13

The Analogical Origins of Errors in Problem Solving

John R. Anderson
Carnegie-Mellon University

Problem solving and learning have long been among Herb Simon's intellectual interests. The major argument of this chapter is that people learn problem solving skills in domains like mathematics and computer programming by analogy. Specifically, when solving a current problem they analogize to the solutions of previous problems (done by them or others). The idea of learning by doing has long been an idea of Simon's (e.g., Anzai & Simon, 1979) and recently Herb has been looking in detail at learning from examples (Zhu & Simon, 1987). This chapter looks at systematic pattern errors that occur in students' interactions with intelligent tutors and argues that these errors arise through the analogy process. The implication is that not only errors but successful learning occurs by analogy. In fact, most analogies appear to be successful as witnessed by the fact that such systematic errors are relatively rare.

We have been working on intelligent tutors to teach students problem solving skills in the domains of generating proofs in geometry (Anderson, Boyle, & Yost, 1985), writing LISP programs (Reiser, Anderson, & Farrell, 1985), and solving algebraic manipulation problems (Lewis, Milson, & Anderson, 1987). Student errors are the most important events in terms of guiding the tutorial dialogues. They are very good signals that the student is in need of help. Other cues do not seem so reliable. Latency of response and requests for help are the two other cues that we have considered in addition to errors. Long latencies may only mean that the student has been distracted and students are notoriously unwilling to ask for help. On the other hand an error comes close to being a necessary and sufficient condition for tutorial intervention. They are the primary signal used by almost all tutorial systems that have been built (e.g., consider Sleeman & Brown, 1982).

Errors can have different etiologies. One class of error is the *slip*, which is characterized by the fact that the subject does not reliably make that error and can self-correct when the error is pointed out. In our work on tutoring, slips appear to decrease in frequency with practice and increase with working memory load. We (Anderson & Jeffries, 1985) argued that slips can be traced to losses from working memory of critical information for solving a problem. Thus, when memory load goes up, slips increase. The decrease in slips with practice reflects a growth in effective working-memory capacity with expertise (Chase & Ericsson, 1982).

A second category of errors that has been discussed (e.g., McCloskey, 1983) is *importations of prior misconceptions into a new domain.* Physics is one domain that has many such errors. However, they have not been a significant category of error in our work given that we are working in domains for which students do not have an abundance of prior conceptions.

A third category of errors, which is the focus of this chapter, is *within-domain misconceptions.* These are misconceptions that arise, not because of prior beliefs that the student has, but as a consequence of the learning that takes place in the domain. These are perhaps the most interesting errors because they reflect on the nature of the learning process itself. Also in contrast to the first category of errors they challenge the tutor. It is not enough to simply point out the error because the student cannot self-correct easily. The tutor must explain to the student something about what the error is and what is required for a correct answer.

A very elegant set of analyses have been done by Brown and VanLehn (1980) and VanLehn (1983) on within-domain misconceptions in multicolumn subtraction. Brown and VanLehn's basic argument was that these errors arose when students tried to bridge points in their problem solving where they had inadequate knowledge. These points are called *impasses* and the bridging process produces *repairs.* If students believe their repairs, a permanent misconception appears and if they do not, an inconsistent pattern of "bug migration" appears. The analysis to be offered here has a lot in common with their analysis. Its major point of departure is that it conceives of the repair process as rather simpler than what Brown and VanLehn proposed. This chapter argues that these errors are caused by making analogies to misleading examples of problem solution.

This chapter begins with a sketch of the basic learning processes that we think are at work in the subjects we have tutored. Basically, this involves students doing a causal analysis of the examples and trying to extend that analysis analogically. The most straightforward way errors can occur in this analogy process is for students to choose an inappropriate example to map to the current solution. This is certainly not the only way misunderstandings can arise and it is of interest to get a sense of what fraction of student errors may be explained in this fashion. Therefore, I present a list of some of the dominant misconceptions that students have displayed interacting with our tutors and consider how many can be interpreted in this fashion. Then, I speculate on how many of subtraction errors, which were the focus of Brown and VanLehn's analysis, can be so explained. Finally, I compare this analysis of the source of errors with the analysis offered by VanLehn (1983, 1985).

THE PUPS THEORY OF LEARNING

We have developed a simulation program called PUPS (Penultimate Production System—Anderson & Thompson, in press), which embodies our current theory of learning. It is somewhere between an elaboration and a successor of the ACT* theory that I developed a few years ago (Anderson, 1983). It differs from that theory mainly in the emphasis it gives to analogy in the knowledge acquisition process. Data from sources such as Pirolli (1985) convinced us that analogy is very important to the learning process. Such research finds constant reference by subjects to examples when they are faced with learning a new concept. This chapter provides another converging set of data.

In the PUPS theory, learning progresses from concrete examples to abstract principles through four stages:

1. Examples are encoded into declarative structures that record their perceptually available form and attributes. Thus, if we see someone type (+ 4 2) into a CommonLISP system and the system responds 6, PUPS would encode this with something like the following structures:[1]

   event1: isa typing
   form (type person1 message computer)

   message: isa list
   form (list + four two)

   two: isa integer
   form (text 2)

   event2: isa response
   form (print computer six screen)

   six: isa integer
   form (text 6)

2. The example is "understood." This involves placing the components of the structure into a causal framework. Thus, the student might infer that the first event caused the second. This information gets encoded in function slots of the examples.

[1]The general format of these PUPS structures is structure name plus a list of attributes. In the first structure "event1" is the structure name and "isa typing," "form (type person1 message computer)," and "context CommonLISP" are the attributes. Each attribute consists of a relation like "isa" plus a value like "typing" or "(type person1 message computer)."

3. When an appropriate problem arises, an attempt is made to analogically extrapolate the understanding of the example to produce a solution to the current problem.
4. Successful extrapolations are encoded as production rules.

As for the first step of processing, I am simply assuming that our perceptual system (perhaps including the linguistic system) can deliver an encoding of the forms and attributes of things in our environment. Steps 2 and 3 are far from trivial and occupy most of our attention. Step 4 receives some analysis.

Step 2: Causal Induction

The first step in using an example is to understand its causal structure. For instance, what is the relationship between event1 and event2 in the previous example? The majority of our adult causal ascriptions flow from theories we have already acquired, but people are quite capable of making these ascriptions in the absence of a domain theory. Thus, most people would think event1 caused event2 in the absence of any knowledge about LISP or computers. It is from these pretheoretical inferences that domain theories eventually arise. There are at least three well-documented factors inducing people to perceive one thing as causing another in the absence of an existing theory (Lewis, 1986; Shultz, 1982; Siegler, 1976). Each of these can be fairly easily justified as a rational basis for making causal ascriptions:

1. Contiguity. People tend to perceive something as the cause the closer in time and space it is to the effect with the strong discontinuous provision that effects cannot precede their causes.
2. Similarity. People tend to perceive something as a cause the more similar it is to the effect. It is difficult to specify an all-encompassing metric for similarity, but for our purposes the important feature is that two things are more similar if they overlap in a number of components. For instance, suppose we observe two events involving an unknown computer system—the user points to an icon of an apple and he points to an icon of a dog. After both of these events a third event happens—the icon of the apple disappears. We are more likely to think the cause of the third event is the pointing to the apple icon than the dog icon (Lewis, 1986). This is because both cause and effect involve the apple icon.
3. Regularity. If a cause has been accompanied fairly regularly by an effect and the effect has seldom occurred in the absence of the cause, we are likely to perceive a causal relationship. Note perception of causality does not demand a perfect predictive relationship. There can always be extenuating circumstances.

It is an open question just how these three factors should be computed and combined to produce an attribution of a causal relationship. One can imagine doing psychological research by creating somewhat artificial situations to test for refined predictions of one scheme versus another. However, typically causal attributions are highly overdetermined. For instance, suppose we have no knowledge of computers and that we see (+ 4 2) typed into the computer and see 6 as a response. We would decide that the typing caused the 6 on the basis of contiguity, similarity (6 is the sum of 4 and 2), or by statistical trials noting the regularity of the relationship.

Causal information is stored in special slots of PUPS knowledge structures. Thus, if we were to take our earlier example and embellish it with causal information it would look like:

```
event1:   isa typing
          form (type person1 message computer)
          cause (event2)
          context CommonLISP
message:  isa list
          form (list + four two)
event2:   isa response
          form (print computer six screen)
four:     is a integer
          form (text 4)
fact:     isa addition-fact
          form (sequence four plus two is six)
two:      isa integer
          form (text 2)
six:      isa integer
          form (text 6)
```

I have also included the addition fact that 4 + 2 = 6, which is critical to being able to extrapolate this event. The form slot is used to record the physical form of the object or event, and cause slots record the causal position of these objects.

In many situations it is useful to compose causal relationships into higher order relationships for compact representation. This composed information is stored under function slots. Thus, rather than having the event1 cause event2 and event2 be the printing of the message on the screen, we might represent event1 as:

```
event1:   isa typing
          form (type person1 message computer)
          function (display computer six)
          context CommonLISP
```

where (display X Y) means (cause (print X Y screen)). The original development of analogy by Anderson and Thompson (in press) was with respect to function slots and not cause slots. In the remaining discussion we continue this practice.

Knowledge Extrapolation

Knowledge extrapolation involves trying to extend a causal analysis to a new situation. Suppose, for instance, one wanted to predict what would happen when (+ 5 7) was typed into CommonLISP. Analogical extrapolation in PUPS (Anderson & Thompson, in press) allows one to map the past example onto the current example provided the categories (isa slots) of the objects are the same. In this case, 4 from the past example would map onto 5, 2 onto 7, and 6 onto 12. Thus, PUPS could predict the computer would display 12. In effect, PUPS has extracted the following rule from the example:

IF	= structure:	isa typing form (type =person =message =computer)
	= message:	isa list form (list + =num1 =num2)
	= fact:	isa addition-fact form (sequence =num1 plus =num2 is =num3)
THEN	= structure:	function (display =computer =num3)

The rule above is a production rule that predicts the function of = structure given its form. PUPS can also create problem-solving productions in which the form necessary to achieve a function is specified:

IF	goal:	isa typing function (display =computer =num3)
	=fact:	isa addition-fact form (sequence =num1 plus =num2 is =num3)
THEN	goal:	form (type =person =message =computer)
	=message:	isa list form (list + =num1 =num2)

These analogical extrapolations depend on two basic assumptions:

1. One can map one member of a category to another. Thus, we are able to replace 4, 2, and 6 by variables (implicitly restricted to integers). Anderson and Thompson (in press) called this the *no function in identity principle,* because it asserts that elements from the same category appearing in analogous positions in form slots will appear in analogous positions in function slots. The assumption is that elements of the same category and in the same position in the form will have the same functional relationships.
2. One is able to trace paths through the function and form slots of structures to find paths of connections from the condition side to the action side of a production. Thus, in the aforementioned rule we find a path from =num3 through the addition fact that relates it to =num1 and =num2 in the action side. We call this the *principle of elaboration.*

For more discussion of both principles, see Anderson & Thompson (in press).

Although we often talk about knowledge extrapolation in terms of the rules extracted from an example, it is only evoked when there is an example and a target problem to map it to. The rule is just a specification of the mapping from the example to the target. Thus, availability of an appropriate example is key. Much of the research of Ross (1984) on analogy turns on manipulating variables that make the example more or less available in the target context. The hypothesis that recent examples are highly available for analogy is the key to our analysis of errors.

Knowledge Compilation

The processes of causal inference and knowledge extrapolation produce general production rules that can be used for prediction and problem solving. Those rules that prove successful get permanently recorded as production rules. Anderson (in press) observed dramatic 2-1 reductions in time and speed from the first occasion that a rule is used to later occasions. Pirolli (1985) noted subjects made analogies to examples only the first time they needed a piece of knowledge. Later trials only produce very gradual improvements. I have assumed that this reflects the compilation of a production rule on the first trial that can then be used more efficiently in later trials. This chapter is not very concerned with these compiled rules. Our concern is with the analogical processes that first produce the target rules.

SOURCES OF ERRORS IN PUPS

Given this general picture of learning where can systematic errors arise? There are in fact multiple ways that PUPS can make errors. However, this chapter is devoted to exploring what seems to be the most probable way that it would make systematic errors—which is to map an example inappropriately to the problem. Typically this is because the example that is chosen is inappropriate for solving the target problem, but it is also possible that an appropriate example is selected but mapped inappropriately. Either of these cases will be put under the heading of "misleading example." As evidence of how obvious

this is as an error mechanism I can report that the most frequent criticism I get from people to whom I describe PUPS goes like this: "Yes, it works all right if you choose just the right example and encode it just right but how can you assume people always do this?"

The major goal of the remainder of the chapter is to look at systematic errors in our three tutors and see how many can be interpreted as mistaken examples. I have culled from our protocols cases where at least 30% of the students make the same error at the same point (the 30% is calculated as a percentage of all responses, not just errors). this consistency of the error suggests it is not a slip. Unfortunately for the current purposes, errors tend not to be repeated across problems in the tutors. This is because subjects are immediately corrected. Thus, we do not see in our tutor interactions the kind of consistent bugs that Burton and Brown (1982) found in subtraction. The reason is that they did not intervene tutorially. If we had not intervened, the error we see might become a permanent part of the student's repertoire. However, because we do intervene, we cannot use consistency of error pattern as a basis for diagnosing systematic errors. This is why we resort to a criterion like 30% same errors because it is highly unlikely that this consistency would occur through slips.[2] However, this criterion means we will not see the bizarre and rare errors Burton and Brown were able to document.

It would probably be possible to conjure up some example under some encoding that could serve as an analogical basis for any error we observed. Thus, the mere fact that we can explain an error by analogy to an example is not very compelling evidence that it actually occurred because of analogy. One has to judge the reasonableness of the encoding of the example from which the analogy flowed. I think the encodings we offer are far from strained and, indeed, are quite compelling—perhaps even obvious. Our analysis is more compelling if it turns out that examples that serve as the source of the analogy occur in the vicinity of the error. According to the PUPS theory, examples just processed should still be active and hence highly available for analogy. Our tutoring paradigm allows us to determine what examples a student has studied in the vicinity of an error. This is where our data base is at an advantage over the one used by Burton and Brown. They have no record of the learning context in which the error first appeared. With our tutors we have a very good record.

What we do is show that the majority of the errors we see can be interpreted as bad analogies to close-by examples. What we do not know yet is whether PUPS predicts analogy errors that in fact are not observed—what Brown and VanLehn (1980) would call *star bugs.*

The 30% criterion was chosen because it gave us about 10 consistent errors for each of our three tutors (LISP, geometry, and algebra). In each of

[2]The assumption is that working memory failures are slips at random and produce a wide range of different responses and would not concentrate at one place in the protocol.

the sample data sets we are looking at about 500 interactions. Thus, these consistent errors only occur on about 2% of the possible occasions. This small number testifies to the rather low error rates on the tutors (from 10% to 25% of all interactions are errors, depending on tutors) and the fact that most errors are relatively idiosyncratic to a subject. Thus, these events that we are examining are statistically exceptional in the tutor interactions. The research strategy here is to use exceptional events to tell us what is happening in the learning process. As we said earlier, our assumption is that analogy usually produces successful learning and we are examining the rare occasions where it does not work.

ERRORS WITH THE LISP TUTOR

In the case of the LISP tutor I present a list of all errors meeting the 30% threshold that occurs in lessons 2 and 3 of the tutor. Lessons 2 and 3 are analyzed rather than lesson 1 because lesson 1 has some peculiar properties due to start-up with the LISP tutor.

First

Lesson 2 concerns how to write LISP functions. The very first problem is specified to the students as: "Define a function called *first.* Given any list, it returns the first element of that list. For example, (first '(a b c)) returns a." Given that this function is totally redundant with the LISP function CAR, this is really just an exercise in function definition. The following is the correct code:

```
(defun first (lis)
  (car lis))
```

A full 30% of the subjects make the following error:

```
(defun first (lis)
  (car (lis)))
```

Anderson, Farrell, and Sauers (1984) discussed at length a simulation of a protocol of a subject making this error. In that simulation it came from an inappropriate analogy to the following function definition:

```
(defun f-to-c (temp)
  (quotient (difference temp 32) 1.8))
```

The first argument to the function quotient is in parentheses and the subject places the argument in parentheses in writing f-to-c. This is an error in the representation of the example. The subject is not representing the fact that the parentheses in the *f-to-c* example are to hold an embedded function call

not just an argument. An important observation for current purposes is that the f-to-c example is used to illustrate a function definition and so the example is occurring immediately prior to coding first.

In informal interviews with students, we have also noticed some who make this error out of analogy to the parameter list in the first line of the function definition. In fact, many teachers have complained that students are confused because the parameters in LISP definitions are placed in parenthesis and the arguments to functions are not. This is another case of an error by analogy to an inappropriate example. The student either does not encode or ignores that a list is used for specifying parameters and not for holding arguments to functions. In this case the subject is using one piece of the code as an analog for another piece of the code.

Replace

The function that follows first is called *second* and it is supposed to return the second element of the list. This does not display a comparably consistent error. The next function does however. It is specified to the student as: "Write a function *replace* that replaces the first element of a list with a new element. The function takes two parameters—the new element and the list. For example, (replace 'rings '(ties hats pants)) returns (rings hats pants)." The correct code is:

```
(defun replace (item lis)
    (cons item (cdr lis)))
```

A full 55% of the students start out trying to define the function:

```
(defun replace (variable). . . .
```

Variable stands for whatever variable name they choose. The relevant fact is that the two previous function definitions both involved a single list parameter. This is further evidence that some subjects are not understanding what the parameter list is about and inappropriately map it here. Our guess is that subjects incorrectly conjecture that each parameter must separately be placed in parentheses. We would predict that subjects would have encoded a second parameter in a second list. That is, their code would have taken the form:

```
(defun replace (item) (lis)
```

with a list for each parameter. Unfortunately, the tutor stops them after their first error and so we do not see how they would have continued the code. The important observation about this analysis of the error and others like it is that it attributes the error to incorrectly understanding an example and, consequently, inappropriately mapping that example.

Sqr

The next function to display a high consistent error pattern is specified to the student as "Define a function called sqr that returns a list of the perimeter and the area of a square, given the length of one side. For example, (sqr 2) returns (8 4)" The following is the correct code:

```
(defun sqr (side)
    (list (times side 4) (times side side)))
```

A full 35% of the subjects started their code as follows:

```
(defun sqr (side)
    (times
```

List is missing. This error has no obvious analog and so stands as the first example of an error that cannot be explained by analogy to a close-by example. It is an instance of what we have called a *part error.* That is, subjects never had to code a list answer before and we assume this 35% could not figure out how to achieve that. What they did was to proceed to code that part of the answer that they did know how to code—namely the circumference of a square. Subjects have a tendency to do part of a problem if they cannot do it all.

Ends

The next function that produces a high error rate is specified to the students as "Define a function called ends, that has one argument and returns a list containing the first and last items in that argument. For example, (ends '(a b c d)) returns (a d)." The following is one form of the correct code:

```
(defun ends (lis)
    (list (car lis) (car (last lis))))
```

A full 50% of the subjects tried the following code:

```
(defun ends (lis)
    (list (car lis) (last lis)))
```

This argument could be classified as a part error—subjects did not know how to code the first of a list and so they coded just the list—(last lis) returns a *list* of the last element of the list. However, this does not seem very plausible. Subjects have coded car successfully many times (including in this function) and this is the first time they used last. More likely it represents a misunderstanding of what last does. Our guess is that subjects do not know what last does—that they believe it returns the last element rather than a list of the last element. We have observed many students firmly assert that last returns the last element despite the fact that they have read instruction and seen examples to the contrary. This misunderstanding of last may come from interpreting it as analogous to car, which does return the first element rather

than the first list. This error is an interesting case. Under this analysis it has its origin in analogy, but it is not an analogy of the problem to a nearby example. Rather it is an analogy of the instruction about last to the knowledge about car. Thus, we cannot count this as a supporting case for our hypotheses although the error may stem from analogy.

Snoc

The next function to produce a high error rate is *snoc,* which is specified to students as: "Write a function called *snoc* that is the opposite of *cons.* Instead of inserting an item into the front of the list, it inserts the item at the end. For example, (snoc 'd '(a b c)) returns (a b c d). Write this function without using append." The following is the correct code:

```
(defun snoc (item lis)
    (reverse (cons item (reverse lis))))
```

A full 70% of the students start their code for this function:

```
(defun snoc (item lis)
    (cons . . . .
```

It is unlikely that this is a part error stemming from an inability to use reverse because they have already used it a number of times successfully. The interesting feature of this example is that the instruction itself implicitly references an example of cons. The error could derive from analogy to this implicit example. We assume students represent the body of the function they are to write as follows:

```
problem: isa lispcode
        function (insert item lis end)
```

and their representation of the implicit *cons* example is:

```
example: isa lispcode
        function (insert item lis front)
        form (cons item lis)
```

The implicit example has the insert at the front where the problem has the insert at the end. Perhaps they intended to reverse the arguments to cons or perhaps they choose to ignore the mismatch between cons and their current problem.

Samesign

The third lesson is on predicates, logical operators, and conditionals. The first function to produce a high consistent error was described to the students as: "Define samesign. It takes two numbers as arguments, and returns *t* if both arguments have the same sign. That is, if both arguments are zero, both positive, or both negative, the function should return *t.* For example, (samesign 0 0) returns *t,* (samesign -2 -5) returns *t,* and (samesign -2 3) returns nil." At this point students should have known about the logical functions *and* and *or* but not *cond.* Thus the code we expected was:


```
(defun samesign (num 1 num2)
    (or (and (greaterp num1 0)(greaterp num2 0))
        (and (greaterp 0 num1)(greaterp 0 num2))
        (and (zerop num1)(zerop num2))))
```

Or some equivalent variant. Eighty percent of the students started their coded displaying the following error:

```
(defun samesign (num1 num2)
    (greaterp . . .
```

That is, they ignored the *or* and the *and.* Because this was the first exercise in *or* and *and* we assume they just did not know how to use them. This is the third case where there is no obvious analog to a nearby example for making the error. This seems like another part error where the subject is again trying to write that part of the code they know how to—namely coding predicates.

Carlis

The first function to use conditional structure was *carlis.* It was specified to the subject "Define *carlis.* It takes one argument. If the argument is a nonempty list, then *carlis* returns the first element of that list. But if the argument is the empty list, then *carlis* returns the empty list. If the argument is an atom, *carlis* returns just that atom. Hint: Be careful how you order your tests. Remember that nil is both an atom and a list. For example, (carlis '(cat rabbit)) returns cat, (carlis 'george) returns george, and (carlis nil) returns nil." The correct code for carlis is:

```
(defun carlis (object)
    (cond ((null object) nil)
        ((atom object) object)
        (t (car object))))
```

or some variant. This problem produced the following two high frequency errors:

```
(defun carlis (object)
    (cond (null object) nil)
        ((atom object) object)
        ((listp object) (car object))))
```

The first error, made by 30% of the subjects is to type just a single left parenthesis before null.[3] This is the first time they have had to code two left parentheses in a row and we assume the many examples of a single left parenthesis dominate. The second error, made by 80% of the students, is to use a predicate in the test for the final clause—(listp object). Although semantically it is not an error, the tutor treats it as a stylistic error and the example function students studied involved a *t* for the last clause. However, the two previous lines in this function provide more recent examples of coding tests and we assume these are the sources of the mistake. Basically, their status as appropriate for only nonfinal tests is not being represented or ignored by students.

This error and the next we discuss are both stylistic. They do not represent code that will not work, but rather solutions that are less than optimal by reasonably well-accepted criteria. Numerous people have questioned mixing these in with true errors. Therefore, it is important to be explicit why such mixing is appropriate for our current purposes. The goal here is not to pass evaluative judgment on the student's behavior or to grade it. The goal is to understand the origin of that behavior by focusing on cases where it deviates from the tutor's prescriptions. For these purposes a stylistic deviation that arises from analogy is every bit as interesting as a true error. Both indicate that analogy is in control of the learning process.

Numline

The next function to produce a high error rate is *numline*. It is specified to the student as "Define a function called *numline*. It takes one argument that is a number and returns a two element list. The first element of the list is *t* if the number is 0 and nil otherwise. The second element of the list is *t* if the number is negative and nil otherwise. For example, (numline -5) returns (nil *t*)." The target code for this function is:

```
(defun numline (item)
    (list (zerop item) (< item 0)))
```

This function just occurs after a series of functions, all coded with cond. Fifty percent of the students produce the same response, which is to try to code the function as follows:

```
(defun numline (item)
    (cond . . .
```

This might lead to a function that works, but it is clearly nonoptimal (which is what the tutor tells them). It seems that it is again analogy to the recent previous functions that produce this mistake.

[3]The single parenthesis error is immediately corrected by the LISP tutor, which is why they know not to make the error before *atom*.

LISP Errors: A Concluding Remark

We have now reviewed the nine dominant high frequency errors in lessons two and three with the LISP tutor. Two of these are explainable as part errors and another six as inappropriate selection of a recent example. The ninth error may arise from a misanalogy in the understanding of the instruction about *last*. I think the relatively simple genesis of these errors is interesting given what a complex domain LISP programming is supposed to be.

ERRORS WITH THE GEOMETRY TUTOR

In the case of the geometry tutor I present an analysis of all above 30% errors that occur in the first chapter. This is a chapter mainly devoted to relating algebraic operations on geometry measurements to some basic geometry concepts like congruence. The high school students who go through this material often find it frustratingly subtle. If we were designing a geometry course we would not begin it with such material, but our tutor was compelled to follow the textbook sequence.

Before going through the individual errors I should say a little about the geometry tutor, although for details consult Anderson, Boyle, & Yost (1985) and Anderson, Boyle, Corbett, & Lewis (in press), Fig. 13.1 shows a completed geometry proof as it appears on the tutor screen. It takes the form of a graph structure connecting a set of premises on the bottom to the conclusion to be proven at the top. Students interact with the tutor by picking a set of premises, stating a rule to apply to the premises, and then stating the conclusion that follows.

In my discussion of the errors, for brevity's sake I present descriptions abstracted away from many of the details of interaction. I just describe the diagram, the premises, the to-be-proven conclusion, and the nature of the error without reference to the exact screen images.

Addition Postulate

The first geometric rule introduced to the student is the addition postulate: if $a = b$ and $c = d$, then $a + c = b + d$. Figure 13.2 shows the three problems on which students practice this postulate. The first two cause no difficulties—students combine the two premises to establish the conclusion. However, 80% of the students make the same error on the third problem. They try to apply the postulate to all three premises at once. Whether this really should be counted as an error is problematical, but the tutor and the text restrict the addition postulate to two premises. What is remarkable is the number of students who make the opposite inference. I think the error quite definitely comes from analogy to this training sequence, but the in-

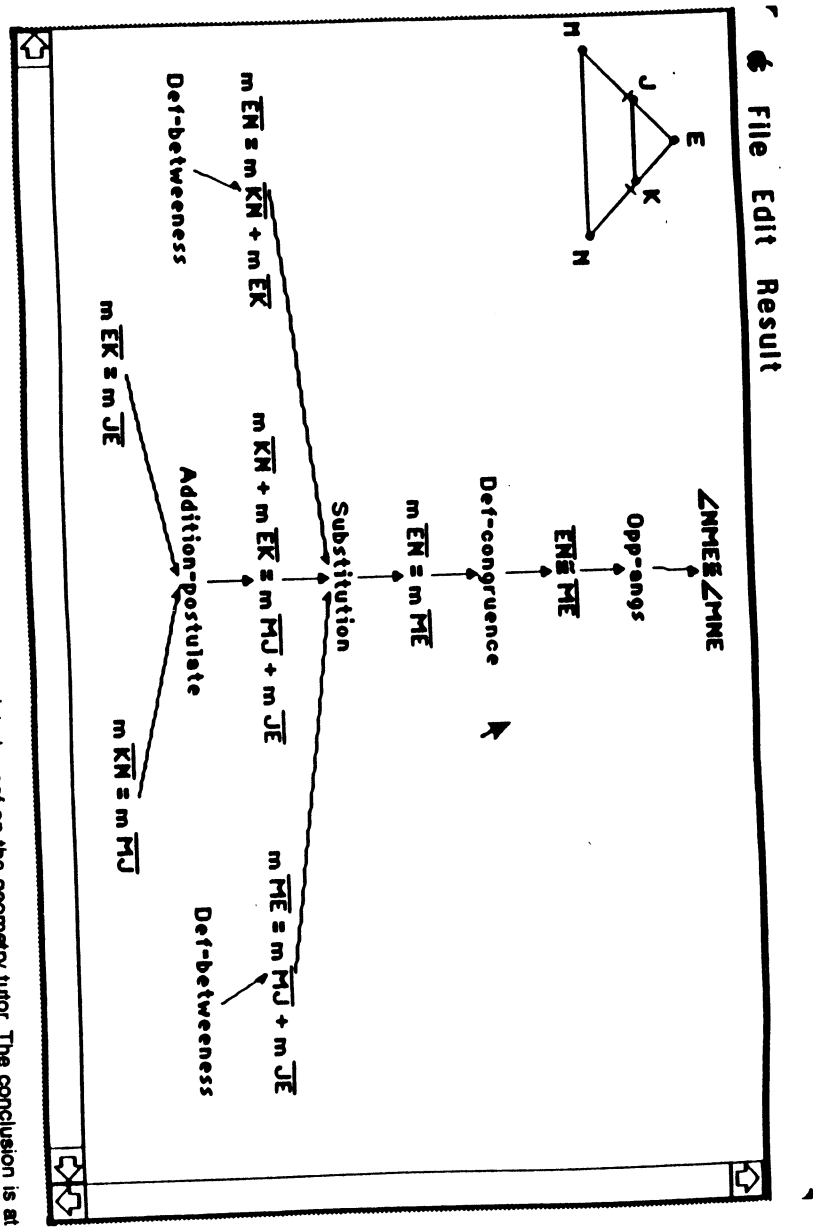


FIG. 13.1. An example of a completed proof on the geometry tutor. The conclusion is at the top of the series and the initial premises are at the bottom of the screen.

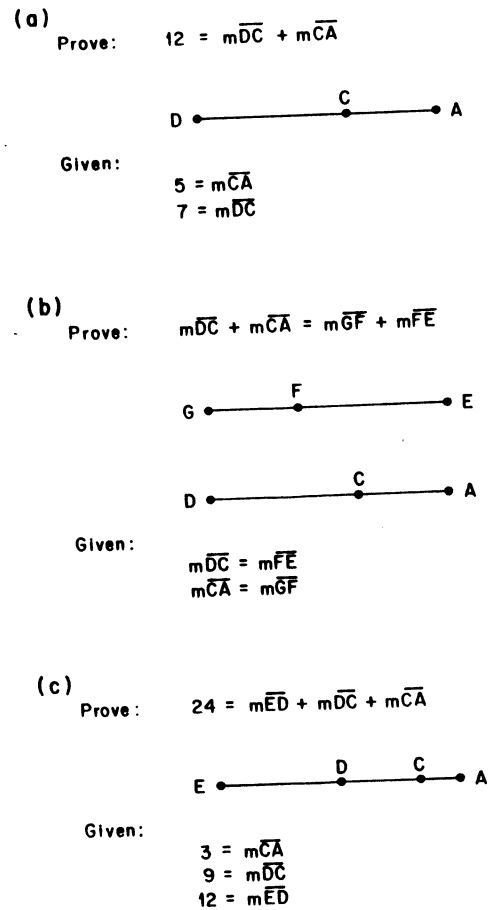


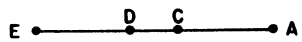
FIG. 13.2. The first three problems in the geometry tutor. (a) and (b) are problems that cause few errors, (c) produces a high-frequency error.

teresting observation is that PUPS, as it is currently implemented, does not make the analogy from two of a kind to three of a kind. This points in a direction that we have to develop the PUPS analogy system.

Reflexive Postulate

The postulate to follow addition is subtraction and it does not promote a similarly consistent pattern of errors. However, t he next postulate, the reflexive postulate, asserting a quantity is equal to itself, does. Figure 13.3 shows the first problem involving this rule. It requires subjects to first establish mDC = mDC and then use subtraction. The peculiar feature about establishing that $m\overline{DC} = m\overline{DC}$ through the reflexive postulate is that this postulate requires no premises. Eighty percent of the subjects first choose the one premise given and try to apply some rule to it. There is no obvious rule that students

Prove: $m\overline{CA} = m\overline{ED}$



Given: $m\overline{DC} + m\overline{CA} = m\overline{ED} + m\overline{DC}$

FIG. 13.3. In this problem, students try to apply a rule to mDC + mCA = mED + mDC rather than first establishing mDC = mDC by the reflexive postulate.

can apply to this premise and so their rule-posting behavior is quite variable—the majority try to post the reflexive rule as applying to this premise but others post the addition rule, others the subtraction rule, others just quit the inference, and others ask for help. Thus, they make the error of selecting the premise and then try a host of different behaviors to get out of corner they have painted themselves into. This seems a clear analogy to their pattern of responding with the tutor up until this point where they have always had to choose a premise.

Definition of Congruence

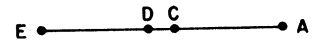
The fourth rule is the definition of congruence. It asserts that segments with equal measure are congruent. Figure 13.4 shows a problem that produces two common errors. Almost all subjects again select the premise on the screen rather than using the no-premise reflexive rule. Secondly, when subjects establish that mDC = mDC, they then combine this and the given premise by the subtraction postulate but choose as the conclusion the statement on the screen involving congruences rather than the fact that mCA = mED, which has to be converted into the congruence. The obvious analog to this problem is the problem in Fig. 13.3, which occurred three problems earlier. The students are simply mapping one to the other, ignoring the difference between congruence and equality.[4]

Substitution

The next rule introduced allows students to substitute an equal term in another equality. Figure 13.5 shows the first problem that students have to solve with this rule. It involves using the definition of congruence to go in the other direction and convert $\overline{HN} \cong \overline{SM}$ into mHN = mSM. Seventy percent of the students pick both statements without first converting, and try substituting

[4]The reader may well consider why this distinction between equality of measure and congruence is being enforced in the geometry curriculum. Without defending the distinction, let me state the rationale: Congruence means that objects will be the same after rotation, reflection, and translation. Equality means the identity of two numerical measures. The distinction becomes important in later chapters where two objects (e.g., triangles) can be equal in some measure (e.g., area) but not congruent.

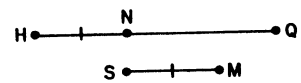
Prove: $\overline{CA} \cong \overline{ED}$



Given: $m\overline{DC} + m\overline{CA} = m\overline{ED} + m\overline{DC}$

FIG. 13.4. A problem that produces two high-frequency errors in the geometry tutor.

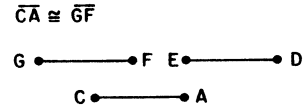
Prove: $12 = m\overline{SM} + m\overline{NQ}$



Given: $\overline{HN} \cong \overline{SM}$
$12 = m\overline{NQ} + m\overline{HN}$

FIG. 13.5. In this problem, students fail to convert $\overline{HN} \cong \overline{SM}$ into mHN = mSM before applying the substitution rule.

Prove: $\overline{CA} \cong \overline{GF}$



Given: $m\overline{CA} = m\overline{ED}$
$m\overline{ED} = m\overline{GF}$

FIG. 13.6. Subjects tend to combine the two equalities by transitivity and conclude the congruence statement.

the congruence statement into the measure statement. Again it is to be explained as students confusing equality and congruence. This time the analogy is to the example that is used to illustrate substitution in the instruction, which does not require such a conversion. This is what they have studied just prior to solving the problem.

Transitivity

The next rule is transitivity and Fig. 13.6 shows the first problem on which 40% of the students make the same error. Perhaps the reader can guess what it is: Students combine the two equalities by transitivity and conclude the congruence. Again the analogy is being made to the instructional example that is being used to illustrate the transitivity of equality.

Midpoint

The next rule is definition of midpoint. Without burdening the reader with the problems, let me just say students do well on the first two problems but once again overextend their knowledge on the third problem, which requires discriminating between congruence and equality.

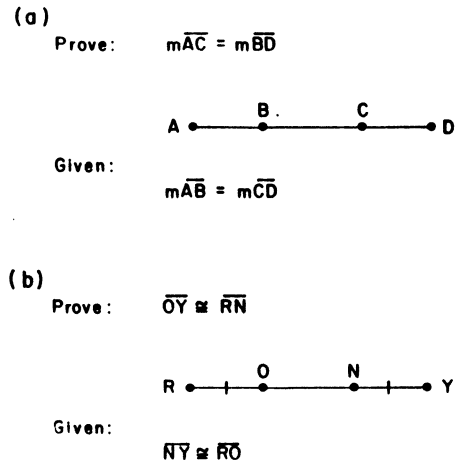


FIG. 13.7. (a) Most students conclude mAB + mBC = mAC and mBC + mCD = mBD and then try unsuccessfully to use these conclusions as premises for a rule. Students make an error on (b) by using (a) as an analogy. They fail to establish congruence as a separate step from equality of measure.

Betweenness

The next rule is the definition of betweenness, which says that three points A, B, C, are collinear with B in between if and only if mAB + mBC = mAC. The first four problems involving this rule are rather uneventful and then subjects come upon the problem in Fig. 13.7a, which high school students always find very difficult. Most students are able to analogize to their past four solutions and make the inferences that mAB + mBC = mAC and that mBC + mCD = mBD. Then the majority of students select these two premises, which is an error—no useful rule will apply to these premises. Having selected them, what they next try to do is unsystematic. Some students abort the inference; some try subtraction, some try addition, some try substitution, and so forth. This is the only high frequency error in our geometry protocols that does not seem to have an explanation as an analogy to some example. In some sense subjects feel that these two premises must be enough to establish the conclusion.

After this problem, the tutor repeats a number of similar problems. Figure 13.7b shows the third in the sequence (i.e., there is one problem between Fig. 13.7a and Fig. 13.7b). Students learn inference pattern required, but then make a systematic error in Fig. 13.7b. Again ignoring the difference between congruence and equality, they try to directly apply an algebraic rule to the congruence $\overline{NY} \cong \overline{RO}$ without first converting it to an equality.

Concluding Remarks About Geometry

We have looked at nine common errors in geometry and found that all but one of them can be explained as an analogy to a close-by example. The majority (5) of these analogy errors turn on subjects failing to block an analogy

by observing the distinction between equality and congruence. It should be stressed that this confusion is well-known in high school geometry and is not a product of the tutor. In fact, it is to the tutor's credit that it eventually gets students to respect the difference between equality and congruence. Many high school teachers have just given in and do not require students to make the distinction.

ERRORS WITH THE ALGEBRA TUTOR

The algebra tutor currently teaches a course that reviews prealgebra (fractions, signed numbers, distribution), solving linear equations in one unknown, polynomials, and solving quadratics. We have analyzed the nine errors that occurred above the 30% threshold in the prealgebra curriculum. Interactions with the algebra tutor involve selecting an operator, passing its arguments, and then producing a result. Most of our high-frequency errors involve calculating the result. Almost always it appears that the error can be explained as an analogy to the immediately preceding problem.[5] Therefore, we present our results in terms of correct answer on the preceding problem and then common error on the current problem.

Reciprocal

One of the early prealgebra skills that students are taught involves finding the reciprocal of a fraction. The following two are the key problems:

correct prior: reciprocal (-3/4) = -4/3
common error: reciprocal (1/4) = 4/1 (rather than 4)

Fifty-five percent of the students make this error although they have been told to enter their answer in simplest possible form. The analogical source of this error is apparent. Again, it is an interesting question in just what sense this is an error. However, our task is not to judge the rules of the game but to observe where students deviate from these rules.

Greatest Common Factor

The next operation to produce a high-frequency common error involves finding the greatest common factor (GCF) of two numbers

[5]It is interesting to consider why the algebra errors can always be traced to the immediately preceding example, whereas in geometry the examples were as much as three back. This is because the immediately preceding problem in geometry and LISP was often not capable of being mapped to the next problem whereas, given our problem sequence, it was almost always possible to make such a mapping in the case of algebra.

correct prior: GCF (44, 28) = 4
common error: GCF (81, 54) = 9 (rather than 27)

Thirty-five percent of the students make this error. In the prior examples when the two numbers have a common factor in the multiplication table (44 = 4 × 11; 28 = 4 × 7) the common factor is the greatest common factor. Subjects appear to be extrapolating this erroneous pattern to the current case and coming up with 9 since 81 = 9 × 9 and 54 = 9 × 6.[6]

Fraction Addition

Another prealgebra skill reviewed by the tutor is fractional arithmetic. The following example produces the high error rate:

correct prior: ADD (3/28, 5/7) = 23/28
common error: ADD (1/21, 7/6) = 51/42 (rather than 17/14)

Forty percent of the students make this error. This is the first case of an example from fraction addition where the result needed simplification. We assume that by analogy to the previous problems some subjects were omitting the simplification step when it was needed.

Variable Combination

Another lesson involves representing the product of a constant and a variable. This produces the following two problem sequence:

correct prior: Varcombine (-10, X) = -10X
common error: Varcombine (-1, Y) = -1Y (rather than -Y)

Fifty-five percent of the students make this error. This is the first case of combining a variable with 1 or -1.

From this example students learn the special case rule involving 1 and apply it correctly until they come across the following pair, which involves multiplying parenthesized expressions by constants.

correct prior: Varcombine (-31, (-1X + 17)) = -31(-1X + 17)
common error: Varcombine (-1, (4/5Z - 3) = -1(4/5Z - 3)
(rather than -(4/5Z - 3))

Forty-five percent of the students made this error. By accident the prior example had gotten into the tutor in a form where the special case rule for 1 was violated. Many students promptly copied this pattern for the next example. This is probably the case where the student's behavior least deserves the classification "error." However, for our purposes it reinforces how much of the students' behavior is analogically based.

[6]One might argue that this error is due to subjects abrogating search for a greatest factor too early and not due to the prior example. However, students had opportunity many times before to make the error (including giving 2 as the answer to the prior exmaple). The striking fact is this is the first time this error appears with any frequency.

Factor

The next operator that produces a common error involves factoring out a common product.

correct prior: factor (5 * 3X + 5 * 1) = 5(3X + 1)
common error: factor (5 * 3Z + 5) = 5(3Z + 5)
(rather than 5(3Z + 1))

Thirty-five percent of the students make this error. This sequence of two was an informally constructed sequence designed to get students to extract the pattern by analogy. However, in making the mapping from the prior example to the target problem subjects map both the 5 and the 1 into the 5 producing the error observed.

Expandexpression

Expandexpression is an operator that rewrites an expression in terms of products involving a specific term. It produces the following sequence:

correct prior: Expandexpression (5 + 20X, 5) = 5 * 1 + 5 * 4X
common error: Expandexpression (18 + 6Y, 6) = 6 * 3 + 6 * 1Y
(rather than 6 * 3 + 6 * Y)

Thirty-five percent of the students make this error. This is another case of subjects failing to take into account the special case nature of 1 in their analogies. Note that this error had laid dormant through many opportunities in prior problems but arises anew when students face a new algebraic operator.

Factor

The next operator to produce a consistent error pattern involves factoring a sum. This error pattern involves choosing a wrong suboperator rather than wrong result.

prior example: Factor (-14Z + 7)
GCF (-14,7) = 7
Expandexpression (-14Z + 7, 7)
common error: Factor (-6 - 8 X)
GCF (-6, - 8) = -2
Expandexpression (-6 - 8X, 2)
(rather than Expandexpression (-6 - 8X, -2))

The first step in factoring is to find the greatest common factor of the two integers and the second step is to use expandexpression with the expression and the greatest common factor. The problem that produces the error is the first case where the greatest common factor is negative. Fifty percent of the students choose to pass a positive integer as the argument to expandexpression by analogy to the previous problems.

Adding Terms

The following is the final problem to produce a consistent error pattern:

prior example: ADD (-3/2Z, 5/2Z) = Z
common error: ADD (2X, -2X) = X rather than 0

Forty percent of the students make the error illustrated. It seems unlikely that this is just an analogy error. This error has been analyzed by other researchers as a failure to properly parse the problem (Matz, 1982). Students analyze the 2 and the -2 cancelling each other leaving the X. Their problem is that they do not know how to combine the 0 that results and the X. This is not a problem due to analogy to the prior example.

SUMMARY

Table 13.1 presents a summary of our analysis of the high-frequency errors in the three domains. Clearly there is a preponderance of analogical errors. Given the informal nature of our classification, there is room for disputing particular analyses of particular errors. However, such disputes would not take away from the conclusion of a heavy proportion of analogical errors. On the other hand, in absolute frequency such errors are rare. This suggests that in most cases where analogy is at work it produces the right result.

EINSTELLUNG

The reader may have noted the similarity between the errors we are observing and those errors that have been called *Einstellung errors* (Luchins, 1942). This is the phenomenon that states when students have had a run of success with a particular solution pattern they are likely to try to repeat that pattern on a problem where it is no longer appropriate. In fact, in the PUPS theory, Einstellung errors are to be attributed to choosing inappropriate examples for analogy. This contrasts with the explanation that had been offered in ACT* and by others (e.g., Lewis, 1978) that saw Einstellung errors as being produced by composing together sequences of production rules. The problem with this explanation has always been that the tendency to make this error is very much

TABLE 13.1
Classification of High Frequency Errors

	Analogy-Based	*Not Analogy-Based*
LISP	6	3
Geometry	8	1
Algebra	8	1
Total	22	5

under conscious control. For instance, Luchins was able to cut these down by 50% just with the admonition "Don't be blind." The demonstration also never works in my class if I introduce it as an example of where people get tricked in problem solving. Composed production rules are not the sort of things in most theories that are subject to such conscious control. On the other hand, it is perfectly plausible that subjects interpreted Luchins' "Don't be blind" instructions as instructions not to use the obvious recent example pattern that had been working.

COMPARISONS WITH VANLEHN'S THEORIES

As a final point it is worth comparing what I am saying here with VanLehn's analysis of errors. He has produced two theories of the origins of bugs. The earlier is the repair theory that he developed with Brown (Brown & VanLehn, 1980) and the more recent is the step theory he developed for his dissertation (VanLehn, 1983). These are complimentary not alternative hypotheses. Step theory attributes bugs to an inductive process of learning from examples, whereas repair theory attributes bugs to repairs that students invent when they come to impasses in their problem solving.

VanLehn has already compared my analogy-based explanation with inductive-based explanation of step theory. He comments, "Although I have not investigated example-exercise analogy in detail, I expect it to behave indistinguishably from learning by generalizing examples" (VanLehn, 1985, p. 19). I think he is right in that there is no difference between analogy from examples generally considered and induction from examples generally considered. It is the case that PUPS and his step theory are not identical, but then I think we both admit that our theories are not sufficient to produce the full class of inductive/analogical errors. The one thing that the analogy perspective of the PUPS theory emphasizes is that one should be able to observe students making mappings between examples and problems and learning from these mappings. A dominant feature of protocols from students' learning is the presence of these analogical mappings. As we noted in the introduction, it was protocol data from subjects that first led us to our theory of analogically based instruction.

On the other hand, repair theory does contrast with the predictions of PUPS analogy. It is interesting to look at the domain most associated with VanLehn, subtraction bugs, and classify those according to whether they seem to have an analogical explanation or a repair explanation. There are bugs that Brown and VanLehn cannot explain with their repair model but that are naturally explained as analogy errors due to wrong selection of an example. There are errors that can be explained either way. There are errors for which the Brown and VanLehn explanation seems much more plausible than any analogy-to-wrong-example explanation I have been able to think of. Finally, there are errors that neither model can explain. I discuss examples of each.

Add Instead of Subtract

The error of addition instead of subtraction has an obvious analogical explanation where the student relaxes the constraint on the sign and uses an addition example as an analogy for a subtraction example. On the other hand, it is not possible to explain this bug as a repair. I think this error is very plausibly explained in analogical terms because it is known that children who can do subtraction perfectly when presented with a sheet of all subtraction problems will make add-instead-of-subtract errors when those subtraction problems are intermixed with addition problems. Such mixed addition and subtraction problems provide students with erroneous analogs in close proximity to the subtraction problems.

Carry Instead of Borrow

Equally explainable in terms of analogy is the error where students perform subtraction correctly except that they carry when they should borrow. Again it cannot be explained by repair theory.

Smaller-From-Larger

Certainly the most common subtraction bug, and one that I have anguished much over with my son, is the smaller from larger bug illustrated:

$$\begin{array}{r} 93 \\ -37 \\ \hline 64 \end{array} \text{ which has the obvious analog - } \begin{array}{r} 7 \\ -3 \\ \hline 4 \end{array}$$

In PUPS terms one can represent the problem of finding the digit in the units column as:

Goal: isa number
  function (difference 3 7)
  form ????

and the example:

example: isa number
  function (difference 7 3)
  form (text 4)

Cast this way there is actually a problem explaining the error in PUPS because PUPS respects the order of the arguments to a relation like *difference* and will not make the analogy. Thus, the feature that has to be relaxed here is the argument order. However, despite the fact that PUPS will not, it is more than plausible that children are willing to relax this order. Most kids, and, I suspect most adults (certainly the ones I have tested), when asked "what the difference between 3 and 7" will respond 4, not −4 or impossible. In fact, memory of examples where it is heard "the difference between 3 and 7 is 4" could serve for PUPS as the analog that would allow it to make the argument reversal.

The Brown and VanLehn explanation of this affirms that students reverse the order of the arguments when they hit an impasse. The difference is that they do not tie it to any specific example of subtraction. Presumably, the two points of view could be separated by careful experimental data.

Borrow-No-Decrement

This error is illustrated below:

$$\begin{array}{r} 62 \\ -44 \\ \hline 28 \end{array}$$

This is produced in repair theory by deleting the decrement rule associated with borrowing from a column. It would be produced in PUPS by not encoding the decrement in an example and hence not learning the rule. This is really a very subtle difference and, if the two accounts could be separated at all, it would require taking protocols to find out what students attend to in the examples they learn from.

Zero Instead of Borrow

This error is illustrated below:

$$\begin{array}{r} 42 \\ -16 \\ \hline 30 \end{array} \text{ could be analogy to } \begin{array}{r} 42 \\ -12 \\ \hline 30 \end{array}$$

However, I think it is implausible to suppose there is very often such a nearby analog. The standard repair explanation, that the student starts to count down from 2 and hits zero, is much more plausible. I think if the student

were going to look for an analog to deal with this dilemna, the smaller-from-larger bug would be produced. Thus, this is one example of a number where I find the repair explanation definitely more compelling.

Subtract-Top-From-Bottom

Just to document an error that neither analysis can handle well consider the following subtraction error, which has apparently been documented in at least one kid's behavior:

$$\begin{array}{r} 81 \\ -27 \\ \hline 46 \end{array}$$

In this error the student chooses to subtract the top number from the bottom. Thus, 6 is written as the difference of 7 and 1. The student borrows a mysterious 1 to convert the 2 to 12 and then subtracts 8 from it to get 4. Apparently this error has stumped all attempts at explanation.

CONCLUSIONS

Although I hate chapters that end with calls for more research, it is essential to end on such a note. This chapter has really been a plausibility argument. We need to extend the PUPS simulation to establish that it can generate the full class of analogy errors. More important, once it does, we need to expose it to the full curriculum that students see to determine if there are overgeneration problems. That is, will the PUPS analogy mechanism produce errors that we do not see in students protocols? The point of such an exercise is not so much to establish that PUPS per se is correct but to establish more precisely the sense in which these are analogy errors.

ACKNOWLEDGMENTS

The research reported on this paper was supported by ONR contract N00014-87-K-0103 and NSF grant MDR-8470337.

REFERENCES

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Anderson, J. R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information

Anderson, J. R., & Thompson, R. (in press). Use of analogy in a production system architecture. In S. Vosniadou & A. Ortony, (Eds.), *Similarity and analogy.* New York: Cambridge University Press.

Anderson, J. R. Boyle, C. F., & Yost, G. (1985). The geometry tutor. In *Proceedings of International Joint Conference on Artificial Intelligence-85* (pp. 1-7). Los Angeles: International Joint Conference on Artificial Intelligence.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (in Press). Cognitive modelling and intelligent tutoring. *Artificial Intelligence.*

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

Anzai, Y., & Simon, H. A. (1979). The theory of learning by doing. *Psychological Review, 86,* 124-140.

Brown, J. S., & VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4,* 379-426.

Burton, R. R., & Brown, J.S. (1982). An investigation of computer coaching for informal learning activities. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 79-98). New York: Academic Press.

Chase, W. G., & Ericsson, K. A. (1982). Skill and working memory. In G. H. Bower (Ed.), *The psychology of learning and motivation.* New York: Academic Press.

Lewis, C. H. (1978). *Production system models of practice effects.* Unpublished doctoral dissertation, University of Michigan, Ann Arbor.

Lewis, C. H. (1986). Understanding what's happening in system interactions. In D. A. Norman & S. W. Draper (Eds.), *User centered system design: New perspectives in human-computer interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Lewis, M. W., Milson, R., & Anderson, J. R. (1987). Designing a intelligent authoring system for high school mathematics ICAI: The teachers apprentice project. In Greg Kearsley (Ed.), *Artificial intelligence and instruction: Applications and methods.* Reading, MA: Addison-Wesley.

Luchins, A. S. (1942). Mechanization in problem solving. *Psychological Monographs, 54,* (248).

Matz, M. (1982). Toward a process model for high school algebra. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems.* New York: Academic Press.

McCloskey, M. (1983). Intuitive physics. *Scientific American, 24,* 122-130.

Pirolli, P.L. (1985). *Problem solving by analogy and skill acquisition in the domain of programming.* Unpublished Ph.D. thesis, Carnegie-Mellon University.

Reiser, B. J., Anderson, J.R., & Farrell, R. G. (1985). Dynamic student modelling in an intelligent tutor for LISP programming. In *Proceedings of International Joint Conference on Artificial Intelligence-85* (pp. 8-14). Los Angeles: International Joint Conference on Artificial Intelligence.

Ross, B. H. (1984). Remindings and their effects in learning a cognitive skill. *Cognitive Psychology, 16,* 371-416.

Shultz, T. R. (1982). Rules of causal attribution. *Monographs of the Society for Research in Child Development, 47*(1, Serial No. 194).

Siegler, R. S. (1976). The effects of simple necessity and sufficiency relationships in children's causal inferences. *Child Development, 47,* 1058-1063.

Sleeman, D., & Brown, J. S. (Eds.). (1982). *Intelligent tutoring systems.* New York: Academic Press.

VanLehn, K. (1983). *Felicity conditions for human skill acquisition: Validating an AI-based theory* (Tech. Rep. No. CIS-21). Palo Alto, CA: Xerox Parc.

VanLehn, K. A. (1985). *Arithmetic procedures are induced from examples* (Tech. Rep. No. ISL-12). Palo Alto, CA: Xerox Palo Alto Research Center.

Zhu, X. & Simon, H. A. (1987). Learning mathematics from examples and by doing. *Cognition and Instruction, 4,* 137-166.