult to follow unless one were a proficient Lisp programmer. The representation presented here is much simpler and much easier to understand. However, it cannot be substituted for Debuggy's representation. It's main deficit is that it cannot accurately model multiple, co-occurring bugs. Much of the complications in Debuggy's representation are for handling the interactions of bugs when they are installed together in a procedure. This representation has no such provisions. It could probably be extended to deal with bug combining, but then it would lose some of the simplicity that makes it useful in this context.

The representation consists of two formalisms, one for correct procedures and one for bugs. First, the correct procedures' formalism will be described. A correct procedure represented by an applicative And-Or graph which, by the way, is the representation used by the generative theory of bugs discussed in the text. Table 6.3 shows an And-Or graph for the standard subtraction procedure. It furnishes a concrete illustration for explaining the formalism.

The subprocedures of subtraction, e.g. SUB/COLUMN, BORROW, etc., are represented as nodes. Each node has a definition. The definition indicates the node's arguments. In table A1, all nodes except the first one have a single argument, C, which holds the column that is the current focus of attention. The definition also indicates the node's type, which is either AND or OR. (The first node, SUBTRACTION, is special. It is neither an AND nor an OR. It is not perturbed by the bugs, so its internal structure doesn't matter.)

TABLE 6.3
And-Or Graph for the Standard Subtraction Procedure

```
(DEFINE SUBTRACTION (P)
        (for C from (FIRST/COLUMN P) to (LAST/COLUMN P)
               do (SUB/COLUMN C)))

(DEFINE SUB/COLUMN (C) OR
        (If (T=# C) then (QUIT)
        (If (B=# C) then (SHOW/TOP C)
        (If (T<B C) then (BORROW C)
        (If (TRUE) then (DIFF C)))

(DEFINE BORROW (C) AND
        (REGROUP C)
        (DIFF C))

(DEFINE REGROUP (C) AND
        (B/FROM (NEXT/COLUMN C))
        (B/INTO C))

(DEFINE B/INTO (C) OR
        (If (TRUE) then (ADD10 C))

(DEFINE B/FROM (C) OR
        (If (T=# C) then (QUIT))
        (If (T=0 C) then (BFZ C))
        (If (TRUE) then (DECR C)))

(DEFINE BFZ (C) AND
        (REGROUP C)
        (DECR C))
```

An AND node's definition has an ordered list of subgoals. They are just like subprocedure calls. They are executed in order. When the last one is finished, the AND itself is finished.

An OR node's definition has an ordered list of if-then rules. If the antecedent (the if-part) of a rule is true, then its consequent (then-part) is executed. The rules are tested in order. The first rule whose antecedent is true runs, and only one rule runs.

The primitive operations are listed in Table 6.4. It lists both the primitive operators employed by bugs as well as correct procedures. Table 6.5 (and also in the text as table 6.2) lists the primitive predicates that are used in the antecedents or rules in correct procedures and bugs.

A bug is represented by list of deletions, insertions, and substitutions. These are to be performed on the standard correct procedure, whose definition was presented earlier. The substitutions convert the standard correct procedure to an alternative, but still correct, procedure for subtraction. The deletions and insertions install the bug. The substitutions are performed first, and the deletions and

TABLE 6.4
Primitive Operators (nodes).
All Take a Column as an Argument.

ADD&TRUNCATE	Adds the column and writes the units digit of the sum in answer.
ADD10	Adds ten to the top digit in the column.
DECR	Decrements the top digit in the column by 1.
DECR/BOT	Decrements the bottom digit in the column by 1.
DIFF	Takes the absolute difference of the digits in the column and writes it in the answer.
INCR	Increments the top digit in the column by 1.
QUIT	Cause the procedure to give up on this problem.
REMEMBER/BORROW	Sets a bit to true, which is read by the predicate BORROWED.
SHOW/BOT	Writes the bottom digit of the column in the answer.
SHOW/ONE	Writes a one in the answer.
SHOW/TOP	Writes the top digit of the column in the answer.
SHOW/ZERO	Writes a zero in the answer.
WRITE10	Changes the top digit of the column to ten.
WRITE8	Changes the top digit of the column to eight.
WRITE9	Changes the top digit of the column to nine.
WRITE9/BOT	Changes the bottom digit of the column to nine.

TABLE 6.5
Primitive Predicates. All Take a column as Argument.

B=#	The bottom digit in the column is blank.
B=0	The bottom digit in the column is zero.
BORROWED	A borrow has taken place already.
DECR'ED	The top digit in the column has been decremented.
DECR'ED/BOT	The bottom digit in the column has been decremented.
IN/LAST/COLUMN	The current column is the leftmost one.
IN/NEXT/TO/LAST/COLUMN	The current column is the penultimate one.
NEXT/B=#	The bottom digit in the next column to the left is blank.
NEXT/T=0	The top digit in the next column left is zero.
NEXT/T=00	The top two digits in the next two columns to the left are zero.
NEXT/T=1	The top digit in the next column left is one.
T<B	The top digit is less than or equal to the bottom digit.
T<B	The top digit is strictly less than the bottom digit.
T>B	The top digit is strictly greater than the bottom digit.
T=B	The top digit is equal to the bottom digit.
T=B/ORIGINALLY	T=B in the original column, before decrementing occurred.
T=#	The top digit in the column is blank.
T=0	The top digit in the column is zero.
T=00	The top digit in the column and the next one to its left are zero.
T=1	The top digit in the column is one.
T=1v0	The top digit is a one or a zero.
T=9	The top digit is nine.

insertions are performed second. They are performed in parallel. Thus, if a bug's description says "Delete rule 1 of node B/FROM; Insert rule XXX in B/FROM after rule 1," then XXX will wind up exactly where rule 1 was because, in both the insertion statement and the deletion statement, the mention of rule 1 of B/FROM refers to the same rule in the correct subprocedure.

The substitutions used in the bugs are all variant of the standard subprocedure, BFZ. BFZ and its variants are listed in Table 6.6. These variants correspond to different ways of ordering the three subgoals that BFZ performs. The last variant, BFZ/2B/FROM is a little unusual, in that it calls 2B/FROM, which is a duplicate copy of B/FROM. Because 2B/FROM is called from a different place than B/FROM, perturbing the rules of 2B/FROM can give different bugs than perturbing the rules of B/FROM.

With these definitions of the representation language behind us, the bugs themselves can be presented. They are listed in two groups. The first group consists of 66 bugs that can be easily represented in this formalism. A short description and an example is provided with each bug in order to explain what it does informally. The formal description, in terms of substitutions, deletions and insertions, follows the informal one. Also, the number of occurrences of the bug is given. There are two numbers. The first is the number of students who had that bug alone, and the second is the number of students who had that bug in combination with some other bug.

TABLE 6.6
BFZ and the Variants that may be
Substituted for it.

```
(DEFINE BFZ (C) AND
        (REGROUP C)
        (DECR C))

(DEFINE BFZ/WRITE9 (C) AND
        (WRITE9 C)
        (B/FROM (NEXT/COLUMN C))

(DEFINE BFZ/3ACTS (C) AND
        (ADD10 C)
        (B/FROM (NEXT/COLUMN C))
        (DECR C))

(DEFINE BFZ/BF/A10/DECR (C) AND
        (B/FROM (NEXT/COLUMN C))
        (ADD10 C)
        (DECR C))

(DEFINE BFZ/2B/FROM (C) AND
        (ADD10 C)
        (DECR C)
        (2B/FROM (NEXT/COLUMN C)))
```

The second group of bugs listed below consists of 12 bugs that can not be easily represented in this formalism. Only their informal descriptions and occurrence frequencies are listed, of course.

The end result is that of the 78 bugs that have occurred, 66 (85%) can be represented in this formalism. Moreover, almost all the frequently appearing bugs are included among these 66. Since the formalism uses only the numerical and visual features listed in tables A2 and A3, these 66 bugs could be induced from examples.

Bugs that are Easily Represented

0 − N = 0/AFTER/BORROW occurrences: (2 3)
When a column had a 1 which changed to a 0 by a decrement, the student uses 0 − n = 0 in that column. Example: 113 − 28 = 105
Insert the rule
 If (AND (T=0 C) (DECR'ED C))
 then (SHOW/ZERO C)
in the Or node SUB/COLUMN before rule 3.

0 − N = 0/EXCEPT/AFTER/BORROW occurrences: (0 2)
When the top digit in a column is 0, the student writes 0 in the answer, ie. 0 − n = 0, unless the 0 is the result of decrementing a 1 during a borrow operation. Example: 80 − 25 = 60
Insert the rule
 If (AND (T=0 C) (NOT (DECR'ED C)))
 then (SHOW/ZERO C)
in the Or node SUB/COLUMN before rule 3.

0 − N = N/AFTER/BORROW occurrences: (1 6)
When a column had a 1 which was changed by a borrow to a 0, the student used 0 − n = n in that column. Example: 113 − 28 = 125
Insert the rule
 If (AND (T=0 C) (DECR'ED C))
 then (SHOW/BOT C)
in the Or node SUB/COLUMN before rule 3.

0 − N = N/EXCEPT/AFTER/BORROW occurrences: (4 7)
When the top digit in a column is 0, the student writes the bottom digit in the answer, ie. 0 − n = n, unless the 0 is the result of decrementing a 1 during a borrow operation. Example: 80 − 25 = 65
Insert the rule
 If (AND (T=0 C) (NOT (DECR'ED C)))
 then (SHOW/BOT C)
in the Or node SUB/COLUMN before rule 3.

1 − 1 = 0/AFTER/BORROW occurrences: (1 7)
When a column starts with a 1 on top and a 1 on the bottom and is then borrowed from, the student writes 0 in the answer for this column. Example: 113 − 18 = 105
Insert the rule
 If (AND (T=0 C) (DECR'ED C) (T=B/ORIGINALLY C))
 then (SHOW/ZERO C)
in the Or node SUB/COLUMN before rule 3.

1 − 1 = 1/AFTER/BORROW occurrences: (0 2)
If a column starts with a 1 in both the top and the bottom, and is borrowed from, the student writes 1 as the answer in the 1 over 1 column. Example: 113 − 18 = 115
Insert the rule
 If (AND (T=0 C) (DECR'ED C) (T=B/ORIGINALLY C))
 then (SHOW/BOT C)
in the Or node SUB/COLUMN before rule 3.

ALWAYS/BORROW/LEFT occurrences: (6 0)
The student always subtracts all borrows from the left-most digit in the top number. Example: 602 − 137 = 375
Delete subgoal 1 from the And node REGROUP.
Delete rule 2 from the Or node B/FROM.
Insert subgoal (B/FROM (LAST/COLUMN))
in And node REGROUP before subgoal 1.
Insert the rule
 If (T=0 C)
 then (QUIT)
in the Or node B/FROM before rule 2.

BLANK/INSTEADOF/BORROW occurrences: (0 1)
The student leaves a blank in the answer for any column which requires borrowing. Example: 208 − 113 = 15
Delete rule 3 from the Or node SUB/COLUMN.
Insert the rule
 If (T<B C)
 then (NO/OP)
in the Or node SUB/COLUMN before rule 3.

BORROW/ACROSS/SECOND/ZERO occurrences: (2 5)
Borrows from the rightmost zero by changing it to nine, but the second and following zeros are skipped over. Example: 1003 − 358 = 45
Substitute BFZ/2B/FROM for the node BFZ.
Delete rule 2 from the Or node 2B/FROM.
Insert the rule
 If (T=0 C)
 then (2B/FROM (NEXT/COLUMN C))
in the Or node 2B/FROM before rule 2.

BORROW/ACROSS/ZERO occurrences: (13 29)
When the student needs to borrow from a column whose top digit is 0, he skips that column and borrows from the next one. Example: 303 − 78 = 135
Delete rule 2 from the Or node B/FROM.
Insert the rule
 If (T=0 C)
 then (B/FROM (NEXT/COLUMN C))
in the Or node B/FROM before rule 2.

BORROW/ACROSS/ZERO/OVER/BLANK occurrences: (0 10)
When borrowing from a column which has 0 on top and a blank in the bottom, the student skips to the next column. Example: 103 − 8 = 5
Insert the rule
 If (AND (T=0 C) (B=# C))
 then (B/FROM (NEXT/COLUMN C))
in the Or node B/FROM before rule 2.

BORROW/ACROSS/ZERO/OVER/ZERO occurrences: (1 14)
When borrowing, the student skips columns which have zero on both the top and the bottom. Example: 303 − 208 = 5
Insert the rule
    If (AND (T=0 C) (B=0 C))
    then (B/FROM (NEXT/COLUMN C))
in the Or node B/FROM before rule 2.

BORROW/DIFF/0 − N = N & SMALL − LARGE = 0 occurrences: (4 0)
The student doesn't know how to borrow. If the top digit in a column is 0, the student writes the bottom digit in the answer (i.e. 0 − N = N). If the top digit is smaller than the bottom digit, then 0 is written in the answer. Example: 204 − 119 = 110
Delete rule 3 from the Or node SUB/COLUMN.
Insert the rule
    If (T=0 C)
    then (SHOW/BOT C)
in the Or node SUB/COLUMN before rule 3.
Insert the rule
    If (T<B C)
    then (SHOW/ZERO C)
in the Or node SUB/COLUMN after rule 3.

BORROW/DON'T/DECREMENT/TOP/SMALLER occurrences: (2 1)
When borrowing, the student will only decrement the top number in the next column if it is greater than or equal to the bottom number in that column. Example: 563 − 388 = 185
Insert the rule
    If (T<B C)
    then (NO/OP)
in the Or node B/FROM before rule 2.

BORROW/DON'T/DECREMENT/UNLESS/BOTTOM/SMALLER occurrences: (2 2)
When borrowing, the student will not decrement the top digit in the next column to the left unless the bottom digit in that column is smaller than the top. Example: 563 − 388 = 185
Insert the rule
    If (T<B C)
    then (NO/OP)
in the Or node B/FROM before rule 2.

BORROW/FROM/ALL/ZERO occurrences: (1 0)
When borrowing from 0, the student writes 9, but does not continue borrowing from the column to the left of the 0. If there are two 0's in a row in the top number, both are changed to 9's. Example: 203 − 98 = 205
Insert the rule
    If (AND (T=0 C) (NOT (NEXT/T=0 C)))
    then (WRITE9 C)
in the Or node B/FROM before rule 2.

BORROW/FROM/BOTTOM/INSTEADOF/ZERO occurrences: (0 1)
When borrowing from a column with 0 on top, the student borrows from the bottom digit instead of the 0 on top. In all other cases the student borrows correctly. Example: 203 − 158 = 65
Insert the rule
    If (AND (T=0 C) (NOT (B=# C)) (NOT (B=0 C)))
    then (DECR/BOT C)

in the Or node B/FROM after rule 1.
Insert the rule
    If (AND (T=0 C) (NOT (B=# C)) (B=0 C))
    then (WRITE9/BOT C) (B/FROM (NEXT/COLUMN C))
in the Or node B/FROM before rule 2.

BORROW/FROM/ONE/IS/NINE occurrences: (0 2)
When borrowing from a column which has a 1 on top, the student treats the 1 as if it were a 10. Example 113 − 58 = 145
Insert the rule
    If (T=1 C)
    then (WRITE9 C)
in the Or node B/FROM before rule 2.

BORROW/FROM/ONE/IS/TEN occurrences: (0 1)
The student writes 10 when he/she borrows from a column with a 1 in the top digit. Example: 913 − 78 = 935
Insert the rule
    If (T=1 C)
    then (WRITE10 C)
in the Or node B/FROM before rule 2.

BORROW/FROM/ZERO occurrences: (10 4)
When borrowing from a column whose top digit is 0, the student writes 9, but does not continue borrowing from the column to the left of the 0. Example: 103 − 45 = 158
Insert the rule
    If (T=0 C)
    then (WRITE9 C)
in the Or node B/FROM before rule 2.

BORROW/FROM/ZERO&LEFT/TEN/OK occurrences: (1 1)
The student changes 0 to 9 without further borrowing unless the 0 is part of a 10 in the left part of the top number. Example: 803 − 508 = 395
Insert the rule
    If (AND (T=0 C) (NOT (NEXT/T=1 C)))
    then (WRITE9 C)
in the Or node B/FROM before rule 2.

BORROW/FROM/ZERO/IS/TEN occurrences: (1 1)
When borrowing from a column with a zero on top, the student changes the zero to a ten. Example: 800 − 168 = 742
Insert the rule
    If (T=0 C)
    then (WRITE10 C)
in the Or node B/FROM before rule 2.

BORROW/INTO/ONE = TEN occurrences: (0 5)
When borrowing into a column whose top digit is 1, the student gets 10 instead of 11. Example: 321 − 89 = 221
Insert the rule
    If (T=1 C)
    then (WRITE10 C)
in the Or node B/INTO before rule 1.

BORROW/NO/DECREMENT occurrences: (10 8)
When the student needs to borrow, he/she adds 10 to the top digit of the current column without subtracting 1 from the top digit of the next column. Example: 143 − 28 = 125
Delete subgoal 1 from the And node REGROUP.

BORROW/NO/DECREMENT/EXCEPT/LAST occurrences: (4 2)
When borrowing, the student does not decrement the top digit unless he/she is working in the leftmost column. Example: 313 − 228 = 95
Insert the rule
    If (NOT (IN/LAST/COLUMN C))
    then (NO/OP)
in the Or node B/FROM before rule 2.

BORROW/ONCE/THEN/SMALLER/FROM/LARGER occurrences: (0 12)
The student subtracts the smaller digit from the larger in all columns after the first borrow. Example: 133 − 38 = 115
Delete rule 3 from the Or node SUB/COLUMN.
Insert the rule
    If (AND (T<B C) (NOT (BORROWED)))
    then (BORROW C) (REMEMBER/BORROW)
in the Or node SUB/COLUMN before rule 3.

BORROW/ONLY/ONCE occurrences: (0 1)
The student will only borrow once per problem. After that he/she will add ten to the top number if it is smaller but will not borrow one from the next column to the left. Example: 1250 − 1088 = 262
Insert the rule
    If (BORROWED)
    then (NO/OP)
in the Or node B/FROM before rule 2.
Insert subgoal (REMEMBER/BORROW)
in And node BORROW after subgoal 2.

BORROW/SKIP/EQUAL occurrences: (0 4)
When borrowing, the student skips over columns in which the top digit and the bottom digit are the same and borrows from the next column.
Example: 293 − 198 = 5
Insert the rule
    If (T=B C)
    then (B/FROM (NEXT/COLUMN C))
in the Or node B/FROM before rule 2.

BORROW/TREAT/ONE/AS/ZERO occurrences: (0 1)
When borrowing from a column that has 1 on top, the student writes 9 and continues to borrow. That is he/she treats 1 as if it were 0 because he/she doesn't like to make more 0's in the top number. Example: 313 − 158 = 145
Substitute BFZ/WRITE9 for the node BFZ.
Delete rule 2 from the Or node B/FROM.
Insert the rule
    If (T=1v0 C)
    then (BFZ C)
in the Or node B/FROM before rule 2.

CAN'T/SUBTRACT occurrences: (1 0)
The student doesn't know how to subtract at all. Example: 1003 − 87 = #

Insert the rule
    If (TRUE)
    then (QUIT)
in the Or node SUB/COLUMN before rule 1.

DECREMENT/ALL/ON/MULTIPLE/ZERO occurrences: (3 3)
When borrowing into a column which has a 0 on top from a column which has a 0, the student gets uses 9 instead of 10 for the top number. Example: 400 − 199 = 200
Substitute BFZ/WRITE9 for the node BFZ.
Insert the rule
    If (AND (T=0 C) (T=0/ORIGINALLY (NEXT/COLUMN C)))
    Then (WRITE9 C)
in the Or node B/INTO before rule 1.

DECREMENT/LEFTMOST/ZERO/ONLY occurrences: (1 0)
When borrowing from two or more zeros in the top number, the student decrements the leftmost zero but leaves all the rest as 10 and does not decrement the column to the left of the zeros. Example: 1003 − 958 = 1055
Substitute BFZ/3ACTS for the node BFZ.
Delete subgoal 3 from the And node BFZ.
Insert the rule
    If (AND (T=0 C) (NOT (NEXT/T=0 C)))
    then (WRITE 9C)
in the Or node B/FROM before rule 2.

DECREMENT/ONE/TO/ELEVEN occurrences: (0 1)
When borrowing from a column which has a one on top, the student writes 11. He/she will also continue borrowing from the next column if there is one. Example: 613 − 238 = 385
Substitute BFZ/3ACTS for the node BFZ.
Delete subgoal 3 from the And node BFZ.
Delete rule 2 from the Or node B/FROM.
Delete rule 1 from the Or node B/FROM.
Insert the rule
    If (T=1v0 C)
    then (BFZ C)
in the Or node B/FROM before rule 2.
Insert subgoal (If (T=0/ORIGINALLY C) then (DECR C))
in And node BFZ before subgoal 3.
Insert the rule
    If (T=# C)
    then (NO/OP)
in the Or node B/FROM befor rule 1.

DECREMENT/TOP/LEQ/IS/EIGHT occurrences: (1 1)
When borrowing from a column in which the top is less than or equal to the bottom, the top digit is changed to an 8.0 Example: 283 − 198 = 95
Insert the rule
    If (AND (NOT (B=# C)) (T<B C))
    then (WRITE8 C)
in the Or node B/FROM before rule 2.

DIFF/0 − N = 0 occurrences: (0 10)
Whenever the top digit in a column is 0, the student writes 0 in the answer, i.e. 0 − N = 0. Example: 140 − 21 = 120

Insert the rule
If (T=0 C)
then (SHOW/ZERO C)
in the Or node SUB/COLUMN before rule 3.

DIFF/0 − N = 0 & N − 0 = 0 occurrences: (0 3)
The student writes 0 in the answer when either the top or the bottom digit is 0. Example: 308 − 293 = 105
Insert the rule
If (OR (T=0 C) (B=0 C))
then (SHOW/ZERO C)
in the Or node SUB/COLUMN before rule 3.

DIFF/0 − N = N occurrences: (1 37)
Whenever the top digit in a column is 0, the student writes the bottom digit in the answer, i.e. 0 − N = N. Example: 140 − 21 = 121
Insert the rule
If (T=0 C)
then (SHOW/BOT C)
in the Or node SUB/COLUMN after rule 2.

DIFF/0 − N = N & N − 0 = 0 occurrences: (1 0)
The student gets 0 when subtracting 0 from anything and also gets N taken from 0 is N. Example: 302 − 192 = 290
Insert the rule
If (OR (T=0 C) (B=0 C))
then (SHOW/BOT C)
in the Or node SUB/COLUMN before rule 3.

DIFF/0 − N = N/WHEN/BORROW/FROM/ZERO occurrences: (0 2)
The student writes n in the answer when subtracting n from 0 if he/she would have to borrow from a column which contains a 0 in top. Example: 1003 − 892 = 291
Insert the rule
If (T=00 C)
then (SHOW/BOT C)
in the Or node SUB/COLUMN before rule 3.

DIFF/N − 0 = 0 occurrences: (0 2)
Whenever the bottom digit in a column is 0, the student writes 0 in the answer, i.e. N − 0 = 0. Example: 403 − 208 = 105
Insert the rule
If (B=0 C)
then (SHOW/BOT C)
in the Or node SUB/COLUMN before rule 3.

DIFF/N − N = N occurrences: (0 1)
Whenever the top digit in a column is the same as the bottom digit, the student writes that digit as the answer for that column, i.e. N − N = N. Example: 235 − 134 = 131
Insert the rule
If (T=B C)
then (SHOW/TOP C)
in the Or node SUB/COLUMN before rule 3.

DOESN'T/BORROW/EXCEPT/LAST occurrences: (0 1)
Quits instead of borrowing, unless the borrow is from the last column. Example: 345 − 120 = 225

Insert the rule
If (NOT (IN/LAST/COLUMN C))
then (QUIT)
in the Or node B/FROM before rule 2.

DON'T/DECREMENT/ZERO occurrences: (3 4)
When borrowing from a column in which the top digit is 0, the student rewrites the 0 as 10 by borrowing from the next column to the left but forgets to change 10 to 9 when he/she adds 10 to the column which originally needed the borrow. Example: 603 − 138 = 475
Substitute BFZ/3ACTS for the node BFZ.
Delete subgoal 3 from the And node BFZ.

DON'T/DECREMENT/ZERO/OVER/BLANK occurrences: (4 2)
When borrowing, the student will not decrement a zero when it is above a blank. Example: 103 − 8 = 105
Insert the rule
If (AND (T=0 C) (B=# C))
then (NO/OP)
in the Or node B/FROM before rule 2.

DON'T/DECREMENT/ZERO/UNTIL/BOTTOM/BLANK occurrences: (0 1)
The student forgets to change 10 to 9 after borrowing from a column which had a 0 on top. The exception is when 0 is part of the leftmost part of the top number then 1 is decremented correctly. Example: 304 − 259 = 55
Substitute BFZ/3ACTS for the node BFZ.
Delete subgoal 3 from the AND node BFZ.
Insert subgoal (If (B=# C) then (DECR C))
in And node BFZ before subgoal 3.

DOUBLE/DECREMENT/ONE occurrences: (1 2)
When borrowing from a column with a 1 in the top, the student changes the 1 to a 9 and continues borrowing to the left. Example: 313 − 128 = 175
Substitute BFZ/WRITE9 for the node BFZ.
Insert the rule
If (AND (T=1 C) (NOT (NEXT/B=# C)))
then (BFZ C)
in the Or node B/FROM before rule 2.

FORGET/BORROW/OVER/BLANKS occurrences: (1 3)
The student borrows correctly except he/she doesn't take 1 from the top digits that are over blanks. Example: 143 − 88 = 155
Insert the rule
If (B=# C)
then (NO/OP)
in the Or node B/FROM before rule 2.

IGNORE/LEFTMOST/ONE/OVER/BLANK occurrences: (0 6)
The student ignores the leftmost digit in the top number if it is a one and has a blank under it. Example: 188 − 33 = 55
Insert the rule
If (AND (B=# C) (T=1 C) (IN/LAST/COLUMN C))
then (QUIT)
in the Or node SUB/COLUMN before rule 2.

N − N/AFTER/BORROW/CAUSES/BORROW occurrences: (0 2)
When a column has the same number in both the top and the bottom and the digit has been

decremented by a borrow to be the same as the bottom digit, the student borrows from the next column even though they don't really need to. Example: 1073 − 168 = 8105
Insert the rule
If (AND (T=B C) (DECR'ED C) (NOT (IN/LAST/COLUMN C)))
then (BORROW C)
in the Or node SUB/COLUMN before rule 3.

N − N/CAUSES/BORROW occurrences: (1 0)
When a column has the same number on the top and bottom, the next column is decremented and 0 is written in the answer. Example: 288 − 83 = 1105
Insert the rule
If (AND (T=B C) (NOT (IN/LAST/COLUMN C)))
then (BORROW C)
in the Or node SUB/COLUMN before rule 3.

N − N = 1/AFTER/BORROW occurrences: (1 3)
The student gets 1 when subtracting *n* from *n* in a column which has been borrowed from. That is, the student knows that he/she doesn't need to borrow to subtract n from n, but he feels he must do something with the borrow, so he writes it in the answer. Example: 354 − 159 = 215
Insert the rule
If (AND (T=B/ORIGINALLY C) (DECR'ED C))
then (SHOW/ONE C)
in the Or node SUB/COLUMN before rule 3.

ONLY/DO/UNITS occurrences: (0 1)
Student only does the units column. Example: 78 − 52 = 6
Insert the rule
IF (NOT (IN/FIRST/COLUMN C))
then (QUIT)
in the Or node SUB/COLUMN before rule 2.

QUIT/WHEN/BOTTOM/BLANK occurrences: (0 5)
The student stops working the problem as soon as the bottom number runs out. Example: 178 − 59 = 19
Delete rule 2 from the Or node SUB/COLUMN.
Insert the rule
If (B=# C)
then (QUIT)
in the Or node SUB/COLUMN before rule 2.

SMALLER/FROM/LARGER occurrences: (103 12)
The student subtracts the smaller digit in a column from the larger digit regardless of which is on top. Example: 253 − 118 = 145
Delete rule 3 from the Or node SUB/COLUMN.

SMALLER/FROM/LARGER/EXCEPT/LAST occurrences: (0 3)
Student only borrows when decr is in the last column. Takes absolute difference until then. Example: 313 − 228 = 95
Delete rule 3 from the Or node SUB/COLUMN.
Insert the rule
If (AND (T<B C) (IN/NEXT/TO/LAST/COLUMN C))
then (BORROW C)
in the Or node SUB/COLUMN before rule 3.

SMALLER/FROM/LARGER/INSTEAD/OF/BORROW/FROM/ZERO occurrences (0 5)
Instead of borrowing from a column which has a 0 in the top, the student subtracts the smaller digit from the larger. Example: 101 − 56 = 55
Delete rule 3 from the Or node SUB/COLUMN.
Insert the rule
If (AND (T<B C) (NOT (NEXT/T=0 C)))
then (BORROW C)
in the Or node SUB/COLUMN before rule 3.

SMALLER/FROM/LARGER/INSTEADOF/BORROW/UNLESS/BOTTOM/SMALLER occ.: (2 5)
The student takes the absolute different instead of borrowing unless the borrow would decrement a digit that is strictly greater than the digit beneath it. Example: 300 − 39 = 339
Insert the rule
If (AND (T<B C) (OR (NEXT/B=# C) (T<B (NEXT/COLUMN C))))
then (DIFF C)
in the Or node SUB/COLUMN before rule 3.

SMALLER/FROM/LARGER/WHEN/BORROWED/FROM occurrences: (0 7)
The student subtracts the smaller digit from the larger in any column that has been borrowed from. Example: 133 − 38 = 115
Delete rule 3 from the Or node SUB/COLUMN.
Insert the rule
If (AND (T<B C) (NOT (DECR'ED C)))
then (BORROW C)
in the Or node SUB/COLUMN before rule 3.

STOPS/BORROW/AT/MULTIPLE/ZERO occurrences: (2 1)
The student doesn't borrow from two zeros in a row. He/she will just add ten to the column that needs it without decrementing anything. Example: 1003 − 358 = 655
Insert the rule
If (T=00 C)
then (NO/OP)
in the Or node B/FROM before rule 2.

STOPS/BORROW/AT/ZERO occurrences: (34 30)
The student borrows from zero incorrectly. He/she doesn't subtract 1 from the 0 (though he adds 10 correctly to the top digit of the current column). Example: 203 − 178 = 35
Delete rule 2 from the Or node B/FROM.

SUB/ONE/OVER/BLANK occurrences: (0 2)
The student subtracts one from the top number in any column with a blank in the bottom. Example: 343 − 28 = 215
Delete rule 2 from the Or node SUB/COLUMN.
Insert the rule
If (B=# C)
then (DECR C) (SHOW/TOP C)
in the Or node SUB/COLUMN before rule 2.

TOP/INSTEAD/OF/BORROW/FROM/ZERO occurrences: (0 1)
The student doesn't know how to borrow from zero. When such a borrow is required, the

student just writes the top number of the column instead. Example: 300 − 39 = 270
Insert the rule

```
If (AND (T<B C) (NEXT/T=0 C))
then (SHOW/TOP C)
```

in the Or node SUB/COLUMN before rule 3.

TREAT/TOP/ZERO/AS/TEN occurrences: (0 1)
The student treats zeros in the top number as if they were ten. Example: 109 − 81 = 128
Insert the rule

```
If (AND (T<B C) (T=0 C))
then (ADD10 C) (DIFF C)
```

in the Or node SUB/COLUMN before rule 3.

X − N = 0/AFTER/BORROW occurrences: (0 1)
In any column except the leftmost one that has been borrowed from, the student writes 0 in the answer. The leftmost column is done correctly. Example: 313 − 98 = 305
Insert the rule

```
If (AND (DECR'ED C) (NOT (IN/LAST/COLUMN C)))
then (SHOW/ZERO C)
```

in the Or node SUB/COLUMN before rule 3.

X − N = N/AFTER/BORROW occurrences: (0 1)
In any column that has been borrowed from, the student writes the bottom number in the answer. Example: 313 − 98 = 395
Insert the rule

```
If (DECR'ED C)
then (SHOW/BOT C)
```

in the Or node SUB/COLUMN before rule 3.

ZERO/INSTEADOF/BORROW occurrences: (1 0)
The student writes a 0 in any column in which borrowing is needed. Example: 140 − 28 = 120
Delete rule 3 from the Or node SUB/COLUMN.
Insert the rule

```
If (T<B C)
then (SHOW/ZERO C)
```

in the Or node SUB/COLUMN before rule 3.

Bugs That are not Easily Represented

ADD/INSTEADOF/SUB occurrences: (1 0)
The student adds instead of subtracts. Example 118 − 5 = 123

ADD/LR/DECREMENT/ANSWER/CARRY/TO/RIGHT occurrences: (1 0)
The student is adding from left to right, decrementing every column except the rightmost and carrying into every column except the leftmost. Example: 411 − 215 = 527

BORROW/ACROSS/TOP/SMALLER/DECREMENTING/TO occurrences: (2 0)
When decrementing a column in which the top is smaller than the bottom, 0 the student adds ten to the top digit, decrements the column being borrowed into and borrows from the next column to the left. Also the student skips any column which has a zero over a zero or a blank in the borrowing process. Example: 183 − 95 = 97

BORROW/ONLY/FROM/TOP/SMALLER occurrences: (1 3)
The student will try to borrow only from those columns in which the top digit is smaller

than the bottom digit. If he can't find one, then borrowing is done properly. Example 9283 − 3566 = 5627

BORROW/UNIT/DIFF occurrences: (0 1)
When the student needs to borrow, he borrows the difference between the bottom digit and the top digit of the current column. Example: 86 − 29 = 30

DECREMENT/MULTIPLE/ZEROS/BY/NUMBER/TO/LEFT occurrences: (1 1)
When borrowing from more than one zero in a row, the student decrements each zero by the number of columns to the left that had to be scanned to find a nonzero digit to decrement. Example: 8002 − 1714 = 6278

DECREMENT/MULTIPLE/ZEROS/BY/NUMBER/TO/RIGHT occurrences: (3 1)
When borrowing from more than one zero in a row, the student decrements each zero by the number of columns to its right that are borrowed from. Example: 8002 − 1714 = 6188

DON'T/WRITE/ZERO occurrences: (1 3)
The student does not write zero in the answer; he/she just leaves a blank. Example: 24 − 14 = 1

SIMPLE/PROBLEM/STUTTER/SUBTRACT occurrences: (1 0)
When the bottom number is only one digit and the top number is at least three digits, the bottom number is subtracted from every column. Example: 348 − 2 = 126

STUTTER/SUBTRACT occurrences: (2 0)
When there are blanks in the bottom number, the student subtracts the leftmost digit of the bottom number from every column that has a blank. Example: 4369 − 22 = 2147

SUB/BOTTOM/FROM/TOP occurrences: (1 0)
The student always subtracts the top digit from the bottom number. If the bottom number is smaller, he decrements the top digit and adds ten to the bottom first. If the bottom digit is zero, however, he writes the top digit in the answer. If the top digit is one greater than the bottom, he writes 9. Example : 4723 − 3065 = 9742

SUB/COPY/LEAST/BOTTOM/MOST/TOP occurrences: (1 0)
The student makes the answer by taking the most significant digits of the top and the least significant digits from the bottom number. Example: 648 − 231 = 631

ACKNOWLEDGMENTS

John Seely Brown has contributed substantially to this research, although he may not agree with all its conclusions. This research was supported by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract No. N00014-82C-0067, Contract Authority Identification No. NR667-477. Reproduction in whole or in part is permitted for any purpose of the United States Government. Approved for public release; distribution unlimited.

REFERENCES

Anderson, J. R. (1983). *The Architecture of Cognition*. Cambridge, MA: Harvard.

Anderson, J. R., Farrell, R., & Saurers, R. (1984). Learning to program in LISP. *Cognitive Science*, 8, 87–129.

Anderson, J. R., Greeno, J., Kline, P. J. & Neves, D. M. (1981). Acquisition of problem solving skill. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition*. Hillsdale, NJ: Erlbaum.
Ashlock, R. B. (1976). *Error patterns in computation*. Columbus, OH: Bell and Howell.
Badre, N. A. (1972). *Computer learning from English text*. Berkeley, CA: University of California at Berkeley, Electronic Research Laboratory. ERL-M372.
Biermann, A. W. (1972). On the inference of turing machines from sample computations. *Artificial Intelligence, 10*, 181–198.
Brown, J. S. & Burton, R. B. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science, 2*, 155–192.
Brown, J. S. & VanLehn, K. (1980). Repair Theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4*, 379–426.
Brownell, W. A. (1941). *The evaluation of learning in arithmetic*. In *Arithmetic in general education*. Washington, DC: Council of Teachers of Mathematics.
Brueckner, L. J. (1930). *Diagnostic and remedial teaching in arithmetic*. Philadelphia, PA: Winston.
Burton, R. R. (1982). Debuggy: Diagnosis of errors in basic mathematical skills. In D. H. Sleeman & J. S. Brown (Eds.), *Intelligent Tutoring Systems*. New York: Academic.
Buswell, G. T. (1926). *Diagnostic studies in arithmetic*. Chicago, IL: University of Chicago Press.
Carbonell, J. G. (1983). Derivational analogy in problem solving and knowledge acquisition. In R. S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop*. Urbana, IL: University of Illinois.
Cohen, P. R. & Feigenbaum, E. A. (1983). *The Handbook of Artificial Intelligence*. Los Altos, CA: William Kaufmann.
Cox, L. S. (1975). Diagnosing and remediating systematic errors in addition and subtraction computation. *The Arithmetic Teacher, 22*, 151–157.
Gelman, R. & Gallistel, C. R. (1978). *The child's understanding of number*. Cambridge, MA: Harvard University Press.
Greeno, J. G., Riley, M. S. & Gelman, R. (1984). Conceptual competence and children's counting. *Cognitive Psychology, 16*, 94–143.
Groen, G. J. & Parkman, J. M. (1972). A chronometric analysis of simple addition. *Psychological Review, 79*, 329–343.
Groen, G. J. & Resnick, L. B. (1977). Can preschool children invent addition algorithms? *Journal of Educational Psychology, 69*, 645–652.
Hayes-Roth, F. & McDermott, J. (1978). An interference matching technique for inducing abstractions. *Communications of the ACM, 21*, 401–411.
Lankford, F. G. (1972). *Some computational strategies of seventh grade pupils*. Charlottesville, VA: University of Virginia. ERIC document.
Mitchell, T. M., Utgoff, P. E. & Banerji, R. B. (1983). Learning problem-solving heuristics by experimentation. In R. S. Michalski, T. M. Mitchell & J. Carbonell (Eds.), *Machine Learning*. Palo Alto, CA: Tioga Press.
Resnick, L. (1982). Syntax and semantics in learning to subtract. In T. Carpeter, J. Moser & T. Romberg (Ed.), *Addition and Subtraction: A cognitive perspective*. Hillsdale, NJ: Erlbaum.
Resnick, L. B. & Omanson, S. F. (in press). Learning to understand arithmetic. In R. Glaser (Ed.), *Advances in Instructional Psychology*. Hillsdale, NJ: Erlbaum.
Rich, C. (1981). *Inspection methods in programming* (Tech. Rep. AI-TR-604). Massachusetts Institute of Technology, Artificial Intelligence Laboratory. Cambridge, MA.
Roberts, G. H. (1968). The failure strategies of third grade arithmetic pupils. *The Arithmetic Teacher, 15*, 442–446.
Sacerdoti, E. (1977). *A structure for plans and behavior*. New York: Elsevier North-Holland.
Shaw, D. J., Standiford, S. N., Klein, M. F. & Tatsuoka, K. K. (1982). *Error analysis of fraction arithmetic-selected case studies* (Tech. Report 82-2-NIE). University of Illinois. Computer-based Education Research Laboratory.

Sleeman, D. H. (1984). Basic algebra revised: A study with 14-year olds. *International Journa Man-Machine Studies*.
Sussman, G. J. (1976). *A computational model of skill acquisition*. New York: Springer Ver
Tatsuoka, K. K. & Baillie, R. (1982). *Rule space, the product space of two score component signed-number subtraction: an approach to dealing with inconsistent use of erroneous ru* (Tech. Report 82-3-ONR). University of Illinois, Computer-based Education Research Labor ry. Urbana, IL.
VanLehn, K. (1982). Bugs are not enough: Empirical studies of bugs, impasses and repair procedural skills. *The Journal of Mathematical Behavior, 3*(2), 3–71.
VanLehn, K. (1983). *Felicity conditions for human skill acquisition: Validating an AI-based th* (Tech. Report CIS-21). Xerox Palo Alto Research Center.
VanLehn, K. & Brown, J. S. (1980). Planning Nets: A representation for formalizing analogies semantic models of procedural skills. In R. E. Snow, P. A. Federico & W. E. Montague (E *Aptitude, Learning and Instruction: Cognitive Process Analyses*. Hillsdale, NJ: Erlbaum.
Winston, P. H. (1978). Learning by creating transfer frames. *Artificial Intelligence, 10*, 147–1
Winston, P. H. (1979). *Learning by understanding analogies* (Tech. Report AI-TR-520). M sachusetts Institute of Technology, Artificial Intelligence Laboratory. Cambridge, MA.
Young, R. M. & O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science, 5*, 1 177.