COGNITIVE SCIENCE 4, 379–426 (1980)

Repair Theory: A Generative Theory of Bugs in Procedural Skills

JOHN SEELY BROWN
KURT VANLEHN
Xerox Palo Alto Research Center

This paper describes a generative theory of bugs. It claims that all bugs of a procedural skill can be derived by a highly constrained form of problem solving acting on incomplete procedures. These procedures are characterized by formal deletion operations that model incomplete learning and forgetting. The problem solver and the deletion operator have been constrained to make it impossible to derive "star-bugs"—algorithms that are so absurd that expert diagnosticians agree that the alogorithm will never be observed as a bug. Hence, the theory not only generates the observed bugs, it fails to generate star-bugs.

The theory has been tested on an extensive data base of bugs for multidigit subtraction that was collected with the aid of the diagnostic systems BUGGY and DEBUGGY. In addition to predicting bug occurrence, by adoption of additional hypotheses, the theory also makes predictions about the frequency and stability of bugs, as well as the occurrence of certain latencies in processing time during testing. Arguments are given that the theory can be applied to domains other than subtraction and that it can be extended to provide a theory of procedural learning that accounts for bug acquisition. Lastly, particular care has been taken to make the theory principled so that it can not be tailored to fit any possible data.

1. INTRODUCTION

This paper presents our current efforts to form a generative theory of bugs in procedural skills. Given a procedural skill, it predicts which systematic errors or *bugs* will occur in the behavior of students learning the skill.

1.1 Background: Bugs and "Bug Stories"

Over the past few years our group has been engaged in the task of fusing computer science tools with modelling techniques from cognitive science in order to construct systems for diagnosing systematic student errors. These diagnostic systems, BUGGY and more recently DEBUGGY, have been used to analyze thousands of students' work (Brown & Burton, 1978; Burton, 1981; VanLehn & Friend, 1980) and have enabled us to construct an extensive catalogue of precisely defined systematic errors or bugs for place-value subtraction. Several other investigations of errors in arithmetic have uncovered the same "bug" phenomenon (Buswell, 1926; Brownell, 1941; Roberts, 1968; Lankford, 1972; Cox, 1975; Ashlock, 1976; Young & O'Shea, forthcoming).

A child's errors are said to be systematic if there exists a procedure that produces his erroneous answers. In nearly all cases, we have found that systematic errors are minor peturbations from the correct procedure for that skill. Precisely defined erroneous variations of a procedure are known as bugs. To say that a subject "has" a certain bug is to predict not only which problems he will answer incorrectly on a test, but also to predict the digits of those answers as well. Because an entire test's answers must be generated by a bug before we are willing to say the bug exists, there is very little chance that bugs are just "random" errors. Indeed, bugs seem to be complex, intentional actions reflecting mistaken beliefs about the skill. This is not to say that random, unsystematic errors do not occur. They do. But such errors have the appearance of "slips," where the subject did something which they did not intend to do. (Norman, 1979, argues for the widespread existence of slips in adult performance.) The subtleties of the slip/bug distinction and the data analysis techniques that were used to determine the difference are discussed in VanLehn (forthcoming). For this paper, we will assume the viability of the bug concept. Appendix 2 lists the subtraction bugs that we have observed.

BUGGY and DEBUGGY provided both a notation for precisely describing bugs and a powerful diagnostic tool which we used to sieve large amounts of student data in search of still unaccounted for errors, which could then be analyzed by hand to determine if they stemmed from a new, previously undiscovered bug. Now that several thousand student tests have been analyzed, we have reached a stage where our data base of bugs is converging. We are able to account for a substantial number of student errors and only a small number of new bugs are being discovered.

This rather extensive data base of bugs now enables a much deeper question to be investigated, namely, what is the cause of these bugs and why do just they occur and not others? Whereas our earlier effort explained a student's errors as symptoms manifested by bugs in a correct procedure, our current effort is to explain these known bugs in terms of a set of formal principles that transform a procedural skill into all of its possible buggy variants. We shall call the set of principles and the process that interprets them a *generative theory* of bugs. Using "→" to mean "explains," this can be graphically stated:

generative theory of bugs → *bugs* → *systematic errors*

The challenge of a generative theory of bugs is twofold. It must generate all the known or expected bugs for a particular skill and it must generate no others.

Many bugs appear to have a rational basis. That is, it is often easy to construct a plausible "bug story" about how a certain bug could have been acquired. We are not alone in this belief in rational genesis. Young and O'Shea (forthcoming) show that models of bugs can be constructed by editing a model of the correct skill in such a way that most of the edits have plausible, albeit informal explanations. For example, a model of a certain, observed bug is created by replacing the rule that normally decides when to borrow by a rule that says to borrow always. By similar replacements, deletions and additions, models for many common bugs can be created. However, it is not the case that every possible edit creates a model for a bug—the theorists must carefully select the edit. Hence, the fact that editing can produce models for bugs is just a tribute to the expressive power and modularity of their representation language. What is important are the bug stories that accompany and presumably constrain most of the edits they describe. For example, in describing the edit mentioned above, they say "Such a rule could result from a student's believing that borrowing is an essential part of subtraction, perhaps as a consequence of being given a series of examples in which borrowing was always necessary." Such bug stories are insightful but informal. A generative theory can be viewed as formalizing such bug stories. Indeed, before we constructed our generative theory, we constructed multiple bug stories for each of our bugs in order to discover possible patterns that would enable us to decide which of each bug's stories to choose in order to build a unified theory.

1.2 The Key Idea is Repairing Impasses

In this paper we describe our current efforts to form a generative theory of bugs, one that is capable of explaining why we found the bugs that we did and not other ones, one that is capable of explaining how bugs are caused, and most importantly, one that is capable of predicting what bugs will exist for procedural skills we have not yet analyzed.

The theory is motivated by the belief that when a student has unsuccessfully applied a procedure to a given problem, he will attempt a *repair*. Let us suppose that he is missing a fragment of some correct procedural skill, either because he never learned the fragment or maybe he forgot it. Attempting to rigorously follow the impoverished procedure will often lead to an *impasse*. That is a situation in which some current step of the procedure dictates a primitive action which the student believes cannot be carried out. For example, an impasse would follow from an attempt to decrement a zero, provided the student knows (or discovers) that the decrement primitive has as a precondition that its input argument can't be a zero. When a constraint or precondition gets violated the student, unlike a typical computer program, is not apt to just quit. Instead he will often be *inventive*, invoking his problem solving skills in an attempt to repair the impasse so that he can continue to execute the procedure, albeit in a potentially erroneous way. We believe that many bugs can best be explained as "patches" derived from repairing a procedure that has encountered an impasse while solving a particular problem.

The key idea of the generative theory is the notion of *repair*. Hence, we refer to the theory as Repair Theory. A bug's derivation in the theory has two parts. The first is a series of operations that generate an incomplete procedure, namely, a procedure that may reach an *impasse* on certain problems. The second part is a series of operations that represent the repair of the procedure so that it can proceed. It is an important assertion of the theory that these two parts are independent. That is, the kind of repair attempted depends only on the procedure and its current impasse, not on how the incomplete procedure was derived.

This paper reports on work in progress. Although a precise theory will be presented, it is not as empirically adequate as we would like. The first part of the theory, namely that which generates incomplete procedures, has known inadequacies. However, the repair generation part appears adequate. The theory is worth presenting now, in its naive form, because it raises many new distinctions that have allowed us to frame several interesting theoretical and empirical questions in a sharp, clear fashion. In particular, several predictions concerning phenomena such as processing time, bug stability and bug frequency, fall naturally out of what was originally conceived of as a theory of bug occurrence.

The paper first presents the theory and gives examples of bug derivations. The second half of the paper is a discussion of the methodology of our research along with a careful statement of its claims and their empirical support. We have tried to be very clear about what the core support is, and how it is extended through adoption of hypotheses that project the claims of the theory to become claims about other phenomena. We think such an examination of methodology is important for understanding how complex theories of complex cognitive phenomena can be evaluated and extended.

2. THE FORM OF THE GENERATIVE THEORY

As mentioned above, the generation of a bug has two parts: generation of an incomplete procedure and generation of a repair to any impasse that that procedure may encounter. Repair Theory defines the set of incomplete procedures by applying a set of *deletion principles* to a formal representation of the correct procedure. The set of repairs is defined by a set of *repair heuristics* and a set of *critics* in the following manner. When an incomplete procedure is applied to a problem and reaches an impasse, a set of repairs is performed by a *generate and test* problem solver. The set of all observable repairs is characterized by the set of repair heuristics in conjunction with a "tester" or filter which can reject certain proposed repairs based on a set of critics. That is, the heuristics suggest repairs and the critics veto some of them. Given this form, there are four major components that must be designed:

1. *A representation of the given procedural skill.* In determining this, several issues must be addressed. One concerns the representation language and its associated interpreter. Another is the representation of the physical page which bears the test problems. A very important issue concerns the structural decomposition of the skill that is to be embedded in the chosen representation language. The same procedural skill or method can often be decomposed in more than one way, which can have subtle theoretical ramifications.

2. *A set of principles for deleting fragments of the correct procedure.* These principles will determine what parts of the original skill can be deleted, thereby reflecting what parts of the procedure might become inaccessible in long term memory or may never have been learned (given the circumstances of our testing, it is often the case that students are given problems requiring subprocedures that they have not been taught yet). For example, the simplest principle might assert that any step of a procedure can be deleted; other principles might restrict the deletions to reflect a possible learning sequence of the procedure.

The next two constituents concern the elements of the generate-and-test problem solver charged with carrying out the requisite repairs.

3. *A set of repair heuristics to propose repairs.* The generator can examine the preconditions that have been violated on a primitive and propose explicit repairs based on a set of repair heuristics. Our later discussion of this component will circumvent control issues of how one repair heuristic might be initially chosen over another. Instead, we will focus on what the actual repair heuristics are and claim that *any* heuristic whose repair is not rejected by the tester must generate a bug.

4. *A set of critics to filter out some repairs.* Closely allied to the generator is the tester, which filters out those repairs that it considers to be unreasonable based on the form of the solution stemming from the proposed patch. Again, our interest here will be on the precise set of filtering conditions or "critics" and not so much on the process of invoking the critics and performing the necessary backtracking.

There are several noteworthy points to the form of this theory. The most important concerns its *composite* nature. We could have tried to account for all the known bugs in a skill by searching for a set of transformations that operate on the skill and directly produce all and only those bugs. Our theory, on the other hand, involves two parts. The first part edits the skill as dictated by a set of deletion principles which in themselves are not intended to explain all the sought after bugs. Instead, each possible edit or deleted portion generates a procedural variant which when followed (or executed) will often lead to an impasse that sets the stage for part two, the repair process. This second part uses a set of repairs to fix the procedure and allow it to continue.

It is the set of all valid repairs (i.e. those not filtered out by critics) to all possible impasses that is meant to predict the set of all possible bugs.

2.1 The Method of Investigation

Since our primary concern is to provide a *principled* account of a set of buts and to use these principles to predict bugs for skills yet to be analyzed, we invoke as little problem solving machinery as possible to account for the data. We fully recognize that there exists much more powerful problem solving models that may, in fact, better capture what a student is actually thinking while inventing a patch. We will also utilize as little of an actual process model as is possible and instead proceed under the assumption that if a rule is applicable it will be used. The trouble with invoking a process model is that it is hard to get a crisp boundary on precisely what bugs will be generated by the model since, for example, it is never certain what scheduling strategies a student might be using to select his rules or what specialization strategies he might possess for transforming a weak heuristic into a specific repair rule. We will sidestep such issues and see just how far we can get with specific repair rules that apply universally.

It is particularly important not to interpret the deletion principles in process model terms. *We are not claiming that a student knew the correct procedure, then forgot part of it. The deletion principles are a formal characterization of the set of incomplete procedures, and hence impasses, that are used and possibly repaired.* One of us (VanLehn) is constructing a learning theory which can generate that same set by simulating a student's miscomprehension of examples in the teaching sequence. We use a set of deletion principles operating on the correct skill as a precise way to characterize the set of procedures that are subject to repair while realizing that a deeper explanation for this set may be found in theories of forgetting or mislearning.

The evaluation of a generative theory rests on its ability to generate all the known bugs but to avoid predicting wild, improbable bugs. To expedite the evaluation of such theories on our data base, a "workbench" has been implemented on a computer. The workbench allows the construction of a representation language and its deletion principles, then systematically applies a deletion operator to every part of a correct procedure's representation. This generates a set of incomplete procedures, which after being repaired, are run on a highly diagnostic screening test. Their answers are analyzed by the workbench, and the set of known bugs, if any, that match each procedure's behavior are reported. Thus the workbench allows rapid comparison of representation languages, as well as help in settling fine points in the structuring of a correct procedure's representation. Our experience has been that comparison of representations has proven to be a powerful tool for zeroing in on the right skill decomposition. This topic is treated in detail in (VanLehn, 1980).

We have adopted the principle that each piece of information in the representation of a procedure must be used in the correct solution of at least one problem. This principle rules out the representation of bugs as "dead code," or information that is accessed only in the case of a deletion. With no principles governing the presence of dead code, allowing it would mean that the explanation for a bug that involved the dead code would not be completely contained within the theory, a situation we would like to avoid.

2.2 The Representation of the Procedure

The representation of procedures has an impact on all parts of the theory. Some of the issues involved are the decomposition of the skill, the level of primitives, and the language for expressing the procedure. A great deal of effort has been spent comparing various choices along these dimensions in order to find ones that maximize the expected empirical fit of the theory. The method used in this part of the investigation involved extensive use of the workbench to assess the ramifications of a simplified version of the theory on the representation. The results of this investigation are presented in a technical paper (VanLehn, 1980).

The language that was finally chosen to express procedures is a descendent of production systems (i.e. a collection of condition-action rules, c.f. Newell & Simon, 1972). There are several syntactic restrictions on the rules. Each rule's conditions must mention exactly one internal symbol (= goal). The other conditions, if any, test some features of the external world (i.e. the test problem being worked on). Each rule has exactly one action, which is either a primitive action or a subgoal. Rules are labeled for ease of reference, but the labels play no role in their interpretation.

Like production systems, rules are eligible to be run when their conditions are true. When more than one rule is eligible, the following conflict resolution strategies are applied in order until the choice is unambiguous:

Only try once: If this rule has been executed before, and the goal it matches this time is the same instantiation (token) as the goal it matched last time, then eliminate this rule from consideration.

Try special case rules first: If the conditions of this rule are a subset of the conditions of some other eligible rule (i.e. the other rule is a special case of this rule), then eliminate this rule from consideration.

Stipulated order: If there is still more than one eligible rule, then take the one that occurs first in the list of rules.

These conflict resolution strategies are found in many production system languages (McDermott & Forgy 1978).

Unlike most production systems, rules are interpreted with a stack. When the action of a rule is a goal, execution of that action pushes the current internal state onto the goal stack, and the new goal becomes current. Only rules matching the current goal are eligible to run. Goals have a type-like construction that controls when they are exited (i.e. when the stack pops). Two types are AND and OR. What the AND type means is to "exit only when all my rules have been executed." The OR type means to "exit as soon as one of my rules has been executed."

The control structure described thus far is isomorphic to that found in

And/Or graphs (except that the "try special cases first" conflict resolution strategy is not used—a minor difference). The nodes and links of AO graphs correspond, respectively, to the goals and rules of this language. However, this language provides a generalization of the And and Or types of AO graphs. The generalization of these two types is to allow a goal to exit when a given *condition* is true. This exit condition is named the "satisfaction condition" of the goal. Rules of a goal are executed in sequence until either the goal's satisfaction condition becomes true, or all the applicable rules have been tried. Note that this is not an iteration construct—an "until" loop—since a rule can only be executed once. The AND types become satisfaction conditions consisting of the constant FALSE. Since rules are executed until the satisfaction condition becomes true (which it never does for the AND) or all the rules have been tried, giving the AND goal FALSE as the satisfaction condition means that it always executes all its rules. Conversely, OR's become the constant TRUE—the goal exists after just one rule is executed. The language is named Generalized And/Or graphs (GAO graphs).

An important concept in the representation is "focus of attention." By this term, we mean where the procedure is in the problem, that is, its "current location" on the test page. Focus is strongly associated with subgoals. Focus can only be shifted by calling a subgoal—there is no assignment statement for focus. Similarly, when control returns to a goal after a subgoal that it called is finished, focus is restored as well. (In computer science terms, focus is bound locally. In fact, focus is represented syntactically by giving arguments to goals just like the arguments of procedures. For example, the SubCol goal which processes a column has three arguments named TC, BC and AC which are bound by the caller to the top, bottom and answer cells of the current column.) Thus, there are no calls to a focus shifting function to move focus back to the initiating column after a borrow is completed. Instead, focus is restored automatically when the goal stack pops. In short, the procedure's control location and its external location are maintained in exactly the same manner.

Figure 1 shows the GAO graph for a standard version of subtraction taught in the United States. Since it will be used for examples throughout this paper, it is worth a moment to explain it. The Sub goal simply initializes the column traversal to start with the units column. ColSequence is the loop across columns, expressed recursively of course since there are no loop constructions in the language. SubCol processes a column. If the bottom cell is blank, it writes the contents of the top cell in the answer (L4). If the top digit is less than the bottom digit, it calls the Borrow subgoal (L5). Otherwise, it calls the primitive Diff which writes the difference on the top and bottom digits in the answer (L6).

The Borrow goal is a conjunction of borrowing into the column originating the borrow (L8—Add10 is a primitive), and borrowing from the column (L7). By convention, borrowing-from occurs before borrowing-into. Borrowing from the next column is easy if its top digit is non-zero; the digit is decremented by the

The syntax is:

Goal (*goal's arguments*) Satisfaction Condition: *goal's satisfaction condition*
  label: {*rule's conditions*} ---> *rule's action*
  other rules for achieving the goal...

The rules for the version of subtraction used in this paper are:

```
Sub () Satisfaction Condition: TRUE
    L1:   {} --->                  (ColSequence RightmostTopCell
                                     RightmostBottomCell RightmostAnswerCell)

ColSequence (TC BC AC) Satisfaction Condition: (Blank? (Next TC))
    L2:   {} --->                  (SubCol TC BC AC)
    L3:   {} --->                  (ColSequence (Next TC) (Next BC) (Next AC))

SubCol (TC BC AC) Satisfaction Condition: (NOT (Blank? AC))
    L4:   {(Blank? BC)} --->       (WriteAns TC AC)
    L5:   {(Less? TC BC)} --->     (Borrow TC)
    L6:   {} --->                  (Diff TC BC AC)

Borrow (TC) Satisfaction Condition: FALSE
    L7:   {} --->                  (BorrowFrom (Next TC))
    L8:   {} --->                  (Add10 TC)

BorrowFrom (TC) Satisfaction Condition: TRUE
    L9:   {(Zero? TC)} --->        (BorrowFromZero TC)
    L10:  {} --->                  (Decr TC)

BorrowFromZero (TC) Satisfaction Condition: FALSE
    L11:  {} --->                  (Write9 TC)
    L12:  {} --->                  (BorrowFrom (Next TC) )
```

TC, BC and AC are variables. Their names are mneumonic for their contents, which happen to be the top, bottom and answer cells of a column.

The primitive actions and their associated preconditions are listed below. All of their arguments are cells. The actions expecting digits in certain arguments have a precondition that those cells not be blank.

Diff -- Subtracts the digit contained in its second argument from the digit contained in its first argument and writes the result in the third argument. The second argument can not be larger than the first argument.

Decr -- Subtracts one from the digit contained in its argument and writes the result back in the same cell. The input digit must be larger than zero.

WriteAns -- Writes the digit contained in its first argument in its second argument.

Add10 -- Adds ten to the digit contained in its argument and writes the result back in the same cell.

Write9 -- Writes a nine in its argument. The cell can not be blank originally.

Figure 1. A GAO graph for a standard version of subtraction

primitive Decr (L10). If the digit is zero, it is changed to a nine (L11) and BorrowFrom is called recursively (L12) to try to decrement the next column. When borrowing is completed, control returns to SubCol. Because SubCol's satisfaction condition is not true yet, L6 runs and Diff takes the column difference.

Although there are many other versions of subtraction, and several other ways to express this version in the GAO language, the decomposition of figure 1 has been found to optimize the empirical predictions of the theory.

This concludes the discussion of the representation. As mentioned earlier, there are arguments for each of the architectural features of GAO graphs. These arguments are long and subtle enough to deserve a paper of their own (VanLehn, 1980).

2.3 Deletion

Concomitant with the development of the representation, a variety of deletion principles were tried. The one that performed the best was deletion of rules.

When a rule is deleted, its sister rules will often be executed in its place, which frequently leads to an impasse. For example, when L4 of Figure 1 has been deleted, and the procedure is run on the problem

$$\begin{array}{r} 27 \\ \underline{-4} \end{array}$$

an impasse is reached in the tens column because the interpreter choses L6, the only rule that applies given that L4 is gone. Running L6 results in calling the primitive action Diff. Diff takes a column difference by taking the difference of its first two arguments' contents and writing the result in the cell pointed to by the third argument. But Diff has a precondition that neither of its arguments be blank. Since this precondition is violated when Diff is called on the tens column, the procedure is at an impasse. This impasse can be repaired in a variety of ways. For example, the procedure could simply do nothing instead of take the column difference (the "no-op" repair heuristic). Control would return from Diff, and ultimately the procedure would terminate normally leaving 3 as the answer. This way of repairing the impasse generates the bug Quit-When-Bottom-Blank. (The bug names were published (Brown and Burton, 1978) before Repair Theory was developed, so some of the names are a little inappropriate.)

Not all deletions lead to impasses. For example, when L12 is deleted, the only action that is executed during a borrow across zero is the action Write9, which scratches out a zero and writes a 9 over it. No preconditions are violated, so no repairs are needed. The resulting procedure is the bug Borrow-From-Zero. A test item answered by this procedure would look like

$$\begin{array}{r} 207 \\ \underline{128} \\ 179 \end{array}$$

Unconstrained deletion overgenerates. That is, deleting certain links leads to procedures that we have never observed, and moreover, the procedures are so counter-intuitive that we strongly believe they never will be observed. Such procedures are called ''star-bugs'' after the linguistic convention of putting a star before sentences judged to be unacceptable. Deleting L10 would generate a star-bug. The procedure resulting from the deletion never violates a precondition and hence is not repaired. It has the strange property that it only borrows correctly when the borrow is from zero—regular, ''simple'' borrows are not done. A test item solved by this star-bug would be

$$\begin{array}{r} 3075 \\ \underline{1298} \\ 2787 \end{array}$$

Intuitively, it seems implausible to delete the ordinary case while leaving the special case intact since presumably the ordinary case had been mastered some time before the special case had been taught. Indeed, in VanLehn's learning theory (forthcoming), learning the rule that is a special case of another rule, in that its conditions are a super-set of the other rule's conditions, requires the prior existence of the ordinary-case rule. Hence, a deletion blocking principle can be derived from a somewhat more plausible principle, namely, that a new rule is forgotten more readily than an old one (or, recalling that the testee's are often in the middle of the subtraction curriculum, that rules are taught in the order that they can be learned). In short, there is a basis in learning for the following principle:

A Deletion Blocking Principle

If two rules have the same goal and one is a special case of the other rule (i.e. its conditions are a superset of the more general rule's conditions), then the general rule can not be deleted unless the special case rule is deleted as well.

Since L9 is a special case of L10, the latter can not be deleted in isolation, and hence the star-bug mentioned earlier is not generated. It may seem that incorporating a special-case predicate into the theory is ad hoc and unmotivated, but this is not the case. Special case checking is needed anyway by the interpreter to sequence rules (c.f. the preceding discussion of conflict resolution).

The deletion blocking principle is in fact just the tip of the learning theory iceberg. We now believe that a better way to generate incomplete procedures would involve a complicated derivation that mimics in part the learning sequence of the subject prior to the point of testing. In this light deleting when constrained by the deletion blocking principle can be seen as an elegant, simple way to derive the incomplete procedures of subjects who never mis-learn anything, but have only learned (or remembered) part of the total algorithm. Further comments on this view will be made after discussion of the problem solver that generates repairs.

2.4 Examples of Repair Generation

We believe that a student following a procedure that specifies that a particular primitive is now to be executed but which can't be, for whatever reason, is apt to invent some repair to circumvent his current dilemma. For example, suppose he is trying to perform a column subtract with a larger number from a smaller number and he can't because there is no appropriate entry in his facts table (or because he knows he can't). What might he do? One obvious repair might be to *skip* trying to execute that primitive action and move on to the next step of the procedure. Another repair might be to simply *quit* doing the problem. Yet another repair might be to *swap the focus vertically* before calling Diff—that is, if it doesn't work taking the bottom digit from the top one, try swapping them around. And a last example might involve his being very clever and resorting to invoking the counting-based subtraction procedure that he originally used to generate or understand the facts table. For example, he might reason that if he had five apples and Tommy took seven away, then he certainly wouldn't have any apples left. Or he might count backwards from five in synchrony with counting up from seven, ending as the former becomes zero. Either way, the overall effect of reverting to the "semantics" of the facts table is to arrive at zero as the column's answer.

Examples of simple repairs. In a moment, the details of how repairs are created will be presented. But first, we will go through some examples to see how repair-generated bugs produce erroneous answers.

Suppose that rule L5 (see Figure 1) is deleted. This is the rule that says to borrow when the top digit is too small. If L5 is deleted, then L6, the rule for processing ordinary columns, will be executed on every column, including larger from smaller (LFS) columns where one ought to borrow. LFS columns violate a precondition of Diff (the action called by L6), namely that the first input number be larger than the second input number. This precondition violation is an impasse, and the problem solver is called in to repair it.

Several bugs can be generated by repairing this impasse in different ways. A natural repair is to skip the primitive whose precondition is violated. In this example, the so-called "No-op" repair heuristic (because it replaces the primitive with a null operation) generates a bug named Blank-Instead-of-Borrow. Since Diff is simply skipped when its precondition is violated, the bug does not write an answer in an LFS column.

Other repairs to the same impasse generate other bugs. If the "Quit" repair heuristic is used, then the bug is Doesn't-Borrow, because the problem is given up as soon as a LFS column is encountered. A more complicated repair heuristic is to swap cells when they are in the same column (as they are in this case). When this "Swap Vertically" repair heuristic is used, the bug Smaller-From-Larger results. This bug takes the absolute difference of each column's digits.

An even more complicated repair heuristic is used to generate the Zero-

Instead-of-Borrow bug. This bug answers all LFS columns with zero. It is generated by forgetting about the facts table and reverting to the counting procedure that underlies it. As mentioned above, there are several procedural "semantics" for the facts table that return zero when invoked with such arguments. We call the repair "Dememoize" because it is the inverse of the computer programing technique of "memorizing" a function by replacing it with a table that pairs its inputs with the outputs it would generate if it had been run.

In short, four procedures are generated by repairing the same impasse four different ways. We can summarize these procedures as:

Repairs	*Bugs*
Skip	Blank-Instead-of-Borrow
Quit	Doesn't-Borrow
Swap Vertically	Smaller-From-Larger
Dememoize	Zero-Instead-of-Borrow

These four heuristics can be used in conjunction with the deletion of L8 to generate four more procedures. L8 is the rule that adds ten to the top of the column being borrowed into. When L8 is deleted, the procedure does the decrement part of borrow correctly but fails to add ten to the top digit of the column which caused the borrow. Hence, after borrowing is done, and rule L6 is run, Diff is entered with the column in its original LFS state. Hence, the precondition that was mentioned above is violated, and an impasse occurs. Repairing this impasse with the same four repair heuristics generate four new procedures:

Repairs	*Bugs*
Skip	*Blank-With-Borrow
Quit	Doesn't-Borrow
Swap Vertically	Smaller-From-Larger-With-Borrow
Dememoize	Zero-After-Borrow

The first procedure, *Blank-With-Borrow, is a star-bug. By convention, star-bugs' names are preceded with astericks. Star-bugs have not occurred and are judged by experts to be so absurd that they will never occur. They should not be generated by the theory. In this case, the generation of *Blank-With-Borrow is blocked by a critic which filters out repairs that leave blanks in the interior of the answer. This critic also filters out the observed bug Blank-Instead-of-Borrow, which is generated by deleting L5, as illustrated just above. In order to avoid generating the star-bug, it was necessary to forgo generation of a good bug. Critics, and this tradeoff in particular, will be discussed shortly.

Examples of compound repairs. Sometimes the repair to one impasse creates a procedure that has a second impasse. Repairing the second impasse can result in bug, but on occasion the resulting procedure reaches a third impasse.

Although such a derivation could in principle go on forever, we have yet to see a bug that required more than three repairs in its derivation.

To illustrate such compound repairs, suppose that rule L9 is deleted. L9 is the rule that tells the procedure how to borrow from zero. When it is deleted, and the procedure is given a problem that requires borrowing across zero, L10 will run instead of the missing L9. Since L10 is the ordinary borrow rule, Decrement will be called with zero as its input. This violates one of its preconditions. Although many repairs lead immediately to bugs, a compound derivation can be illustrated by supposing that this impasse is repaired by the heuristic "Backup." Backup is a well known strategy in problem solving: one backs up control to the last point where a choice was made. In this case, control moves up through Borrow, which is an AND goal, and settles on SubCol. The effect of this shifting of attention is to skip the Decrement operation and the Add10 operation. In other words, instead of trying to decrement a zero, the procedure forgets about borrowing entirely and returns to examining the LFS column, which is still in its original form. Since the borrowing rule L5 has already once for this instantiation of SubCol, it can not be run again, so the ordinary column processing rule L6 is run, and Diff's precondition is violated. The four repair heuristics mentioned above now generate these procedures:

Repairs	*Bugs*
Skip	?Blank-Instead-of-Borrow-From-Zero
Quit	Borrow-Won't-Recurse
Swap Vertically	Smaller-From-Larger-Instead-of-Borrow-From-Zero
Dememoize	Zero-Instead-of-Borrow-From-Zero

Due to the critic that objects to blanks inside answers, the procedure ?Blank-Instead-of-Borrow-From-Zero is filtered out. In this case, no harm nor good is done, since it is neither an observed bug nor a star-bug. Procedures that are possible bugs (i.e. the experts do not consider them absurd enough to be star-bugs) but have not been observed are preceded by "?".

In short, with just three deletions and four repair heuristics, we have generated eleven different procedures (Doesn't-Borrow is generated two different ways), eight of which are observed bugs. An important point to notice is that repair is not always a simple process because the repair of the original impasse can create secondary impasses.

Another important point is that there is not a one-to-one correspondence between deletions and impasses: some deletions leave procedures that do not violate any preconditions. For example, if rule L7 is deleted, the resulting procedure never does the decrementing half of borrowing, but only the add-ten half. This procedure does not violate and preconditions, and hence no repairs are generated. This is the bug Borrow-No-Decrement. This repair-less generation of bugs is not common. Only two of the nine possible deletions of rules in Figure 1 lead to impasse-free procedures. Hence, of all the bugs generated from this GAO graph, the derivations of all but two involve some repair.

2.5 The Problem Solver Is Local

The architecture of the problem solver is very simple. First, a repair heuristic proposes that a certain action be done instead of the primitive action that is stuck. Second, the preconditions of the new action are checked. If a precondition is violated, the repair is unusable. Also, each critic checks to see if its condition would be violated by executing the action. If a critic is violated, the repair is rejected as well (in the discussion section, we consider what the impact of relaxing this last restriction would be on the empirical coverage of the theory). If neither a precondition nor a critic is violated, the repair occurs and the procedure derived from this repair is predicted to occur as a bug.

There are several ways that this architecture makes the problem solver weak. First, there is no ability to "look ahead" and see what the effects of a proposed repair might be some number of steps further in the problem. If there were such an ability, then the problem solver could avoid having to do secondary repairs by looking ahead far enough to see the secondary impasses such as the ones involved in the compound repair mentioned above. However, bugs involving compound repairs have occurred, so it seems the problem solver should have no ability to look ahead. In other words, the "vision" of the problem solver is local—it can only see the current state of the interpreter and the subtraction problem.

A second restriction is that the problem solver can propose only a single action. That is, the solver can't generate a repair that is a new goal, complete with new rules for satisfying it. It can propose a primitive action (see VanLehn 1980 for a discussion of the "grain size" of the primitive actions) or a "known" goal, such as Borrow. In other words, its repairs are small. Because of the local "vision" of the problem solver and the restriction that its repairs be small, the problem solver is called a *local* problem solver.

2.6 A Set of Repair Heuristics

The following set of repair heuristics seems fairly optimal. The empirical adequacy of the theory given this set will be discussed in the discussion section of the paper. We introduce the heuristics here, grouped under some suggestive headings, in order to discuss some theoretical points:

Four Weak Methods or General Purpose Heuristics

I. Escape and Flee
- a) Skip
- b) Quit
- c) Backup to last choice

II. Relocate/refocus the operation
- a) Swap Vertically

b) Refocus Left
c) Refocus Right

III. Use an operation that worked in an analogous situation/focus

a) Use Increment (from carrying) for Decrement
b) Use a top-row operation (i.e. Add10, Write9 or Decrement) to replace another top-row operation.
c) Use a column operation (i.e. SubBlank, Diff) to replace another column operation.

IV. Dememoize

The headings are meant to suggest that the repair heuristics are really just instances of more general purpose heuristics. Take the third category for example. The repair heuristic Increment for Decrement is just an instance of a general heuristic: if incrementing worked in an analogous situation, namely the "left half" of the regrouping operation of addition, then it ought to work here, in the "left half" of the regrouping operation of subtraction.

In one sense, it is quite heartening that the repair heuristics that fit best empirically can be viewed as instances of more general problem solving heuristics. It is a little easier to believe that students bring a few powerful heuristics, perhaps developed elsewhere, to subtraction than that they bring a diverse set of special purpose subtraction repair heuristics. Indeed, we propose to constrain the power of the theory by stipulating that all repair heuristics be *domain independent* in that they could plausibly be derived from general purpose problem solving strategies. Although this is not very constraining, it allows us to hope that equivalent repair heuristics will be found when the theory is applied to a new domain. There will be more discussion of this point later.

There is a tradeoff in designing a set of repair heuristics. Too few heuristics means an inability to generate some known bugs. But too many heuristics means predicting nonbugs. For example, there is a bug called Stutter-Subtract where the student reacts to nonrectangular problems by subtracting the last digit in the bottom row from top digits that are over blanks. Here is an example of Stutter-Subtract's solution to a problem:

```
7654
  31
----
4323
```

The Repair Theory analysis of this bug involves deleting the rule L4. Since this is the rule that handles blanks in the bottom row, deleting it causes Diff to be entered with a blank as its second argument, which causes a precondition violation. To generate Stutter-Subtract, we need a repair heuristic, Refocus Right. (This heuristic searches horizontally, moving rightward from the place where Diff expected to find its second argument. It stops at the first cell which is nonblank, and gives the digit to Diff as the second argument.) In short, Refocus

Right seems necessary for generating Stutter-Subtract. However Refocus Right can now be used to repair other precondition violations as well. Suppose, for example, that it is used to repair the zero precondition of Decrement. This would generate a procedure that instead of borrowing across zero would decrement the column borrowed *into*. This bug has never been observed, and seems rather implausible. In short, one has a choice of failing to generate Stutter-Subtract, or generating bugs that have not been observed.

We have chosen to accept the intuitively more plausible position that repair heuristics are just special cases of general purpose problem solving heuristics, and therefore we accept Refocus Right as a legitimate repair. To deal with the overgeneration, we propose to use a set of *critics* that test and filter out proposed repairs.

2.7 Critics

In the generate-and-test architecture of the problem solver that creates repairs, the ''test'' or filter component is driven by a collection of critics. A critic signals that something about the current state is unusual. For example, decrementing a digit that is the result of a previous decrement triggers a certain critic.

Critics, most likely, are tacitly acquired by the student's observing and abstracting the patterns that all computations appear to satisfy—especially those that were produced by the teacher working through example subtraction problems. These abstractions fall naturally into several categories. The most obvious of these concern the form of what gets written (the answer and the scratch marks). Some examples of critics in this category are:

I. Form-of-the-Writing Critics (or Constraints)

1. Don't leave a blank in the middle of the answer.
2. Don't have more than one digit per column in the answer.
3. Don't decrement a digit that is the result of a decrement.

Another category of critics has to do with the information flow. They could also be induced from examples, or they may perhaps have been deduced from more general beliefs about procedures. Some examples are:

II. Information Theoretic Critics

1. Don't change a column after its answer is written (or more generally, each operation must make a difference to the answer).
2. Don't borrow twice for the same LFS column (or more generally, avoid infinite loops).

Originally, critics were included in the theory in order to prevent overgeneration. When a procedure has been modified by repair, it may violate some critics. Such procedures can not, by hypothesis, become bugs. Thus, for example, the No-op repairs which lead to the unobserved procedures *Blank-With-Borrow and ?Blank-Instead-of-Borrow-From-Zero would be rejected because they leave a blank in the answer, thus violating the "No blanks inside the answer" critic. Critics explain why such procedures have not been observed.

After critics were incorporated in the theory, it was noted that the function of filtering repairs was also being performed by the preconditions of the primitives. That is, a repair heuristic sometimes generates an action that violates a precondition, in which case it can not be used. For example, when L4 is deleted, an attempt is made to execute Diff when the bottom cell in the column is blank. This violates the precondition that Diff's inputs must be digits. Repairing this impasse with Swap Vertically produces an attempt to execute Diff with the other input blank. Since this violates a precondition (in fact, the same precondition), the Swap Vertically repair is reject. In short, preconditions also filter repairs, just as critics do.

Since both critics and preconditions filter repairs, symmetry suggested that violating a critic ought to create an impasse just as violating a preconditions does. We tested this hypothesis, and found that indeed some bugs could be generated from impasses triggered by critics. It had escaped our attention earlier since such bugs are much less common than precondition-triggered bugs.

An example of a critic-triggered impasse involves the bug Don't-Decrement-Zero. This bug does borrowing across zero by changing the zero to a ten instead of a nine. It cannot be generated from the version of subtraction of Figure 1, but is generated instead from another common version. Whereas the version of Figure 1 does borrowing across zero by changes the zeros to nines as it moves to the left, the version needed here changes the zeros to tens as it moves left, then decrements them to nine as it moves right. (This version is in fact the one most often taught in school, and the version of Figure 1 is an optimization of it.) To generate Don't-Decrement-Zero, one deletes the rule that decrements the newly written ten to nine as the procedure moves rightward. Hence, the net effect is a procedure that changes zeros to tens as it borrows across them.

This deletion creates a procedure that does not violate any preconditions. But it does violate a critic on certain problems. When the procedure must borrow across a zero that is over another zero, such as

$$\begin{array}{r} 504 \\ \underline{108} \end{array}$$

it first changes the upper zero to ten. When it later comes to processing the tens column, it subtracts zero from ten and attempts to write ten in the answer. Since ten is two digits long, there is a critic violation.

Most of the repair heuristics are inapplicable, but two succeed in generating bugs. The first is to form an analogy to addition: the units digit of ten is written in the answer and the tens digit is carried to the next column. This generates the bug Don't-Decrement-Zero-Carrying-Answer-Overflow. The bug Don't-Decrement-Zero is generated by a hitherto unmentioned repair heuristic, namely simply ignoring the critic violation. In this case, the action generated is just to write ten as the column's answer. The "Ignore" repair heuristic can only be applied to critic-triggered impasses, which is why it wasn't mentioned earlier. Preconditions of primitives, almost by definition, can not be ignored.

The symmetry between preconditions and critics is to be built into the theory as a principle:

> *The filter/trigger symmetry*
> Any condition which can trigger impasses can filter repairs, and vice-versa.

This principle serves to constraint the class of critics. A new critic can not be added to the class unless it can function both as a repair filter and as a trigger for impasses. That is, adding a critic to block a certain star-bug may cause a different star-bug to be generated via an impasse triggered by the new critic.

2.8 Filtering Repairs versus Triggering Secondary Impasses

(This subsection discusses some technical details of the theory and can be skipped by the general reader.) Suppose a repair has just generated an action, and the action violates a critic. Since a critic can both filter repairs and trigger impasses, which will it do?

A convention is needed. The convention captures the intuition that the student will not use a repair if he can tell there is something wrong with using it *at the time he is considering whether to use it*. However, if he can successfully apply it before running into trouble, he will not backup and reject the repair but instead will repair the impasse at hand. This is an intuition about the performance of repair, but it can be captured with a locality constraint on the theory:

> *Definition of "filtering out" a repair*
> Given a repair and a procedure that has an impasse generated by running it on a certain problem, if application of the repair heuristic to the impasse generates an action whose execution would *immediately violate* a critic or a precondition, then that repair is *filtered out* as a repair of that impasse.

This definition is essentially a formal expression of the locality of the problem solver. It says that the problem solver can not look ahead to see future violations of critics and preconditions. Filtering occurs only when a violation is detected concerning the proposed action in the context of the current state. For example, the Backup repair to the "Don't decrement zero" impasse, which was discussed in section 2.4, generates an action that pops the goal stack. This popping action does not violate any critics or preconditions, so the repair is not filtered. Of course, when the stack is reset, the interpreter choses a rule that executes Diff on the original LFS column, which violates a precondition as mentioned in section 2.4. But this violation can not be "seen" by the local problem solver at the time it is considering whether to use the Backup repair. The problem solver can not, by definition, predict what the interpreter will do after the popping action is executed so it can not tell that Diff will be chosen to run.

A contrasting example is the application of Refocus Right to the same decrement-zero impasse. On certain problems, such as

```
(a)    5061
       1278
```

the repair immediately violates a critic. On problem (a), the impasse occurs when an attempt is made to decrement the zero in the hundreds column. Refocusing right generates an action that would decrement the tens column. However, the six has been decremented by a previous borrow, so this action would violate the critic "don't decrement twice." The problem sets up a situation where the use of Refocus Right causes an immediate critic violation. By definition, Refocus Right is filtered out as a repair to the decrement-zero impasse.

This example with Refocus Right illustrates a new difficulty with the distinction between filtering and triggering. On some problem, such as that displayed above, Refocus Right is filtered out as a repair for the given impasse. However, on other problems, such as

```
(b)    5069
       1278
```

it is not filtered out. The procedure generated by applying Refocus Right to the impasse created by problem (b) is not the end-product of the derivation since it will reach an impasse if it is run on problems (a). That is, we do not consider the derivation of a procedure to have ended until the procedure can be applied to any problem without reaching an impasse.

One solution to this problem is simply to let this second impasse be repaired. That is, we could allow a derivation to span more than one problem. But this entails that all possible sequences of problems be investigated because different problem sequences can generate different procedures. The sequence [b, a] generates procedures that the sequence [a, b] does not.

There is no way to avoid examining the behavior of a procedure on all possible problems since we need to determine whether it is completely derived. However, we can avoid examining all possible *sequences* of problems by adopting the following constraint:

Definition of "valid" repairs
A repair is a valid repair to an impasse only if it is not filtered out on any problem.

That is, a repair is valid only if it is universally unfiltered. To invalidate a repair, it is only necessary that *some* problem exist where the repair violates a critic.

This definition seems to entail a search through an infinite number of problems, thus raising the issue of decidability. In practice, however, the search for problems to filter the repair is not really infinite. The crucial simplification depends on the fact that preconditions and critics are local:their conditions test only the arguments of the action they filter, or in some cases the other cells in the column referred to by the arguments. Because of this locality, one can apply the repair, then examine just the column effected by the action the repair generates. Since only a few hundred combinations of digits and blanks are possible for a column, it is quite feasible to determine if there exists a column such that the action violates a critic or precondition. Provided the preconditions and critics are local, this suffices for checking all possible problems for filtering. This heuristic argument is only meant to show that the definition is usable. It is not meant to describe actual application of critics by a subject during a test situation—that would require a performance theory.

This definition of validity is not without its penalties. It causes us to reject a repair which would otherwise have generated an observed bug, Borrow-From-Bottom-Instead-of-Zero. Applying the Swap Vertically repair to the decrement-zero impasse mentioned above generates an action that attempts to decrement the bottom digit of the column. This succeeds on common problems, such as (a) below, but violates the decrement-zero precondition on rare problems such as (b), where both digits in the column borrowed from are zero.

$$\text{(a)}\quad \begin{array}{r}506\\ \underline{139}\end{array} \qquad \text{(b)}\quad \begin{array}{r}506\\ \underline{109}\end{array}$$

In a performance theory, one could regain the generation of Borrow-From-Bottom-Instead-of-Zero by replacing the universal quantifier of the definition with an assertion that the problems causing critic violation are rare enough that the student will have practiced the use of the repair on many problems before encountering a problem that causes a critic violation. The practice would "install" the repair so that the eventual critic violation would not cause its rejection, but rather would trigger a secondary impasse. However, this not a performance theory. Notions of "sufficient practice" and "installed repairs" are beyond its scope, although they play a role in work to reported in (VanLehn, forthcoming). The definition above is the best we have been able to devise within the generative framework.

2.9 Summary

This completes the description of Repair Theory. In form, it is a process that generates bugs using two mechanisms: a constrained rule deletion mechanism acting on a representation of the correct procedure for the given skill, and a local problem solver with a generate-and-test architecture that repairs the impasses that arise during execution of the improverished procedures created by deletion. There are four main "classes" or choices that determine the theory's predictions: (1) the representation used for the procedure that undergoes deletion, (2) the constraints on the deletion operator, (3) the heuristics used to generate repairs, and (4) the critics that are used for filtering repairs and triggering impasses. The most remarkable feature of the theory is perhaps its attention to principles, such as the independence of repairs and impasses. These are the topic of the next section and will be summarized at its end.

The remainder of the paper considers the empirical adequacy of the theory, and suggests some extensions. Appendix 1 details the derivations of all the procedures that are generated from the version of subtraction presented in Figure 1. Each deletion is followed by the impasses, if any, that it entails. Impasses are followed by the results of apply each of the repair heuristics. The predictions stemming from this derivation are discussed in the next section.

3. RESULTS AND DISCUSSION

It is often thought that empirical adequacy is the only measure of a theory's worth. However, it is not very difficult to get empirical adequacy if that is the only goal. We propose five criteria for evaluating this theory, and by extension other information processing theories of cognition.

3.1 Five Criteria for Theories

The first criterion is of course empirical. In the case of this theory, empirical adequacy means generating all the observed bugs and none of the star-bugs. (A star-bug is a procedure that can never occur as a bug.) Star-bugs are a necessity. Unless some bugs are labeled a priori as being unable to occur, then something which could generate all procedures (e.g. an exhaustive generator of GAO graphs) would be empirically adequate since it would generate all the bugs. It is unfortunate that we can not assume our data base is complete in that it contains all possible bugs. Many of the bugs that have shown us the most about how to structure the theory also turn out to be rather rare. To fix the current data base as an approximation of the set of all possible bugs would be to make the theory virtually immune to major revisions instigated by the data. The bug that triggers the crucial insight might be a rare one. Consequently, we must leave the door open for new bugs, and that necessitates taking the judgment of experts as an approximation of star-bugs.

The second criterion is the "tailorability" or degrees of freedom of the theory. A theory that can be tailored to fit any data base is not saying much of interest (Pylyshyn, 1980; forthcoming). For example, the ways that Repair Theory can be tailored are by adding new repair heuristics, adding new critics, or restructuring the GAO graph to get slightly different impasses. One way to limit tailorability is to make the theorist pay a heavy price for such changes. That is, any change to increase empirical adequacy must make other predictions that may or may not be correct. In short, changes have *entailments*. For example, because it is an axiom of the theory that repairs are independent of impasses, adding a new repair heuristic in order to generate a certain bug will cause the theory to predict many new procedures, namely all procedures that are derived by applying the new heuristic to all the other impasses. Some of these predicted bugs may exist, in which case the addition is good. But more often, the procedures are not bugs and in fact may be star-bugs. So, adding a repair heuristic may entail making many dubious if not incorrect predictions. By strict adherence to the principles of the theory, such as impasse-repair independence, tailorability can be limited.

Most theories of cognition are initially stated in a certain domain, in this case subtraction. This is not surprising: to study thinking, the subjects must be thinking about something. The natural question to ask at the completion of such a domain-bounded study is to what extent does the theory depend on the domain. Thus, the third criterion is the degree of domain independence of the theory. In the case of Repair Theory, this means finding out what kinds of procedural skills are such that (a) students' misunderstandings as they learn the skill are stable enough that they make systematic errors, (b) the systematic errors can be analyzed as bugs, and (c) the bugs are predicted by Repair Theory.

The fourth criterion for the theory is its ability to elucidate phenomena other than the one studied. One need not go outside the domain for such phenomena. For example, Repair Theory could perhaps explain some of the mysteries of ''bug migration.'' Bug migration is a phenomenon wherein a subject has a different bug on two tests given only a few days apart. This phenomenon appears to have a pattern to it, in that only certain bugs ''migrate'' into each other, and moreover, this migration appears to define an equivalence relation on the set of all bugs. For example, Stops-Borrow-At-Zero has been observed to migrate into Borrow-Across-Zero, and vice-versa. Now it just so happens that these two bugs can be derived from the same deletion via different repair heuristics. To explain the pattern, a ''projection hypothesis'' is adopted. In this case, the hypothesis is that bugs will migrate into each other if they are derived by different repairs to the same impasse. Note that without this hypothesis, Repair Theory has nothing to say about bug migration. Moreover, the hypothesis might not be quite right. One must have a projection hypothesis, but it could be wrong. Hence, the empirical success of the projection supports the theory, but the lack of empirical success does not refute it since the projection hypothesis could be wrong.

The fifth criterion stems from a desire to replace Repair Theory with one that is even deeper and has a strong sense of explanation to it. In that arithmetic is certainly learned rather than innate, a learning theory seems essential to a complete understanding of the cognition involved. Repair Theory is a theory of what bugs exist. Its model is a process, which gives the derivation a chronology. However, we have never asserted that this chronology has anything to do with the chronology of a subject's acquisition of a bug. The job of explaining acquisitional chronology (or perhaps difficulty) belongs to a theory of learning. Much of Repair Theory can be seen as groundwork for that learning theory. A number of principles, such as the independence of impasses and repairs, could be taken as constraints on the learning theory. In this light, Repair Theory succeeds to the extent that such principles can be abstracted from it and made available to its successor.

Having introduced the five criteria that Repair Theory will be measured by, the tradeoffs in its evaluation can be discussed in detail.

3.2 Empirical Adequacy

Repair Theory using the GAO graph, the heuristics and the critics described above generates 33 different subtraction procedures. GAO graphs for several other versions of correct subtraction have been tried, including one that does subtraction without using scratch marks, but their predictions differed only slightly from the predictions of the given GAO graph. Of the 33 procedures, 21 are well documented bugs, one is a star-bug, one is a correct procedure, and the other 10 have not been observed and hence are the theory's predictions for future bug discoveries. To give a sense of context, it is worth pointing out that when Repair Theory was first tested in September 1979, only 16 of its 33 procedures were known bugs. The current figures are from December 1979. So, in the intervening three months, 6 of the predicted bugs were actually found. We fully expect to find the other 10 predicted bugs eventually. The derivation of the 33 procedures, as well as the derivations of the procedures that are filtered out by critics, are summarized in Appendix 1.

The star-bug that is generated could be blocked, but only by adding an ad hoc deletion blocking principle. The star-bug results from deleting the tail-recursive call to ColSequence (L3) so that the procedure will only process the units column. That in itself is not unusual, but since no other links are deleted, the star-bug will borrow perfectly, even when it must borrow from zero. That a student would have mastered borrowing and yet be unable to traverse the columns is utterly unlikely. Blocking this star-bug requires a new deletion blocking principle. Unfortunately, the new principle would have to end up mentioning links from different nodes, namely L3 and L5. Since the other deletion blocking principles mention only links from the same node, this means dropping a constraint on deletion blocking principles, a move that would lead us one step closer to infinite tailorability.

In this case, such a move can be justified since the deletion mechanism is a prime candidate for replacement by a learning-based mechanism. In other words, we don't think the constraint on locality of deletion blocking principles holds universally, and since this leaves the principles virtually unconstrained (and thus infinitely tailorable), there must be something wrong with the deletion mechanism itself. We believe that it should be viewed as an instance of a much more complicated sub-theory that takes into account the fact that many of our subjects are in the middle of the subtraction curriculum. In short, the theory overgenerates by one bug, but it can be easily blocked. However, blocking the star-bug would leave the theory too unconstrained. Actually, the star-bug reveals that deletion is a fundamental inadequacy in the theory.

The theory only generates 21 of the observed 89 bugs. This is a rather severe undergeneration problem. Several extensions to the theory are possible. Their pros and cons will now be discussed.

One extension is to give the problem solver more power by equiping it with more powerful heuristics. For example, one of the bugs that can not currently be derived is Diff-0-N=N. To generate this bug, which borrows normally except when the top digit is zero, the problem solver would have to be called before borrowing occurs. This means that rule L5 would have to be deleted, so the procedure can't borrow at all. No other deletion would do. Hence, the problem solver must be powerful enough to synthesize the whole borrow procedure so that Diff-0-N=N will borrow correctly when the top digit is not 0. However, allowing such powerful heuristics abandons one of the major principles of the theory: the repair generator is a *local* problem solver. Dropping this principle allows the theory to be too easily tailored to the data. Also, this locality principle could turn out to be a very important one in a theory of learning. So, let us leave the problem solver weak, and search for another solution to undergeneration.

3.3 Interrupt Conditions and the "Periodic Table"

On considering Diff-0-N=N, the undergeneration problem isn't that the heuristics aren't powerful enough, but the opportunities to perform repairs are too infrequent. For Diff-0-N=N, the Swap repair will suffice, but the place where this repair should be triggered does not involve a precondition violation. Indeed, the needed impasse occurs just when Borrow has been entered and $T = 0$ is true (T stands for the top digit in the current column, and B stands for the bottom digit). What is needed is a way to interrupt the execution of the procedure just when the goal Borrow is set and $T = 0$ is true.

What follows is an ad hoc extension to Repair Theory. We have not incorporated it in the theory even though it doubles the empirical coverage. To do so would make the theory too easily tailored. However, it is described here as a target for explanation. If principles can be found that generate or constrain this extension, the increased coverage of this extension could be had by incorporating them in the theory.

Suppose the deletion operator is replaced with a new operator that simply affixes a condition to a goal in such a way that when the goal is entered (i.e. is called by some rule's right hand side), an impasse is generated. In other words, the operator attaches an *interrupt condition* to the goal. The interrupt condition installation operator will produce procedures that will trigger repair in hopefully just the right places.

As an attempt to avoid infinite tailorability, we will stipulate that if a condition can be an interrupt for one goal, it *must* be an interrupt for any goal. Hence, $T = 0$ must generate bugs when attached to any of the four non-terminal nodes of the GAO graph of Figure 1 (we are cheating a little here by ignoring the top two nodes, which do the main column traversal). This constraint is analogous to the stipulation that the deletion operator can delete any rule (almost—the deletion blocking principles define the exceptions); the interrupt condition installation operator can install an interrupt condition on any goal (almost).

Naturally, a list of predictions will have to be provided for use as interruption conditions. T = 0 will be one. The following set of nine conditions has been tailored, through experimentation on the workbench, to optimize the empirical coverage of the extended theory:

Interrupt Conditions for Triggering Repairs

T = 0	T = B	B = 0
T < B	T = 1	TRUE
B = #	(DoubleZero? T)	(EverDecremented?)

The only ones whose meaning might be obscure are those in the last row. B=# tests whether the bottom digit is a blank. (DoubleZero? T) is true if the top digit and the digit immediately to its left are both zero. (EverDecremented?) is true whenever at least one Decr action has happened in the current problem. To reiterate, these predicates have been chosen to fit the data; in a sense, they are just as ad hoc as a list of the bugs themselves.

Using these nine conditions as interrupts, the number of bugs generated increases to 43 from the 22 of Repair Theory. Since there are currently 89 bugs in the data base, there is still an undergeneration problem, but it is drastically reduced by the replacement of deletion with interrupt conditions. Unfortunately, we do not know how many of the procedures generated by the extended theory are star bugs, due to the way the extension was implemented on the workbench.

Figure 2 contains a "periodic table" of the procedures generated by the extended theory. (It is so named because it displays a pattern but doesn't explain it, just as the periodic table of elements does.) It demonstrates the independence of impasses and repair heuristics in a particularly graphic manner. The impasses are displayed along the vertical axis, and the repair heuristics (some of them) along the horizontal axis. Each cell of the matrix represents the procedure formed by putting the given interrupt condition on the given goal, thus establishing an impasse, and applying the given repair heuristic to that impasse. In the matrix, a cell has a "B" in it if the procedure is a known bug or "OK" if it is the correct procedure. If the procedure violates a critic, "C" appears in the cell. The empty cells of the table are procedures that no critic triggers on and yet have not been observed. The point of the periodic table is that each row has more than one entry, and each column has more than one entry. This illustrates the independence of impasses and repairs.

One problem with extending the theory with interrupt conditions is that there are no constraints on the set of interrupt condition predicates, and thus the extended theory is too easily tailored to fit the data. A second problem is that the extension degrades the theory's domain independence. The predicates of the interrupt conditions are specific to subtraction (e.g. EverDecrement? mentions

Impasses		Repairs					
when	what	Noop	Quit	Swap	Left	FAdd	FSelf
T<B	SC	B+C	B	B	B+C		
T<B	BF	B		B	C	B	
T<B	BFZ		B		C		
B = #	SC	B	B	B	B	OK	OK
B = #	B	B		OK	C	NA	
B = #	BFZ	B		C	B	B	C
T = 0	SC	C				B	B
T = 0	BF			B+C	C		B
T = 0	BFZ	B	B		B+C	B	B+C
T = 1	SC	C					B
T = 1	B	C		C	C	NA	
T = 1	BF				C		C
T = 1	BFZ	OK	OK	OK	OK	OK	OK
T = B	SC	C					B
T = B	B	OK	OK	OK	OK	NA	OK
T = B	BF				B		C
T = B	BFZ			C	B		C
EvD	SC	B	B	B	B		C
EvD	B	C		B+C	C	NA	B
EvD	BF	B			C		C
EvD	BFZ			C	C	C	C

B	= bug	SC = SubCol
C	= critic	B = Borrow
OK	= correct procedure	BF = BorrowFrom
NA	= not applicable	BFZ = BorrowFromZero

Figure 2. The "Periodic Table"

decrementing). Hence, they are not as domain independent as the repair heuristics and the critics.

A solution to both these problems is to *generate* the interrupt predicates instead of just postulating their existence. Since there are only nine predicates in the class, and the whole GAO graph is available as a potential source, it isn't very difficult to devise some operators to generate the class. In fact, *it is so easy that we can't tell which of several alternative schemes is right*, in the sense that it will produce accurate predictions when the theory is applied in a new domain. Our strategy is to look outside the theory for constraints on the choice of a scheme to generate the interrupt conditions. Hopefully, the generation of interrupt conditions will fall out of the learning mechanisms. To show how this might come about, consider the following story for how the interrupt condition T = 0 might be acquired.

Suppose that a subject (actually our model of the subject) is missing

L9—the subject hasn't learned about borrowing across zero. Suppose further that he doesn't realize that T = 0 is a precondition of decrementing. When he encounters a decrement-zero impasse, he will attempt to subtract one from zero, perhaps by counting backwards, and discovers that he can not do so. That is, he discovers the precondition. Now he not only has the opportunity to abstract and remember his repair, but also to abstract and store the newly discovered precondition. Suppose that he abstracts the precondition, but in the process, he overgeneralizes and thinks T = 0 is an exception not only to the "left half" of borrowing, but to the "right half" as well. That is, he generalizes from "you can't borrow *from* zero" to "you can't borrow *into* zero." The next time he processes a column of the form 0-N, he believes he has a precondition violation, and hence does a repair. If he applied Swap Vertically, for example, the bug Diff-0-N=N would be generated. Applying the Dememoize repair heuristic generates the bug Diff-0-N=0. So, this approach—overgeneralizing preconditions—can generate interrupt conditions.

This approach to generating interrupt conditions is being explored and will be reported in (VanLehn, forthcoming). If it is successful, the effects of the ad hoc extension discussed above can be had by making a principled extension to Repair Theory.

3.4 Acquisition of Critics

Another approach to solving the undergeneration problem that is independent of the interrupt condition extension involves the critics. The proposal is to drop the stipulation that critics *always* filter out repairs. This amounts to saying that not all subjects have all critics. The approach seems at first sight an admirable one since at least one bug, Blank-Instead-of-Borrow, is generated only to be blocked by a critic, namely "don't leave blanks in the middle of the answer." Since this critic depends only on the form of the answer, it would veto the bug no matter how the bug is generated. The only way to let this bug exist is to turn off the critic. However, making critics optional increases tailorability drastically. To block a certain star-bug, *Blank-With-Borrow, one invokes the critic. To allow the observed bug, Blank-Instead-of-Borrow, one ignores it.

The only way out of the dilemma is to try to say which subjects have which critics. This could probably be done in the context of a learning theory. The basic intuition is that since borrowing is taught early in the curriculum, it is plausible that the subject will not have abstracted the critic and hence Blank-Instead-of-Borrow can be safely generated. However, if one had gotten as far as learning how to borrow across zero, then such naivete would be extremely unlikely. Since the star-bug *Blank-With-Borrow knows how to borrow across zero, a subject who could generate it would also have the critic, and hence would filter it out. In short, it looks like a learning theory is once again necessary to increase the empirical adequacy of the theory.

3.5 Domain Independence

In the case of Repair Theory, domain independence can be tested quite clearly. One picks a new domain, say multidigit multiplication or addition of fractions. The theorist devises a collection of GAO graphs that decompose the multiplication procedure in slightly different ways. If necessary, the repair heuristics are adapted to the new domain, but they remain specializations of the same weak methods. Those critics that are domain independent, notably the information theoretic ones, can be taken over; other critics can be added later. Each GAO graph is run through the deletion/repair program and produces a collection of predicted bugs. These bugs are used to initialize DEBUGGY's data base. DEBUGGY is then run over the test results of a large number of subjects. Any subjects possessing predicted bugs will be found and their work checked by hand. After a sufficiently large number of bugs are verified, the procedures that did not occur are examined by expert (multiplication) diagnosticians to see if any star-bugs were generated. Carrying out this programme and *verifying its predictions* without major overhaul of the theory would demonstrate domain independence.

The theory has been designed to be relatively domain independent, but it has not yet been put to the test. We expect it to be able to predict the bugs that occur during the learning of mathematical skills, such as arithmetic, algebra or calculus. Representation problems in other branches of mathematics involving spatial reasoning may prove too difficult. Other procedural skills, such as operating reactors or computer systems, or controlling air traffic, are not out of the question.

There is a pretheoretic constraint on the choice of the domain, namely that it be possible to observe bugs during the skill's acquisition. Pragmatically, this means that the procedure has to be short enough that a student can solve enough test problems in a testing session to exhibit any systematic errors that may exist. Spreading the diagnosis across several testing sessions is not advisable since bugs can be highly unstable (see the discussion of bug migration below), making systematicity difficult to observe across sessions. A second problem is that devising a highly diagnostic set of test problems is extremely difficult, even for expert teachers. Some technical aids for developing diagnostic tests are discussed in (Burton, 1981).

3.6 A Projection to Bug Frequency

Repair Theory is a theory of which bugs occur. Two closely related kinds of data involve how often a bug occurs in the population (bug frequency), and how long a subject keeps a bug (bug stability). It turns out that some interesting aspects of these phenomena are qualitative ones. Hence, we can begin to speculate on how Repair Theory projects to these phenomena without getting involved in statistical calculation.

As menioned above, it is necessary to adopt a projection hypothesis to make the theory applicable to a phenomenon other than the one it was designed for. In the case of the frequency of occurrence of bugs, the obvious hypothesis to adopt is to assign each impasse and each repair heuristic a probability of occurrence. The indepencence assumption of Repair Theory, when mapped over to the frequency domain by the projection hypothesis, predicts that the frequency of a bug should be the product of the probabilities of its impasse and its repair.

However, complexities arise due to bugs derived without repair (i.e. there was no impasse) or by using multiple repairs. Adjustments would also have to be made for filtering of repairs by preconditions and critics. Also, only a dozen bugs occur frequently enough that their relative frequencies can be reliably compared. Given these difficulties, we don't expect to be able to verify the predictions in any rigorous way. Nonetheless, we have observed that the No-op, Swap and Refocus Left repairs are by far the most common, and that their relative frequency appears to be consistently higher than the other repairs across a variety of impasses. If this observation is correct, then support for the independence of impasses and repairs has been found in the frequency data. At the conclusion of the current testing program, we may be able to present some data that support this informal observation.

There is a very interesting pattern in the frequency data that has defied explanation until just recently. It involves the so-called "compound bugs" (Brown & Burton, 1978. The frequency data used below is contained in an appendix to a technical report superseded by that article. Copies of the appendix are available from the present authors. More comprehensive frequency data will be published in VanLehn & Friend, 1980). Some subjects are diagnosed as having two or more bugs at the same time. One such compound bug, for example, is Diff-0-N=N co-occurring with Borrow-Across-Zero. Compound bugs are quite common.

However, bugs do not compound independently. That is, a successful model could not be constructed wherein primitive bugs are assigned a probability of occurrence such that the probability of a compound bug's occurrence is the product of its constituents' probabilities. For example, Borrow-From-Zero is much more common in isolation than Borrow-Across-Zero. However, the compound [Borrow-From-Zero, Diff-0-N=N] is much *less* common than the compound [Borrow-Across-Zero, Diff-0-N=N]. This could not be predicted by a simple linear model of bug compounding.

Repair Theory provides such a variety of structure that it is not difficult to devise explanations for examples of nonlinear compounding. The particular example cited above, however, has defied explanation until just recently, when the search for a way to generate interrupt conditions led to the following tentative explanation.

Suppose that the story given previously for the generation of Diff-0-N=N from preconditions is correct. That is, the discovery that one can't decrement a zero is overgeneralized to become a T=0 interrupt on borrowing *into* as zero as well as borrowing *from* a zero. Hence, Diff-0-N=N is derived from a decrement-zero impasse. But the decrement-zero impass would itself have to be repaired as well. Hence, Diff-0-N=N is derived at the same time as some decrement zero bug.

This story predicts that Diff-0-N=N will occur more commonly with decrement-zero bugs than in isolation or with other bugs. From the limited frequency data on hand now, this appears to be the case. In particular, since Borrow-Across-Zero is a decrement-zero bug, but Borrow-From-Zero is not (its most common derivation is probably deletion of L12, which creates no impasses), we have an explanation for the nonlinear compounding example mentioned above. The story also predicts that decrement-zero bugs will occur much more commonly with 0-N bugs than they do in isolation—another apparently true prediction.

In short, we believe the structure of Repair Theory is sufficiently rich so that successful projections into the frequency data can be developed. The problem in such a study would be, of course, to find some way to avoid infinite tailorability. A formal projection of the theory would be a major theoretical endeavor.

3.7 A Projection to Bug Stability

The study of bug stability is essentially a study of memory. To make Repair Theory contact this new topic, we once again need a projection hypothesis. One projection hypothesis involves the concept of *a patch retention strategy*. A patch is the instantiation of a repair heuristic for a given impasse. A patch retention strategy determines when to commit repairs to memory. Let us assume that in addition to long term memory, there is some kind of memory which is sufficient to store a patch for the duration of a test (call it "intermediate term memory"). Given these two kinds of memory, a subject could have basically three stragegies for the creation and storage of patches:

Patch Retention Strategies:

1. At the first occurance of a certain impasse on a test, create a patch and use it throughout the test by sorting it in intermediate term memory (ITM). However, don't bother to put the patch in long term memory (LTM).
2. Same as the above, but put the patch in LTM.
3. At some or all of the occurances of the impasse, don't use the patch that was (perhaps) stored in ITM, but instead create a new patch, use it and perhaps store it in ITM.

If a subject always follows the first patch retention strategy, wherein he remembers a bug only for the length of the test, then we would expect to see a phenomenon we call "bug migration." When the subject is given two tests a couple of days apart (long enough to wipe out ITM but short enough that very little learning intervenes), we would observe a consistent bug on the first test and a consistent *but different* bug on the second test. The first bug has "migrated" into the second bug. Repair Theory predicts that bugs which migrate into each other will be related in that they are different repairs to the same impasse. For example, Borrow-Across-Zero would migrate into Stops-Borrow-At-Zero, but not into Borrow-From-Zero. We have anecdotal evidence for this phenomenon, and are currently conducting a pilot experiment to verify bug migration.

If a subject follows the second strategy of memorizing patches, then we would expect to find subjects with the same bug several months apart. Such subjects have been found.

If a subject follows the third patch retention strategy of changing patches in the middle of a test, we would expect to find a phenomenon called "tinkering". This means we would see a certain bug for part of a test, then a related bug for another part of the test, and so on. However, *all the bugs would have to be derivable as different repairs to the same impasse*. It is this constraint that all the bugs on the test be derived from the same impasse that separates tinkering from pure noise. Tinkering is difficult to spot because such subjects are not assigned a diagnosis by DEBUGGY since they are not consistently following a bug. However, by intensive hand examination of a small fraction of the data base, a few examples of tinkering have been found. We are in the process of designing analytic tools to help us find more.

In summary, there is some informal evidence that all three patch retention strategies exist. Their existance would be strong evidence of the veracity of Repair Theory, but the fact that the theory can already make such precise predictions confirms its worth.

New Distinctions for Bugs. The notion of a patch retention strategy suggests that the empirical phenomenon of bugs is *not* necessarily just like the computer science notion of a bug. Since bugs in computer programs are just as stable as the rest of the program, it was assumed that the rules that encode bugs in procedural skills are just as stable as the correct parts of the skill. That is, we had been blinded by our metaphor. Indeed, bug stability was such an inherent part of our computational viewpoint that our first reaction to data suggesting bug migration was shock and disbelief. We now see that a subtraction bug may be systematic and yet not stable. So, the patch retention strategy concept extends the original notion of bug that was used to describe systematic, stable behavior (strategy two) to include systematic but unstable behavior (strategy one, bug migration).

3.8 A Projection to Latency

There is a third phenomenon that could help us understand Repair Theory better. Children sometimes stop in the middle of a problem and appear to think very hard about something. Often, they are just doing the mental equivalent of counting on their fingers in order to reconstruct a subtraction fact. However, it is possible that some of the thoughtful pauses might be due to problem solving. These pauses could provide support for a strong equivalence between the interpretation/repair coroutine and the cognitive mechanisms used by students to work problems.

We propose the projection hypothesis that running the local problem solver takes more resources than running the GAO interpreter. Hence, a repair event will be signalled by a significant pause between two steps in a subject's performance. This pause will occur the first time the impasse occurs on the test (and perhaps on subsequent occurrences if the subject is tinkering), and moreover the step following the pause must be generable by some repair to that impasse. Although a timed protocol of a third grader's activity is guaranteed to have many superfluous pauses, the stringent conditions surrounding the pauses we are looking for may enable us to find them.

Apparatus has been constructed to automatically collect such protocols. If it turns out to be impossible to find such pauses, the theory would not be overturned. Instead, it is likely that the projection hypothesis is wrong, namely, the local problem solver runs just as fast as the interpreter.

3.9 Toward a Theory of Bug Acquisition

There is no doubt that learning should play an active role in a theory of bugs since bugs develop during a period when the subjects are learning the skill. However, learning is a very difficult phenenomenon to study due to the longitudinal nature of the data, questions of motivation, and the variability of the subjects' prior knowledge. And yet choices must be made about representation, primitives, process architecture and so on. It is very difficult to make these choices on an empirical basis especially given only the difficult data that a direct study of learning provides. A major service that a generative theory of bugs, which is based on comparatively clean data, could perform would be to give credibility to a set of principles that constrain a theory of skill acquisition. We believe the principles of Repair Theory do just that.

Some of these principles serve to constrain the architecture of the learning model. For example, the *locality* constraint on Repair Theory's problem solver could be adopted. The prohibitions against multi-step lookahead and creation of non-primitive nodes when adopted by the learning model could perhaps explain why skills are best taught incrementally, one step at a time. That is, the locality constraint provides a precise hypothesis about how to make the learning model incremental.

The overall architecture of Repair Theory—deletions, repairs and critics—serves to break the problem of forming a learning theory into several subproblems and to define constraints that tentative models for each must satisfy. In particular, a learning theory would need to

1. Replicate the effects of the deletion operator and the deletion blocking principles with a method for generating incomplete procedures that is based on the teaching sequence of the skill. More importantly, *this would make the method of generating incomplete procedures more general in the sense that it would not have to be revised (as the deletion blocking principles would) to be consistent with new teaching sequences.*
2. Provide a mechanism to generate interrupt conditions, or at least the bugs that interrupt conditions can generate.
3. Provide an explanation for how critics are abstracted from examples or specialized from domain independent heuristics in such a way that some of the critics can be missing early in the teaching sequence.

These subproblems have been mentioned before as critical for improving the adequacy of Repair Theory. Solving them in the context of a theory of bug acquisition will hopefully allow a unified account wherein their solutions share qualitative if not structural properties with each other and with Repair Theory's local problem solver. The attempt to unify these structures while preserving the principles of Repair Theory will lead to a learning theory that has limited tailorability and excellent support from the bug occurance data while making interesting, precise predictions about learning that can hopefully be verified without enormous longitudinal studies.

Lastly, we expect a learning theory to provide an account for most of the bugs that Repair Theory has not been able to generate. There are several bugs which have cogent, albeit informal, explanations as cases of mis-abstraction. As an example, consider the bug Always-Borrow-Left. This bug always decrements the leftmost, top digit of a problem regardless of where the column that caused the borrow is. Suppose that the subject who has this bug was tested at a point in his schooling where he had only practiced borrowing on two column problems. In such problems, the correct digit to decrement is exactly the leftmost, top digit. The subject has not yet had problems of the proper form to descriminate between the "leftmost" abstraction and the "left-adjacent" abstraction. A learning theory that learns procedures from examples could perhaps predict that such mis-abstraction bugs will occur when the learning process is incomplete. In short, it appears that the solution to the undergeneration problem of Repair Theory could best be attacked by developing a learning theory for procedures.

3.10 Summary and Concluding Remarks

The major constraints on Repair Theory (presented in Section 2) are listed below:

1. Repairs are independent of impasses. Any repair heuristic can be run on any impasse. Unless a critic or precondition filters it out, the repair will lead to a bug.

2. Critics and preconditions can both filter repairs and cause impasses. If a critic is hypothesized for one purpose, it must be usable for the other as well.

3. The problem solver can not look ahead. A repair is filtered only if the action generated by it immediately violates a critic or a precondition.

4. The problem solver can generate only primitive actions or calls to extant subgoals.

5. The rules that represents the correct procedure can not have dead code. Each rule must be used during the correct solution of at least one subtraction problem.

6. Any rule can be deleted, unless deletion is blocked by a deletion blocking principle.

7. Deletion blocking principles must be motivated by the learning sequence of the skill.

8. The repair heuristics must be specializations of domain-independent weak methods. For example, they can not mention the primitives of subtaction explicitly.

The purpose of these principles is to constrain the tailorability of the theory. Without them, the theory would have so many degrees of freedom that it could be fit to any data, and consequently would lose predictive power.

Since the theory is not able to generate all the known bugs, two extensions have been suggested. One is to replace deletion with interrupt conditions, and the other is to make critics optional. Both of these extensions are ad hoc in that they drastically increase the tailorability of the theory. Hence, they have not been incorporated in the current theory but instead are being incorporated in a learning-based theory that is being built on top of Repair Theory. The empirical results of Repair Theory and the interrupt condition extension are:

	Repair Theory	*Interrupt Conditions*
bugs	21	43
star-bugs	1	unknown
correct procedures	1	10
predicted bugs	10	unknown
total	33	180 (apprx.)

The theory can be projected to make predictions about several kinds of performance data, namely the frequency and co-occurrence of bugs, their stability between tests and even during tests, and the temporal latencies in the performance of subjects working problems.

The theory has been designed to be relatively domain independent, but as yet it has not been applied to domains other than place-value subtraction. Investigating new domains is an important direction for further research.

Perhaps one of the most important functions of a theory is to create new distinctions, new ways to look at the world. The distinctions created by this theory are based on a particularly active form of misunderstanding by the child. Since students were clearly not being taught the bugs, they must have been performing some form of invention. Overgeneralization and similar forms of mislearning just do not seem powerful enough to explain the existence of many bugs. Repair Theory formalized this intuition by making a clear distinction between incomplete procedures which are generated by mislearning or forgetting, and the inventions that are necessary to account for certain bugs.

But formalization for its own sake leads nowhere. Crucially, the formalization of repairs vs. mislearning has spawned a host of new general distinctions such as ''impass,'' ''repair heuristic,'' ''critic'' and ''patch retention strategy'' which may be of service in theories of wholly unrelated phenomena. In particular, the notion of unstable but systematic errors may prove quite useful.

Lastly, the struggle for empirical adequacy has forced us into building a whole inventory of theory formation tools. The prime tool is DEBUGGY—the data analysis tool that enables the whole investigation. Using the bugs uncovered by DEBUGGY, the workbench plays the role of the naive informant in linguistics—in a matter of days, a new version of the theory could be subjected to testing. This fast turn around time allowed us to methodically test a variety of representations and other details whose effects are very subtle.

At this stage in our research, we are struck by how different the theory is from our initial intuitive approach of spinning hypothetical ''stories'' concerning how each bug might have been produced. We quickly discovered that there were numerous possible stories for each bug. Was there a consistent basis to all those stories? This theory is a partial answer to that qyestuib,

ACKNOWLEDGMENTS

This research would have been impossible without the support of Richard Burton and Jamesine Friend. We have benefitted from discussions with many people including Richard Young, Alan Collins, Tom Moran, and Tim O'Shea. The strong encouragement of Zenon Pylyshyn, James Greeno, and Lauren Resnick is greatly appreciated. VanLehn is supported by grant N0014-78-C-0022 from the Office of Naval Research to the Learning Research and Development Center, University of Pittsburgh.

REFERENCES

Ashlock, R. B. *Error patterns in computation*. Columbus, Ohio: Bell and Howell, 1976.

Brown, J. S. & Burton, R. B. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 1978, *2*, 155–192.

Brown, J. S. & VanLehn, K. Towards a generative theory of bugs. Proceedings of the Wingspread Conference. Madison: University of Wisconsin, Wisconsin Research and Development Center for Individualized Schooling, 1980.

Brownell, W. A. The evaluation of learning in arithmetic. In *Arithmetic in General Education*. 16th Yearbook of the National Council of Teachers of Mathematics. Washington, D.C.: N.C.T.M., 1941.

Burton, R.B. DEBUGGY: Diagnosis of errors in basic mathematical skills. In D. H. Sleeman & J. S. Brown (Eds.) *Intelligent tutoring systems*. London: Academic Press, 1981.

Buswell, G. T. *Diagnostic studies in arithmetic*. Chicago: University of Chicago Press, 1926.

Cox, L. S. Diagnosing and remediating systematic errors in addition and subtraction computations. *The Arithmetic Teacher*, February 1975.

Lankford, F. G. Some computational strategies of seventh grade pupils. ERIC reports. School of Education, Virginia University, 1972.

McDermott, J. & Forgy, C. L. Production system conflict resolution strategies. In D. A. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed inference systems*. New York: Academic Press, 1978.

Newell, A. & Simon, H. A., *Human problem solving*, Englewood Cliffs, N. J.: Prentice-Hall, 1972.

Norman, D. A. Slips of the Mind and an Outline for a Theory of Action (CHIP report No. 88). La Jolla: University of California, Center for Human Information Processing, 1979.

Pylyshyn, Z. W. Computation and Cognition: Issues in the foundations of cognitive science. *The Behavioral and Brain Sciences*, 1980.

Pylyshyn, Z. W. *Computation and Cognition*, forthcoming.

Roberts, G. H. The failure strategies of third grade arithmetic pupils. *The Arithmetic Teacher*. May, 1968.

VanLehn, K. On the representation of procedures in Repair Theory. Pittsburgh: University of Pittsburgh, Learning Research and Development Laboratory technical report, 1980.

VanLehn, K. & Friend, J. Results from DEBUGGY: An analysis of systematic subtraction errors. Palo Alto California: Xerox Palo Alto Science Center technical report, 1980.

VanLehn, K. *A theory of bug acquisition*. Doctoral dissertation, Massachusetts Institute of Technology, forthcoming.

Young, R. M. & O'Shea, T. Errors in Children's Subtraction. Submitted for publication, forthcoming.

Appendix 1
Procedures Generated by the Current Version of Repair Theory

Deletions refer to the rules of the GAO graph of Figure 1. The names of repair heuristics are abbreviated. Although most of the abbreviations will be clear, two require some explanation. FAdd means to import an analogous action from addition. FSelf means to use an analogous action from subtraction. FSelf will not use an action from the subtree rooted by the deleted rule. This represents the constraint that one can not form an analogy to an action that has not yet been learned. For example, when L9 is deleted, Write9 can not be used by FSelf. Because L9 enters the subskill of borrowing across zero, which is the only part of the algorithm where Write9 is used, we can assume that Write9 has not yet been learned.

*indicates a ''star-bug'', a procedure that is so absurd that we doubt it will ever occur.
?indicates a bug that has not occurred.
Unmarked bugs have occurred.

Delete L1:	Impasse: None. Can't-Subtract
Delete L2:	Deletion is blocked by the Stipulated Orders deletion blocking principle (see VanLehn 1980).
Delete L3:	Impasse: None. *Only-Do-Units-Column
Delete L4:	Impasse: Diff called with blank cell as second argument.
Ignore:	inapplicable
Noop:	Quit-When-Bottom-Blank
Quit:	Quit-When-Bottom-Blank
Backup:	inapplicable
Swap:	filtered out by "Can't subtract blanks"
Left:	filtered out by "Can't subtract blanks"
Right:	Stutter-Subtract
FAdd:	the correct procedure is regenerated
Dememo:	inapplicable
FSelf:	inapplicable
Delete L5:	Impasse: Diff called with T<B.
Ignore:	inapplicable
Noop:	filtered out by "No blanks inside the answer"
Quit:	Doesn't-Borrow
Backup:	inapplicable
Swap:	Smaller-From-Larger
Left:	filtered out by "Can't subtract blanks" and "Don't subtract when T<B"

	Right:	filtered out by "Can't subtract blanks" and "Don't subtract when T<B"	
	FAdd:	?Add-Instead-of-Borrow	
	Dememo:	Zero-Instead-of-Borrow	
	FSelf:	(SubBlank): ?Write-Top-Instead-of-Borrow	
***Delete** L6:*		Deletion blocked by Special Case deletion blocking principle.	
***Delete** L7:*		Impasse: None.	
		Borrow-No-Decrement	
***Delete** L8:*		Impasse: Diff called with T<B after borrow has been completed.	
	Ignore:	inapplicable	
	Noop:	filtered out by "No blanks inside the answer"	
	Quit:	Doesn't-borrow	
	Backup:	inapplicable	
	Swap:	Smaller-From-Larger-With-Borrow	
	Left:	filtered out by "Can't subtract blanks" and "Don't subtract when T<B"	
	Right:	filtered out by "Can't subtract blanks" and "Don't subtract when T<B"	
	FAdd:	?Add-With-Borrow	
	Dememo:	Zero-After-Borrow	
	FSelf:	(SubBlank): ?Write-Top-With-Borrow	
***Delete** L9:*		Impasse: Decr called with T=0.	
	Ignore:	inapplicable	
	Noop:	Stops-Borrow-At-Zero	
	Quit:	Borrow-Won't-Recurse	
	Backup:	Second impasse: called with T<B. Occurs on all problems.	
		Ignore:	inapplicable
		Noop:	filtered out by "No blanks inside the answer"
		Quit:	Borrow-Won't-Recurse
		Backup:	inapplicable
		Swap:	Smaller-From-Larger-Instead-of-Borrow-From-Zero
		Left:	filtered out by "Don't subtract with T<B"
		Right:	filtered by "Can't subtract blanks" & "Don't subtract when T<B"
		FAdd:	?Add-Instead-of-Borrow-From-Zero
		Dememo:	Zero-Instead-of-Borrow-From-Zero
		FSelf:	(SubBlank): ?Write-Top-Instead-of-Borrow-From-Zero
	Swap:	filtered out by "Don't decrement zero"	
	Left:	generates the same impasses and procedures as deleting L11	
	Right:	filtered out by "Don't decrement twice", "Don't decrement blanks," and "Don't change a column after its answer is written."	
	FAdd:	Borrow-Add-Decrement-Instead-of-Zero	
	Dememo:	Stops-Borrow-At-Zero	
	FSelf:	(Add10): Second Impasse: On problems where the zero that was borrowed from is over a zero, Diff trys to write a two digit number (10) as the answer, violating the answer overflow critic. A similar impasse occurs with zeros over blanks.	
		Ignore:	Borrow-From-Zero-Is-Ten
		Noop:	filtered by "No blanks inside the answer"
		Quit:	?Borrow-From-Zero-Is-Ten-Quit-Answer-Overflow
		Backup:	inapplicable

		Swap:	inapplicable
		Left:	filtered out by "No answer overflows"
		Right:	filtered out by "No answer overflows"
		FAdd:	Borrow-From-Zero-Is-Ten-Carrying-Answer-Overflow
		Dememo:	inapplicable
		FSelf:	inapplicable
Delete L10:			Deletion is blocked by Special Case deletion blocking principle.
Delete L11:			Impasse: Deleting L11 creates a procedure that does not change zeros to nines when borrowing across zero. Consequently, a borrow is often needed in these "touched zero" columns. This borrow trys to decrement the same digit that was decremented on the first borrow, violating the "Don't decrement twice" critic.
	Ignore:		Borrow-Across-Zero
	Noop:		Borrow-Across-Zero-Touched-Zero-Is-Ten
	Quit:		?Borrow-Across-Zero-Quit-On-Touched-Zero
	Backup:		Second Impasse: Diff called with T<B.
		Ignore:	inapplicable
		Noop:	filtered by "No blanks inside the answer"
		Quit:	?Borrow-Across-Zero-Quit-On-Touched-Zero
		Backup:	inapplicable
		Swap:	Borrow-Across-Zero-Touched-0-N=N
		Left:	filtered out by "Don't subtract when T<B"
		Right:	filtered by "Can't subtract blanks" & "Don't subtract when T<B"
		FAdd:	Borrow-Across-Zero-Touched-0-N=N
		Dememo:	Borrow-Across-Zero-Touched-0-N=0
		FSelf:	(SubBlank): Borrow-Across-Zero-Touched-0-N=0
	Swap:		filtered out by "Don't decrement zero" and "Can't decrement a blank"
	Left:		filtered out by "Can't decrement a blank"
	Right:		filtered out by "Don't decrement zero"
	FAdd:		?a subtle variant of Stops-Borrow-At-Zero
	Dememo:		inapplicable
	FSelf:		(Add10): ?Borrow-Across-Zero-Add10-For-Double-Decr
Delete L12:			Impasse: None.
			Borrow-From-Zero

Appendix 2
Description of Procedural Errors (Bugs)

0-N=0/AFTER/BORROW

When a column has a 1 that was changed to a 0 by a previous borrow, the student writes 0 as the answer to that column. (914 − 486 = 508)

0-N=N/AFTER/BORROW

When a column has a 1 that was changed to a 0 by a previous borrow, the student writes the bottom digit as the answer to that column. (512 − 136 = 436)

1-1=0/AFTER/BORROW

If a column starts with 1 in both top and bottom and is borrowed from, the student writes 0 as the answer to that column. (812 − 518 = 304)

1-1=1/AFTER/BORROW

If a column starts with 1 in both top and bottom and is borrowed from, the student writes 1 as the answer to that column. (812 − 518 = 314)

ADD/BORROW/CARRY/SUB

The student adds instead of subtracting but he subtracts the carried digit instead of adding it. (54 − 38 = 72)

ADD/BORROW/DECREMENT

Instead of decrementing the student adds 1, carrying to the next column if necessary.

$$\begin{array}{r} 863 \\ -134 \\ \hline 749 \end{array} \qquad \begin{array}{r} 893 \\ -104 \\ \hline 809 \end{array}$$

ADD/BORROW/DECREMENT/WITHOUT/CARRY

Instead of decrementing the student adds 1. If this addition results in 10 the student does not carry but simply writes both digits in the same space.

$$\begin{array}{r} 863 \\ -134 \\ \hline 749 \end{array} \qquad \begin{array}{r} 8\ 93 \\ -1\ 04 \\ \hline 7109 \end{array}$$

ADD/INSTEAD/OF/SUB

The student adds instead of subtracting. (32 − 15 = 47)

ADD/NOCARRY/INSTEADOF/SUB

The student adds instead of subtracting. If carrying is required he does not add the carried digit. (47 − 25 = 62)

ALWAYS/BORROW

The student borrows in every column regardless of whether it is necessary. (488 − 299 = 1159)

ALWAYS/BORROW/LEFT

The student borrows from the leftmost digit instead of borrowing from the digit immediately to the left. (733 − 216 = 427)

BLANK/INSTEADOF/BORROW

When a borrow is needed the student simply skips the column and goes on to the next (425 − 283 = 22)

BORROW/ACROSS/TOP/SMALLER/DECREMENTING/TO

When decrementing a column in which the top is smaller than the bottom, the student adds 10 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 0 over a 0 or a blank in the borrowing process.

```
 183     513
− 95    −268
----    ----
  97     254
```

BORROW/ACROSS/ZERO

When borrowing across a 0, the student skips over the 0 to borrow from the next column. If this causes him to have to borrow twice he decrements the same number both times.

```
 904     904
−  7    −237
----    ----
 807     577
```

BORROW/ACROSS/ZERO/OVER/BLANK

When borrowing across a 0 over a blank, the student skips to the next column to decrement. (402 − 6 = 306)

BORROW/ACROSS/ZERO/OVER/ZERO

Instead of borrowing across a 0 that is over a 0, the student does not change the 0 but decrements the next column to the left instead. (802 − 304 = 308)

BORROW/ACROSS/ZERO/TOUCHED/0-N=0

Instead of borrowing across a 0, the student does not change the 0 but decrements the next column on the left instead. Also, if borrowing is needed in a column headed by a zero that should have been changed, the student writes zero in the answer instead. (802 − 324 = 508)

BORROW/ACROSS/ZERO/TOUCHED/0-N=N

Instead of borrowing across a 0, the student does not change the 0 but decrements the next column to the left instead. Also, if borrowing is needed in a column headed by a zero that should have been changed, the student writes the bottom digit in the answer instead. (802 − 324 = 528)

BORROW/ACROSS/ZERO/TOUCHED/ZERO/IS/TEN

Instead of borrowing across a 0, the student does not change the 0 but decrements the next column to the left instead. Also, if borrowing is needed in a column headed by a zero that should have been changed, the student adds ten to the zero but does no decrementing. (802 − 324 = 588)

BORROW/ADD/DECREMENT/INSTEADOF/ZERO

Instead of borrowing across a 0, the student changes the 0 to 1 and doesn't decrement any column to the left. (307 − 108 = 219)

BORROW/ADD/IS/TEN

The student changes the number that causes the borrow into 10 instead of adding 10 to it. (83 − 29 = 51)

BORROW/DECREMENTING/TO/BY/EXTRAS

When there is a borrow across 0's, the student does not add 10 to the column he is doing but instead adds 10 minus the number of 0's borrowed across.

```
 308     3008
−139    −1359
----    -----
 168     1647
```

BORROW/DIFF/0-N=N&SMALL-LARGE=0

The student doesn't borrow. For columns of the form 0 − N he writes N as the answer. Otherwise he writes 0. (304 − 179 = 270)

BORROW/DON'T/DECREMENT/TOP/SMALLER

The student will not decrement a column if the top number is smaller than the bottom number.

732	732
−484	−434
258	298
Wrong	Correct

BORROW/DON'T/DECREMENT/UNLESS/BOTTOM/SMALLER

The student will not decrement a column unless the bottom number is smaller than the top number.

732	732
−484	−434
258	308

BORROW/FROM/ALL/ZERO

Instead of borrowing across 0's, the student changes all the 0's to 9's but does not continue borrowing from the column to the left. (3006 − 1807 = 2199)

BORROW/FROM/BOTTOM

The student borrows from the bottom row instead of the top one.

87	827
−28	−208
79	839

BORROW/FROM/BOTTOM/INSTEAD/OF/ZERO

When borrowing from a column of the form 0 − N, the student decrements the bottom number instead of the 0.

608	108
−249	− 49
379	79

BORROW/FROM/LARGER

When borrowing, the student decrements the larger digit in the column regardless of whether it is on the top or the bottom. (872 − 294 = 598)

BORROW/FROM/ONE/IS/NINE

When borrowing from a 1, the student treats the 1 as if it were 10, decrementing it to a 9. (316 − 139 = 267)

BORROW/FROM/ONE/IS/TEN

When borrowing from a 1, the student changes the 1 to 10 instead of to 0. (414 − 277 = 237)

BORROW/FROM/ZERO

Instead of borrowing across a 0, the student changes the 0 to 9 but does not continue borrowing from the column to the left.

306	3006	103
−187	−1807	− 45
219	1299	158

BORROW/FROM/ZERO/IS/TEN

When borrowing across 0, the student changes the 0 to 10 and does not decrement any digit to the left. (604 − 235 = 479)

BORROW/FROM/ZERO/IS/TEN/CARRYING/ANSWER/OVERFLOW

When borrowing across 0, the student changes the 0 to 10 and does not decrement any digit to the left. However, if the newly created 10 is over zero, the student carries instead of trying to write ten in the answer. (604 − 205 = 509)

BORROW/FROM/ZERO&LEFT/OK

Instead of borrowing across a 0, the student changes the 0 to 9 but does not continue borrowing from the column to the left. However if the digit to the left of the 0 is over a blank then the student does the correct thing.

306	3006	103	203
− 187	− 1807	− 45	− 45
219	1299	58	158
Wrong	Wrong	Correct	Correct

BORROW/FROM/ZERO&LEFT/TEN/OK

Instead of borrowing across a 0, the student changes the 0 to 9 but does not continue borrowing from the column to the left. However if the digit to the left of the 0 is a 1 over a blank then the student does the correct thing.

306	103	203
− 187	− 45	− 45
219	58	258
Wrong	Correct	Wrong

BORROW/IGNORE/ZERO/OVER/BLANK

When borrowing across a 0 over a blank, the student treats the column with the zero as if it weren't there.

505	508
− 7	− 7
4 8	501
Wrong	Correct

BORROW/INTO/ONE=TEN

When a borrow is caused by a 1, the student changes the 1 to a 10 instead of adding 10 to it. (71 − 38 = 32)

BORROW/NO/DECREMENT

When borrowing the student adds 10 correctly but doesn't change any column to the left. (62 − 44 = 28)

BORROW/ONCE/THEN/SMALLER/FROM/LARGER

The student will borrow only once per exercise. From then on he subtracts the smaller from the larger digit in each column regardless of their positions. (7127 − 2389 = 4278)

BORROW/ONCE/WITHOUT/RECURSE

The student will borrow only once per problem. After that, if another borrow is required the student adds the 10 correctly but does not decrement. If there is a borrow across a 0 the student changes the 0 to 9 but does not decrement the digit to the left of the 0.

535	408
− 278	− 239
357	269

BORROW/ONLY/FROM/TOP/SMALLER

When borrowing, the student tries to find a column in which the top number is smaller than the bottom. If there is one he decrements that, otherwise he borrows correctly. (9283 − 3566 = 5627)

BORROW/ONLY/ONCE

When there are several borrowers, the student decrements only with the first borrower. (535 − 278 = 357).

BORROW/SKIP/EQUAL

When decrementing, the student skips over columns in which the top digit and the bottom digit are the same. (923 − 427 = 406)

BORROW/TEN/PLUS/NEXT/DIGIT/INTO/ZERO

When a borrow is caused by a 0 the student does not add 10 correctly. What he does instead is add 10 plus the digit in the next column to the left. He will give answers like this: (50 − 38 = 17)

BORROW/TREAT/ONE/AS/ZERO

When borrowing from 1, the student treats the 1 as if it were 0; that is, he changes the 1 to 9 and decrements the number to the left of the 1. (313 − 159 = 144)

BORROW/UNIT/DIFF

The student borrows the difference between the top digit and the bottom digit of the current column. In other words, he borrows just enough to do the subtraction, which then always results in 0. (86 − 29 = 30)

BORROW/WON'T/RECURSE

Instead of borrowing across a 0, the student stops doing the exercise. (8035 − 2662 = 3)

BORROWED/FROM/DON'T/BORROW

When there are two borrows in a row the student does the first borrow correctly but with the second borrow he does not decrement (he does add 10 correctly). (143 − 88 = 155)

CAN'T/SUBTRACT

The student skips the entire problem. (8 − 3 =)

DECREMENT/ALL/ON/MULTIPLE/ZERO

When borrowing across a 0 and the borrow is caused by 0, the student changes the right 0 to 9 instead of 10. (600 − 142 = 457)

DECREMENT/BY/TWO/OVER/TWO

When borrowing from a column of the form N − 2, the student decrements the N by 2 instead of 1. (83 − 29 = 44)

DECREMENT/LEFTMOST/ZERO/ONLY

When borrowing across two or more 0's the student changes the leftmost of the row of 0's to 9 but changes the other 0's to 10's. He will give answers like: (1003 − 958 = 1055)

DECREMENT/MULTIPLE/ZEROS/BY/NUMBER/TO/RIGHT

When borrowing across 0's the student changes the rightmost 0 to a 9, changes the next 0 to 8, etc. (8002 − 1714 = 6188)

DECREMENT/ON/FIRST/BORROW

The first column that requires a borrow is decremented before the column subtract is done. (832 − 265 = 566)

DECREMENT/ONE/TO/ELEVEN

Instead of decrementing a 1, the student changes the 1 to an 11. (314 − 6 = 2118)

DIFF/0-N=0

When the student encounters a column of the form 0 − N he doesn't borrow; instead he writes 0 as the column answer. (40 − 21 = 20)

DIFF/0-N=N

When the student encounters a column of the form 0 − N, he doesn't borrow. Instead he writes N as the answer. (80 − 27 = 67)

DIFF/0-N=N/WHEN/BORROW/FROM/ZERO

When borrowing across a 0 and the borrow is caused by a 0, the student doesn't borrow. Instead he writes the bottom number as the column answer. He will borrow correctly in the next column or in other circumstances.

```
  100      400
 − 32     −248
 ----     ----
   72      168
```

DIFF/1-N=1

When a column has the form 1 − N the student writes 1 as the column answer. (51 − 27 = 31)

DIFF/N-0=0

The student thinks that N − 0 is 0. (57 − 20 = 30)

DIFF/N-N=N

Whenever there is a column that has the same number on the top and the bottom, the student writes that number as the answer. (83 − 13 = 73)

DOESN'T/BORROW

The student stops doing the exercise when a borrow is required. (833 − 262 = 1)

DON'T/DECREMENT/SECOND/ZERO

When borrowing across a 0 and the borrow is caused by a 0, the student changes the 0 he is borrowing across into a 10 instead of a 9. (700 − 258 = 452)

DON'T/DECREMENT/ZERO

When borrowing across a 0, the student changes the 0 to 10 instead of 9. (506 − 318 = 198)

DON'T/DECREMENT/ZERO/CARRYING/ANSWER/OVERFLOW

When borrowing across a 0, the student changes the 0 to 10 instead of 9. However, if the newly created to is over a zero, the student carries instead of writing a ten as the answer for that column. (506 − 308 = 208)

DON'T/DECREMENT/ZERO/OVER/BLANK

When borrowing across a 0 that is over a blank, the student skips over the 0 and decrements the next digit to the left. (305 − 9 = 106)

DON'T/DECREMENT/ZERO/OVER/ZERO

When borrowing across a 0 that is over another 0, the student skips over the 0 and decrements the next digit to the left. (305 − 107 = 208)

DON'T/DECREMENT/ZERO/UNTIL/BOTTOM/BLANK

When borrowing across a 0, the student changes the 0 to a 10 instead of a 9 unless the 0 is over a blank, in which case he does the correct thing.

```
  506      304
 −318     −  9
 ----     ----
  198      295
 Wrong   Correct
```

DOUBLE/DECREMENT/ONE

When borrowing from a 1, the student treats the 1 as a 0 (changes the 1 to 9 and continues borrowing to the left. (813 − 515 = 288)

FORGET/BORROW/OVER/BLANKS

The student doesn't decrement a number that is over a blank. (347 − 9 = 348)

IGNORE/LEFTMOST/ONE/OVER/BLANK

When the left column of the exercise has a 1 that is over a blank, the student ignores that column. (143 − 22 = 21)

IGNORE/ZERO/OVER/BLANK

Whenever there is column that has a 0 over a blank, the student ignores that column. (907 − 5 = 92)

INCREMENT/OVER/LARGER

When borrowing from a column in which the top is smaller than the bottom, the student increments instead of decrementing. (833 − 277 = 576)

INCREMENT/ZERO/OVER/BLANK

When borrowing across a 0 over a blank, the student increments the 0 instead of decrementing. (402 − 6 = 416)

N-9 =N-1/AFTER/BORROW

If a column is of the form N − 9 and has been borrowed from, when the student does that column he subtracts 1 instead of subtracting 9. (834 − 796 = 127)

N-N=1/AFTER/BORROW

If a column had the form N − N and was borrowed from, the student writes 1 as the answer to that column. (944 − 348 = 616)

N-N=9/PLUS/DECREMENT

When a column has the same number on the top and the bottom the student writes 9 as the answer and decrements the next column to the left even though borrowing is not necessary. (94 − 34 = 59)

ONCE/BORROW/ALWAYS/BORROW

Once a student has borrowed he continues to borrow in every remaining column in the exercise. (488 − 229 = 1159)

QUIT/WHEN/BOTTOM/BLANK

When the bottom number has fewer digits than the top number, the student quits as soon as the bottom number runs out. (439 − 4 = 5)

SIMPLE/PROBLEM/STUTTER/SUBTRACT

When the bottom number is a single digit and the top number has two or more digits, the student repeatedly subtracts the single bottom digit from each digit in the top number. (348 − 2 = 126)

SMALLER/FROM/LARGER

The student doesn't borrow; in each column he subtracts the smaller digit from the larger one. (81 − 38 = 57)

SMALLER/FROM/LARGER/INSTEAD/OF/BORROW/FROM/ZERO

The student does not borrow across 0. Instead he will subtract the smaller from the larger digit.

```
  306      306
−   8     −148
 ----     ----
  302      162
```

SMALLER/FROM/LARGER/WHEN/BORROWED/FROM

When there are two borrows in a row the student does the first one correctly but for the second one he does not borrow; instead he subtracts the smaller from the larger digit regardless of order. (824 − 157 = 747)

SMALLER/FROM/LARGER/WITH/BORROW

When borrowing the student decrements correctly, then subtracts the smaller digit from the larger as if he had not borrowed at all. (73 − 24 = 411)

STOPS/BORROW/AT/MULTIPLE/ZERO

Instead of borrowing across several 0's, the student adds 10 to the column he's doing but doesn't change any column to the left. (4004 − 9 = 4005)

STOPS/BORROW/AT/ZERO

Instead of borrowing across a 0, the student adds 10 to the column he's doing but doesn't decrement from a column to the left. (404 − 187 = 227)

STUTTER/SUBTRACT

When there are blanks in the bottom number, the student subtracts the leftmost digit of the bottom number in every column that has a blank. (4369 − 22 = 2147)

SUB/BOTTOM/FROM/TOP

The student always subtracts the top digit from the bottom digit. If the bottom digit is smaller, he decrements the top digit and adds 10 to the bottom before subtracting. If the bottom digit is zero, however, he writes the top digit in the answer. If the top digit is 1 greater than the bottom he writes 9. He will give answers like this. (4723 − 3065 = 9742)

SUB/COPY/LEAST/BOTTOM/MOST/TOP

The student does not subtract. Instead he copies digits from the exercise to fill in the answer space. He copies the leftmost digit from the top number and the other digits from the bottom number. He will give answers like this: (648 − 231 = 631)

SUB/ONE/OVER/BLANKS

When there are blanks in the bottom number, the student subtracts 1 from the top digit. (548 − 2 = 436)

TREAT/TOP/ZERO/AS/NINE

When a borrow is caused by a 0, the student doesn't borrow. Instead he treats the 0 as if it were a 9. (30 − 4 = 39)

TREAT/TOP/ZERO/AS/TEN

When a borrow is caused by a 0, the student adds 10 to it correctly but doesn't change any column to the left. (40 − 27 = 23)

TREAT/ZERO/AS/NOTHING

The student ignores 0's. (407 − 5 = 42)

ZERO/AFTER/BORROW

When a column requires a borrow, the student decrements correctly but writes 0 as the answer. (65 − 48 = 10)

ZERO/INSTEAD/OF/BORROW/FROM/ZERO

The student won't borrow if he has to borrow across 0. Instead he will write 0 as the answer to the column requiring the borrow.

```
 702      702
−  8     −348
 700      630
```

ZERO/INSTEAD/OF/BORROW

The student doesn't borrow; he writes 0 as the answer instead. (42 − 16 = 30)