This bug only does the write-nine half of borrowing across zero. It changes the zero to nine, but does not continue borrowing to the left. Because it does do half of borrowing across zero, it is likely that subjects with this bug have been taught borrowing across zero. But it is also clear that they did not acquire all of the subprocedure, or else forgot part of it. If the subtraction curriculum was such that teachers first taught one half of borrowing across zero and some weeks later taught the other half, then one would be tempted to account for this bug with perfect prefix learning. But borrowing across zero is in fact always taught as a whole (as nearly as I can tell from examining arithmetic textbooks). So some other formal technique, deletion, is implicated in this core procedure's generation.

Deletion is easily formalized by an operator that mutates a given core procedure to produce another. But formalizing perfect prefix learning requires a new technique. In order to capture the belief that perfect prefix learning depends on the sequence of instruction used with the skill, it cannot be captured as an operator on core procedures (Keil's approach and the one followed in the original version of Repair Theory), nor as constraints on just a set of core procedures (the pure Chomskyan approach). Also, it is premature to actually construct a learning algorithm that generates core procedures given something representing classroom instruction. More must be understood about the constraints on learning before such a project can be attempted. The approach used in the current version of Repair Theory is to capture perfect prefix learning with *constraints on instruction/knowledge pairs*. This is the Chomskyan approach, but applied to sets whose elements are not core procedures, but pairs consisting of a core procedure and a prefix of the instructional sequence.

The goal is to find constraints such that only the "good" core procedures are paired in the set. To be a "good" core procedure, not only must it occur in the sense that the bugs that are derived from it by repair occur or are plausible predictions, but *all of the core procedures that are derived from it by deletion must also occur*. When the output of perfect prefix learning is amplified by deletion, it must produce only correct predictions.

There is a potential confusion surrounding the deletion operator. Since it acts upon the output of perfect prefix learning, one might plausibly interpret these comments as making a chronological and almost mechanistic claim: A student first learns perfectly, only to have some process come along later and "clobber" part of the memory for the procedure. Such a connotation goes far beyond the claims of the theory. The intent in separating perfect prefix learning and deletion is to allow formulation of constraints that accurately express the dependence of acquisition on instruction, both perfect acquisition and incomplete acquisition.

So far, three constraints on perfect prefix learning have been identified. They all depend crucially on properties of the representation language. Indeed,

the representation language has been chosen in order to allow these constraints to be chosen (cf. the discussion of explanatory adequacy on pp. 214–215). All three constraints need to mention the input to perfect prefix learning, which is supposed to represent instruction. For the sake of stating these constraints, a very simple representation will be used. It relies on the fact that instruction in procedural skills like subtraction is almost always divided into segments that introduce a new aspect of the algorithm and then drill the students on it. In a typical textbook presentation of subtraction, the segment for borrowing from zero is two pages: one page explains the new subprocedure and shows the sequence of actions needed to solve an example problem and the other is a page of exercises for the student to solve. Other typical segments are borrowing in two-column problems, borrowing in three-column problems where the borrow is initiated in the tens column, and adjacent simple borrows in three-column problems. These examples are cited to illustrate the grain-size of the formalization. In this chapter, that is all that is needed of the instruction formalization.

Given the notion of segments, the three constraints that are the targets of the arguments of the next section can be stated. The first defines what it means for a core procedure to be "perfect" vis a vis a certain prefix of the instruction:

Perfect prefix learning: If x is an example or exercise problem of segment s, and core procedure p is paired with s by perfect prefix learning, then p solves x correctly and without reaching an impasse.

The remaining two constraints describe the fact that subtraction learning is incremental. It is plausible that an individual subject moves through the sequence of core procedures as he or she moves through the curriculum. One would expect the structure of each core procedure to somehow embed the structure of its predecessor. That is, newly learned components are added on top of the existing components without changing them, or perhaps with only minimal changes. This kind of learning is usually called *assimilation*, to distinguish it from more radical kinds of learning. The hypothesis that core procedure learning is assimilation is expressed formally by the following constraint:

The assimilation constraint: If segment s' is the successor of s in the teaching sequence and pairs $\langle p,s\rangle$ and $\langle p',s'\rangle$ are generated by perfect prefix learning, then core procedure p is a proper subgraph of core procedure p'.

This constraint (and the next one as well) uses the fact that a core procedure as represented in the formal knowledge representation language happens to be a labeled, directed graph. Hence, the convenient notions of subgraph and subgraph difference can be appealed to in order to express formally the relationship between the old core procedure and the newly acquired one that is built on top of it. (Actually, a much more technical definition is necessary to be completely accurate. See the discussion of "maximal partial isomorphism" in VanLehn &

Brown, 1980.) Whereas the assimilation constraint says that learning makes only small changes to existing structure, the next constraint says that the amount of new material that can be learned is simple, in a certain sense.

The disjunction-free learning constraint: If segment s' is the successor of s in the teaching sequence and pairs $\langle p,s \rangle$ and $\langle p',s' \rangle$ are generated by perfect prefix learning, then the difference subgraph $p'-p$ contains at most one disjunctive goal, namely the one adjoining it to p.

What this constraint says is that the new material can contain no disjunctions, although a disjunction can be used to attach it to the old material. The intuition behind the disjunction-free learning constraint is that when teaching a conditional or "branch statement" in a procedure, one must first teach one side of the conditional, then the other; they can't both be taught at once. In subtraction, for example, one teaches borrowing from a nonzero digit in one segment and borrowing from a zero in another. A new disjunctive test for zero adjoins the new borrow-from-zero subprocedure, which is free of disjunctions, to the old material as an alternative to borrowing from a nonzero digit.

Stepping back to look at these three constraints as a whole, one sees that the perfect prefix learning constraint sets up a well-defined relationship between core procedures and instruction, whereas the other two, assimilation and disjunction-free learning, deliver the punch line. They begin to suggest something about potential learning mechanisms. The disjunction-free learning constraint is particularly provocative. Computational experiments with a variety of knowledge representations have shown that induction from examples is simple when the language does not allow disjunction (Dietterich & Michalski, 1980; Hayes-Roth & McDermott, 1976; Mitchell, 1978; Vere, 1978; Winston, 1975). However, when disjunction is entertained, domain-specific constraints are needed to select among the multitude of inductions that can be made—domain independent constraints, such as "simplicity," appear to be too weak (Feldman, 1972; Pinker, 1979). Combinatorial explosion continues to be a problem even when the input to the inductive algorithms contains feedback from a teacher concerning negative examples (i.e., the learner is told that the given instance is *not* an example of the concept being induced). In theory, negative examples are sufficient to guarantee convergence of induction in the limit (Gold, 1967). In practice, they have been found to be helpful, but not helpful enough at taming the combinatorics of learning when disjunction is allowed (Knobe & Knobe, 1976). In short, there is every reason to believe from an algorithmic point of view that induction must either be disjunction-free or subject to strong domain-specific constraints. By hypothesis, nonnatural knowledge does not have strong domain-specific constraints on its acquisition. Therefore, if an inductive algorithm is the underlying learning mechanism, there would have to be a disjunction-free learning con-

straint on acquisition. To put it differently, if no constraint like disjunction-free learning can be sustained, then doubt will be cast on any inductive explanation for learning nonnatural knowledge.

There are two comments to make with respect to testing these constraints. First, the data collection procedures for bugs did not record the instructional segment that each subject was in at the time of testing. Consequently, it has been necessary to guess which segments go with which core procedures. In one respect, this is just a mistake which has weakened the inferences that can be drawn from the data. On the other hand, it is a stroke of luck. Relying on anecdotal evidence, I suspect that the teacher's instruction to the class as a whole may be in a certain segment, but some students behave as if they are in some earlier segment. For some reason, they either did not learn or chose not to use the subskills that were taught most recently. If this phenomenon exists, it would not affect the constraints on core procedure generation. The same set of core procedure/segment pairs could be used. However, being unable to use the classroom's segment as the segment of each student in the classroom would complicate the mapping between data and predictions.

The second comment is that these constraints are highly dependent on the representation used for core procedures. Indeed, all parts of the theory depend strongly on the knowledge representation for core procedures. Not only are the acquisition constraints highly sensitive to the syntax of the expressions (procedures) of this language, but the interpreter is designed specifically to execute procedures written in the language, and the repairs (and the critics) operate on the run-time state of the interpreter, which is in turn specified by the language. The influence of the language is felt everywhere in the model. Indeed, the key point of this chapter is that: *Getting the knowledge representation language "right" allows the model to be made simple and to obey strong principles.*

My experience has been that incremental redesign of the representation language was absolutely necessary to bring the model into conformance with psychologically interesting principles. The evolution of the principles of the theory went hand in hand with the evolution of the representation language. The objective of this chapter is to analyze the crucial points in that evolution.

THE GOAL STACK

One of the earliest and most fundamental changes in computer programming languages was the move from register-oriented languages to stack-oriented languages. In register-oriented languages, one represents programs as flow charts or their equivalent. The main control structure is the conditional branch. Data flow is implemented as changes to the contents of various named registers. Stack-oriented languages added the idea of a subroutine—something that could

be "called" from several places and when it was finished, control would return to the "caller" of the subroutine. Although the register-oriented languages need only a single register to keep track of the control state of the program, stack-oriented languages need a last-in-first-out stack so that the interpreter can tell not only where control is now (the top of the stack), but where it is to return to when the current subroutine gets done (the next pointer on the stack), and so on. Stack-orientation also augmented the representation of data flow. A new data flow facility was to place data on the stack, as temporary information associated with a particular invocation of a subroutine. In particular, subroutines could be called with parameters (arguments).

The fundamental distinction between register-orientation and stack-orientation has lapsed into historical obscurity in computer science, but surprisingly, psychology seems ignorant of it. When a psychologist represents a process, it is frequently a flow chart, a finite state machine, or a Markov process that is employed. Even authors of production systems, who are often computer scientists as well as psychologists, give that knowledge representation a register orientation: Working memory looks like a buffer, not a stack, and productions are not grouped into subroutines. For some reason, when psychologists think of temporary memory, whether for control or data, they think only of registers.

There are well known mathematical results concerning the relative power of finite state automata, register automata, and push-down automata. Some of these results have been applied to mental processes such as language comprehension (see Berwick, in press, for a review). However, I find myself rather unconvinced by such arguments. As Berwick and others have pointed out, these arguments must make many assumptions to get off the ground, and not all of them are explicitly mentioned, much less defended.

The project of this section is to argue for a stack-based representation of core procedures based on a rich structure of motivated assertions: the principles and architecture of Repair Theory as presented in the preceding two sections. Those sections did not make assumptions about the representation language because they dealt with the facts at a medium-high level of detail. The arguments in this section and the next show what must be assumed of the representation language in order to push the structure of the preceding sections down to a low enough level that precise predictions can be made, and made successfully. That is, they show what aspects of the mental representation are *crucial* to the theory.

The basic tools of complexity-based arguments on mental representations (cf Berwick, in press) are time and space bounds on the computation. These are risky precisely because we do not know what the processor and memory of the mind are like, or even if VanNeuman machine architecture (i.e., one processor, one memory, serial computation) is appropriate for measuring resource limitations on mental processing. In the near future, it appears that computers will be available with radically different architectures. For example, instead of one

processor and a large memory, there may be thousands of processors, each with a small amount of memory. These architectures are expected to radically change the speed and memory characteristics of computations. Crucially, they are not expected to change *what* can be computed but only what can be computed within certain resource limitations. So, one can assume a VonNeuman architecture is safe from technologically stimulated changes in metaphors, as long as one only cares about what the computations are and not what resources they require.

In the following sections, resource limitations are never used as tools of argumentation. Instead, the tools are two operators that, by hypothesis, manipulate the mental representation directly. One is a certain repair that manipulates the execution state, and the other is the deletion operator that mutates core procedures. By studying the kinds of changes they make, their computations, one can understand what requirements must be placed on the mental representations that they manipulate.

The Backup Repair is Necessary

Control structure is not easily deduced by observing sequences of actions. Too much internal computation can go on invisibly between observed action steps for one to draw strong inferences about control flow. What is needed is an event that can be assumed or proven, in some sense, to be the result of an elementary, indivisible control operation. The instances of this event in the data would shed light on the basic structures of control flow. Such a tool is found in a particular repair called the *backup repair*. It bears this name since the intuition behind it is the same as the one behind a famous strategy in problem solving: backing up to the last point where a decision was made in order to try one of the other alternatives. This repair is so crucial in the remaining arguments that it is worth a few pages to defend its existence.

The existence argument begins by demonstrating that a certain four bugs should all be generated from the same core procedure. There are two arguments for this lemma, one based on explanatory adequacy, the other on observational adequacy. From the lemma, it is argued that the backup repair is the most observationally adequate way to generate the four bugs.

For easy reference, the four bugs will be broken into two sets called big-borrow-from-zero and little-borrow-from-zero. Big-borrow-from-zero bugs seem to result from replacing the whole column processing subprocedure when the column requires borrowing from a zero. Its bugs are

		3	1
Smaller-From-Larger-Instead-of-	3 4 5	3 4¹5	2¹0 7
Borrow-From-Zero:	−1 0 2	−1 2 9	−1 6 9
	2 4 3 ✓	2 1 6 ✓	4 2 ×

Zero-Instead-of-	3 4 5	3 4¹5	2¹0 7
Borrow-From-Zero:	−1 0 2	−1 2 9	−1 6 9
	2 4 3 √	2 1 6 √	4 0 ×

When a column requires borrowing from zero, as the units column does in the last problem, the first bug takes the absolute difference instead of borrowing and taking a regular difference, whereas the second bug just answers the column with the maximum of zero and the difference, namely zero.

The little-borrow-from-zero bugs have a smaller substitution target. Only the operations that normally implement borrowing across the zero are replaced, namely the operations of changing the zero to nine and borrowing from the next digit to the left. Its bugs are

Borrow-Add-Decrement-	3 4 5	3 4¹5	2 0¹7
Instead-of-Zero:	−1 0 2	−1 2 9	−1 6 9
	2 4 3 √	2 1 6 √	5 8 ×
Stops-Borrow-At-Zero:	3 4 5	3 4¹5	2 0¹7
	−1 0 2	−1 2 9	1 6 9
	2 4 3 √	2 1 6 √	4 8 ×

In the first case absolute difference has been substituted for decrementing. Hence, the zero in the third problem is changed to the absolute difference of zero and one, namely one, during borrowing. The second bug, Stops-Borrow-At-Zero, is generated by substituting the max-of-zero-and-difference operation for decrement. This causes the bug to cross out the zero of the last problem, and write a zero over it. (Both these bugs, by the way, have other derivations than the ones discussed here.)

The cross product relationship between these four bugs is exactly the kind of pattern that the repair process captures. The most straight forward way to formalize it would be to postulate two core procedures, one for big-borrow-from-zero and another for little-borrow-from-zero. However, all four bugs miss the same kind of problems, namely just those problems that require borrowing from a zero. Intuitively, they seem to have the same cause: The subskill of borrowing across zero is missing from the subject's knowledge. It is a fact that subtraction curricula generally contain a segment that teaches borrowing from nonzero digits. Now the perfect prefix learning constraint implies that all core procedures associated with a certain segment of instruction will miss just the problems that lie outside the set of examples and exercises of that segment. If two core procedures were used and the perfect prefix learning constraint is to be obeyed, then both core procedures would have to be paired with the instructional segment that

teachers borrowing from nonzero digits. This means that whatever the mechanism is that implements perfect prefix learning, it must explain how it could generate two core procedures that differed *only by how much of the procedure was repaired.* Imposing this added task on the learner could make it more complex. So, in order to simplify the learner and maintain the perfect prefix learning constraint, the four bugs should be generated from the same core procedure.

The previous argument was based on explanatory adequacy. There is a second based upon observational adequacy. It stems from the empirical claim that all bugs in a bug migration class are generable from the same core procedure. Figure 5.3 shows the first six problems of a subtraction test taken by subject 19 of classroom 20. This third-grader gets the first four problems right, which involve only simple borrowing. He misses the next two, which require borrowing from zero. Crucially, these two problems are solved as if the subject had two different bugs from the cross product pattern. This is an instance of intratest bug migration. The fifth problem is solved by a little-borrow-from-zero bug: He hits the impasse (note the scratch mark through the zero) and repairs it by skipping the decrement, a repair that generates the bug stops-borrow-at-zero. He finishes up the rest of the problem without borrowing—Apparently he wants to "cut his losses" on that problem. On the next problem, he again hits the decrement zero impasse, but repairs it this time by backing up and taking the absolute difference in the column that originated the borrow, the units column. These repairs generate the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero. Since both bugs are in the same bug migration class, both are somehow derived from the same core procedure via repair.

Two arguments have forced the conclusion that all four bugs come from the same core procedure. Now the problem is to find repairs that will generate all of them, given that they all stem from the same impasse. One way to do that would be to use four separate repairs. However, that would not capture a fact about these bugs that was highlighted in their description: They fall into a cross product pattern whose dimensions are the "size" of the patch (i.e., just the decrement versus the whole borrowing operation) and its "function" (i.e., absolute difference versus maximum of zero and difference). It will be shown that this pattern can be captured by postulating a backup repair, and hence that approach is more descriptively adequate.

As mentioned earlier, the backup repair resets the execution state of the interpreter back to a previous decision point in such a way that when interpreta-

3 13	7 80	127	7 183	106	7 800
- 7	-24	- 83	- 95	- 38	-168
36	56	44	88	138	648

Figure 5.3. The first six problems show bug migration.

tion continues, it will choose a different alternative than the one that led to the impasse that backup repaired. The backup repair is used for the big-borrow-from-zero bugs but not the little-borrow-from-zero bugs. Using backup in those cases causes a secondary impasse. The secondary impasse is repaired with the same two repairs that are used for the little-borrow-from-zero bugs. This is perhaps a little confusing, so it is worth a moment to step through a specific example.

Figure 5.4 is an idealized protocol of a subject who has the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero. The (idealized) subject does not know about borrowing from zero. When he tackles the problem 305 − 167, he begins by comparing the two digits in the units column. Since 5 is less than 7, he makes a decision to borrow (episode *a* in the figure), a decision that he will later come back to. He begins to tackle the first of borrowing's two subgoals, namely borrowing-from (episode *b*). At this point, he gets stuck since the digit to be borrowed from is a zero and he knows that it is impossible to subtract a one from a zero. He's reached an impasse. The backup repair gets past the decrement-zero impasse by "backing up," in the problem-solving sense, to the last decision

a.
```
 305
-167
```
In the units column, I can't take 7 from 5, so I'll have to borrow.

b.
```
 305
-167
```
To borrow, I first have to decrement the next column's top digit. But I can't take 1 from 0!

c.
```
 305
-167
   2
```
So I'll go back to doing the units column. I still can't take 7 from 5, so I'll take 5 from 7 instead.

d.
```
  2
 3 05
-167
   2
```
In the tens column, I can't take 6 from 0, so I'll have to borrow. I decrement 3 to 2 and add 10 to 0. That's no problem.

e.
```
  2
 3 05
-167
 142
```
Six from 10 is 4. That finishes the tens. The hundreds is easy, there's no need to borrow, and 1 from 2 is 1.

Figure 5.4. Pseudo-protocol of a bug generated with backup repair.

which has some alternatives open. The backing up occurs in episode *c* where the subject says, "So I'll go back to doing the units column." In the units column he hits a second impasse, saying, "I still can't take 7 from 5," which he repairs ("so I'll take 5 from 7 instead"). He finishes up the rest of the problem without difficulty. His behavior is that of Smaller-From-Larger-Instead-of-Borrow-From-Zero. The other big-borrow-from-zero bug would be generated if he had used a different repair in episode *c*. (e.g, He might say, "I still can't take 7 from 5, but if I could, I certainly wouldn't have anything left, so I'll write 0 as the answer.")

It has been shown that the backup repair is the best of several alternatives that generate the four bugs. Needless to say, it plays an equally crucial role in the generation of many other bugs, but the argument stuck with just four bugs for the sake of simplicity. Backup is the tool that will be used to reveal constraints on core procedures.

Flexible Execution State is Necessary

There are several important aspects of the preceding example that indicate something about the kind of execution environment that backup operates within. The decrement-zero impasse that backup repaired occurred when the focus of attention was on the tens column. After backup had returned control back to an earlier decision (the decision about whether to borrow), the focus of attention was on the ones column. Somehow, backup knew to shift not only the flow of control, but the data flow as well. (I am assuming that the control and data flow are distinct, and that the column that is the focus of attention is held in a data flow construct—a register, variable, message, local binding, etc. There is an argument for this assumption, which is based on the ability of a competent subtractor to answer problems requiring borrowing across arbitrarily many zeros, but it will not be presented here.)

An even more impressive example of the power of backup to shift data and control flow occurs with a common core procedure that forgets to change the zero when borrowing across zero. This core procedure produces answers like (a):

(a)	(b)	(c)	(d)
2	1		0
3	0	0	0
$4^{1}0^{1}2$	$1^{1}0^{1}2$	$1\ 0^{1}2$	$1^{1}0^{1}2$
−1 3 9	− 3 9	− 3 9	− 3 9
1 7 3	1 7 3	3	7 3

The 4 was decremented once due to the borrow originating in the units column and then again due to a borrow originating from the tens column because the tens column was not changed during the first borrow as it should have been.

The crucial fact is seen in (b). Because this example is such a critical one

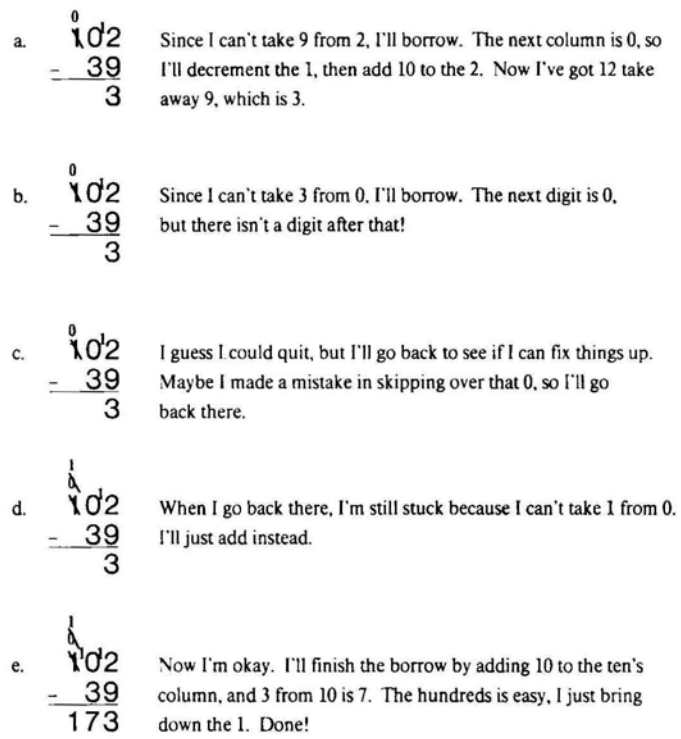


Figure 5.5. Pseudo-protocol of Borrow-Across-Zero with backup.

throughout this section, it is illustrated in Figure 5.5 with by a pseudo-protocol. The procedure decrements the one to zero during the first borrow (episode *a* in the figure). Thus, when it comes to borrow a second time, it finds a zero where the one was and borrows across it. This causes an attempt to decrement in the thousands columns, which is blank. An impasse occurs (episode *b*). The most common repair to this impasse is *quit*—a repair that just gives up on the problem. This would give the answer shown in (c). The answer shown in (b) and in Figure 5.5 is generated by assuming the impasse is repaired with backup. Backup returns to the decision made in the hundred's column concerning the zero-ness of its top digit (episode *c* in the figure). It takes the open alternative to this decision, which causes an attempt to decrement the top digit in the hundred's column. This causes a secondary impasse (episode *d*). If it is repaired one way, one sees the decrement replaced by an absolute difference as in (b) and the figure. If it is repaired a different way, one sees the familiar maximum of zero and difference patch, as in (d). The existence of several repairs to this secondary impasse confirms its existence. So, the assertion that backup returns to the decision at the

hundreds column is well-motivated. The crucial fact is that the backup repair shifted the focus even though both the source and the destination of the backing up were highly similar: They both concerned borrowing-from (as opposed to borrowing-into) and they both involved single digits rather than columns.

The stage has been set to uncover what kind of execution state the interpreter and backup are using. The execution or "run time" state of an interpreter holds whatever temporary information the interpreter needs to execute the procedure. Three alternatives will be contrasted: (*a*) minimal state, (*b*) fixed state, and (*c*) flexible state.

At a minimum, the execution state requires a register to indicate the current locus of control in the procedure. Since data flow is assumed to be independent of control flow, at least one more register would be needed to hold a pointer to the column or digit being operated on. However, this minimal control state puts an enormous burden on backup if it is to account for the facts. Since the only thing that backup has at the time that an impasse occurs is a control pointer to the action that is stuck and a data flow pointer to the column or digit that was being focused on when the impasse occurred, it can only use these as a starting point in a search through the core procedure to locate a decision point to go back to. That is, it must analyze the core procedure enough to "walk control backward" through it, and thus locate a decision to return to. Moreover, it must pay special attention to any data flow manipulations that it passes since these will have to be undone. Only by reversing the data flow can backup account for the shift in focus of attention that was observed in the examples above. Giving backup this much analytic ability means repairs cannot be considered weak and local. If one were to give repairs as much power as this version of backup requires, then it would be possible to account for virtually any subject behavior by postulating a complex enough repair. This much tailorability would render the theory vacuous.

An alternative to minimal state is to have a certain fixed number of registers, say one per "subprocedure." The idea is that each suprocedure would have its own control and data registers. Thus, instead of analyzing the core procedure, backup would search through the registers in the execution state. It doesn't matter much what a subprocedure is, but to account for the facts just stated, borrow-from and process-column would have to be distinct subprocedures. To account for the first example (Figure 5.4), the impasse happens at borrow-from, and backup shifts back to process-column. Because they have separate data flow registers, this effects a shift from the tens column to the units column. Although multiple registers suffice to account for the first fact, they fail on the second (Figure 5.5). There the impasse occurs at borrow-from in the thousands column, but backup shifts to borrow-from in the hundreds column. Since borrow-from has only one data flow register, backup would have to do some clever analysis of the core procedure (which should not be allowed in the theory).

Other possible finite execution states are to assign a register to each data

type, which in this case means roughly one register for columns and one for digits. But this fails to account for the shift of the second example since both the impasse (decrementing a blank) and the decision point returned to (testing a digit for zero-ness) involve the digit data type. Hence, there can be only one register involved, and one is again forced to give backup the intelligence necessary to change it in order to account for the facts.

A further alternative still is to assign one control register to each column. The basic idea (which is inspired by object-oriented programming languages such as Smalltalk and Simula) is that when an impasse occurs in one column, backup moves rightward and resumes with whatever action that column's control register indicates. However, this could not account for cases where the impasse and the decision that backup returns to are in the same column. There are examples of this, but for the sake of brevity, they will not be presented here. Anyway, one could go on in this spirit, using a fixed set of registers with various semantics. However, I think the point has been made that the whole fixed state approach is flawed observationally. If it could be made to work at all, its register semantics would probably be quite implausible.

A third option for the execution state involves a capacity to hold an arbitrary amount of state that is not determined in advance of the procedure's execution. Given this faculty, one can stipulate that each time a decision is made, the control flow and the data flow are saved in a safe place in the execution state. By "safe," I mean that as control flow and data flow change later, these saved values will not be affected. This allows backup to be very simple. It searches among the saved decision points for one meeting its criterion (e.g., the most recent one). Not only does this allow backup to account for the facts just exhibited, but it makes the strong prediction that whenever backup restores control to a past decision, the data flow must be reset to whatever it was at the time of that decision. Whereas the other schemes allowed the possibility that control could be shifted independently of data—backup could choose not to reverse the data flow as it walked control backward through the core procedure—this one forces them to be shifted together, if at all. That is, instead of just accounting for the fact that control and data flow are shifted together, the flexible-state architecture *explains* it.

This kind of argument is a familiar one in computer science. It trades off increased storage of information against decreased processing on the part of backup. The more information is saved in the run time state, the less information backup has to compute. Since there are good methodological reasons for making repairs as simple as possible, this tilts the tradeoff in favor of increasing execution state. The fact that this end of the tradeoff netted us an explanation rather than a mere description of the facts of backing up indicates that this tradeoff is more than a free parameter of the theory, but integral to it.

There is a second argument for flexible execution state, which turns on explanatory adequacy. The argument shows that recursive control flow is neces-

sary for disjunction-free learning. That is, if execution state is not flexible, thus allowing the language to use recursion, then the disjunction-free learning constraint cannot be imposed on the perfect prefix learning. The argument involves learning a certain way to borrow across zero, which is exemplified in the following problem:

```
  2 9
 3¹0¹5
-1 2 9
 1 7 6
```

The zero has ten added to it, then the three is decremented, then the newly created ten is decremented. The claim is that the only way to learn this way of borrowing in a nonrecursive language violates the disjunction-free learning constraint. To make the argument crisp, a particular nonrecursive language, namely flowcharts, is used. Figure 5.6a shows borrowing from a core procedure that only knows how to borrow from nonzero digits. Figure 5.6c shows borrowing after borrowing across zero in the fashion shown earlier has been learned. Clearly, there are two branches to learn. One moves control leftward across a row of zeros, and the other moves back across them until the column originating the borrow is found (i.e., the "Home?" predicate is true of the column B). There are many other ways that borrowing could be implemented, but if recursive control is not available, they would all have to have two loops—one for searching leftward, one for searching rightward.

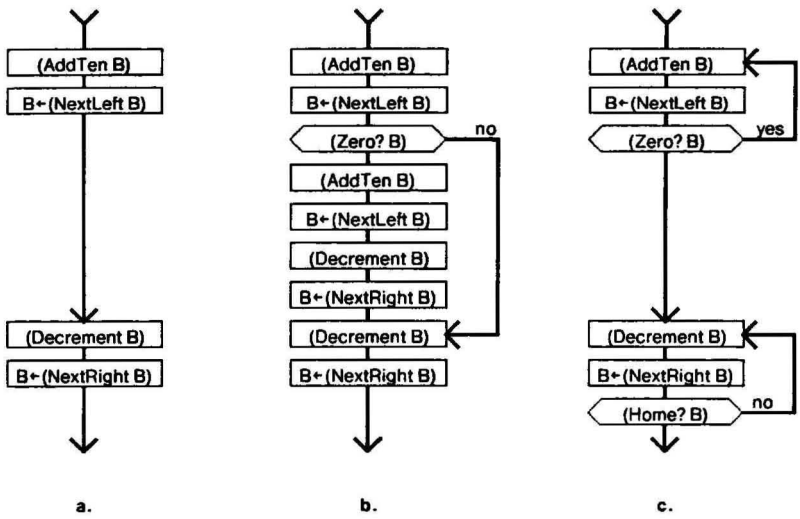


Figure 5.6. Finite state version of borrowing.

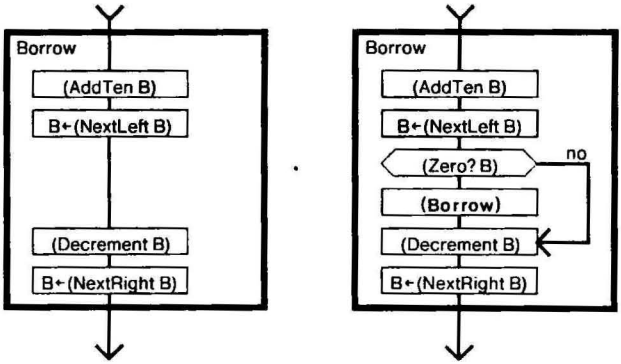


Figure 5.7. Recursive versions of borrowing.

Now disjunction-free learning sanctions the acquisition of at most one branch, and this must be one that adjoins the newly learned steps to the older material. A formal definition of branches and adjunction depends on the syntax of the language, but the essence of it should be clear by examining the difference between Figure 5.6a and 5.6b. Functionally, the difference is that the core procedure of Figure 5.6b has learned how to borrow from one zero. Syntactically, there is one branch, and it is an adjoining branch because one arm of the branch skirts the new material. The essence of adjunction is that one arm of the new conditional replicates the old procedure's control pathway.

It should be clear that the transition from Figure 5.6a to 5.6c requires adding two disjunctions and thus violates the disjunction-free learning constraint. Yet, if the language allowed recursion, then the borrowing-from-zero goal could be represented as in Figure 5.7b, with a recursive call to itself (the heavy box labeled "Borrow"). This representation allows the transition from nonzero borrowing (5.7a) to borrowing-from-zero (5.7b) to obey the disjunction-free learning constraint. Recursion requires a flexible execution state because the control state must be saved across the recursive call to borrow. In short, the language must have a flexible-state control structure so that a certain acquisitional transition obeys the disjunction-free learning constraint.

A Goal Hierarchy and a Stack Are Necessary

The backup repair sends control back to some previous decision. The question is, which decision? There are three well-known backup regimes used in Artificial Intelligence (AI):

Chronological backup: The decision that is returned to is the one made most recently, regardless of what part of the procedure made the decision.

Dependency-directed backup: A special data structure is used to record which actions depend on which other actions. When it is necessary to backup, the dependencies are traced back to find an action that doesn't depend on any other action. This means a decision was made (an "assumption" in the jargon of dependency-directed backtracking) whose first effect was this action. That decision is the one returned to.

Hierarchical backup: To support hierarchical backup, the procedure representation language must be hierarchical in that it supports the notion of goals with subgoals. In order to find a decision to return to, it searches up the hierarchy starting from the current goal, going up from goal to supergoal. The first (lowest) goal that can "change its mind" is the one returned to. (NB: In A.I., this is not usually thought of as a form of backup. It is usually referred to by the LISP primitives used to implement it, e.g., Catch and Throw in Maclisp.)

The argument that follows shows that the first two backup regimes have observational and descriptive problems. Hierarchical backup is the only one of the three that can generate the crucial bugs while avoiding the generation of star bugs. This conclusion has consequences for the representation language, namely that the language must force a modular or goal-subgoal structure on procedures, and furthermore, that this kind of backup forces the execution state to be a stack.

Chronological backup is able to generate the bugs mentioned earlier. The walk-through of Figures 5.4 and 5.5 should be evidence enough of that. However, by the impasse/repair independence principle, it can be used to repair any impasse, and here it has problems. When it is applied to the impasses of certain independently motivated core procedures, it generates star bugs. This point will be made with an example of one of them. It is a core procedure that is needed to generate bugs like:

		3	19
Smaller-From-Larger-	**345**	**345**	**207**
With-Borrow:	**−102**	**−129**	**−169**
	243 ✓	214 ×	32 ×
		3	19
Zero-After-Borrow:	**345**	**345**	**207**
	−102	**−129**	**−169**
	243 ✓	210 ×	30 ×

After completing the borrow-from half of borrowing these bugs fail to add ten to the top of the column borrowed into. Instead, Smaller-From-Larger-With-Borrow answers with the absolute difference of the column, and Zero-After-

Borrow answers the column with zero. Apparently, and quite reasonably, the core procedure is hitting an impasse when it returns to the original column after borrowing. Since the column has not had ten added to the top digit as it should, the bottom digit is still larger than the top. This causes the impasse, which is seen being repaired two different ways in the two bugs.

When the core procedure thus independently motivated, one should find that a bug is generated when chronological backup is applied to the impasse. But since the core procedure knows how to borrow across zero, at the time it reaches the impasse, its most recent decision was that the digit that it was to borrow from was nonzero and hence could be decremented, thereby finishing up the borrow (I'm making assumptions about the order of steps during borrowing, but these are not essential to the argument—somewhere in the middle of borrowing there will be a decision point). Since this is the most recent decision, chronological backup causes control to go back and "continue" the borrowing, even though it already has borrowed. Worse, when it gets done with this superfluous borrowing, it comes right back to the backup patch, which sends it back to borrow again! The chronological backup patch results in an infinite loop, and a rather bizarre one at that. Clearly, this is a star bug and should not be predicted to occur by the theory. Chronological backup is ruled out on grounds of descriptive inadequacy.

Dependency-direct backup is really not a precise proposal for this domain until the meaning of *actions depending on other actions* is defined. Consideration of the star bug that was just described leads to a plausible definition. Part of what makes the star bug absurd is its going back to change columns to the left of the one where the impasse occurred. It seems clear that difficulties in one column don't depend causally on actions in a different column. If dependency between actions is defined to mean that the actions operate on the same column (or more generally, have the same locative arguments), then dependency-directed backup would not make the mistake that chronological backup did. It would never go to another column to fix an impasse that occurs in this column. However, even this rather vague definition limits dependency-directed backup too strongly. Several examples of backup were presented earlier (Figures 5.4 and 5.5) where the location was shifted, and hence dependency-directed backup cannot generate these bugs. As it is presently defined, it is observationally inadequate.

In essence, these two approaches to backing up show that neither time nor space suffice. That is, such natural concepts as chronology or location will not support the kind of backing up that subjects apparently use. That leaves one to infer that they must be using some knowledge about the procedure itself. The issue that remains is what this extra knowledge is. A goal hierarchy is one sort of knowledge that will do the job, as the following demonstration shows.

The basic definition of hierarchical backup is that it can only resume decisions which are supergoals of the impasse that it is repairing. With this stipula-

tion, any one of a number of goal structures will suffice to block the star bug that chronological backup generated (as well as to generate all the backup bugs that have been presented so far). One such goal structure is shown in Figure 5.8. This figure shows the goal–subgoal relationships with arrows. In this goal structure, the borrow-from goal (the goal that tests whether the digit to be borrowed from is zero) is not a supergoal of the take-difference goal (the operation that reaches an impasse in the star bug's generation). There would be a chain of arrows from borrow-from to take-difference if it were. Hence, when backup occurs at the take difference impasse, it cannot go to the borrow-from decision even though that decision is chronologically the most recent.

In the previous section, it was shown that decisions must be saved in the execution state so that backup doesn't have to analyze the core procedure. But now a new constraint has been added to backup, namely one that it only return to decisions that are supergoals of the impasse. To maintain the constraint that the control information that repairs need is in the execution state, the set of saved decision points must be structured to reflect the goal structure. One way to do this is to remove decision points from the execution state when control ascends through their goal in the goal structure. This guarantees that the only decisions that are left in the execution state are ones that are supergoals of whatever goal is currently executing. This amounts to keeping goals in a last-in-first-out stack.

Of course, there are many more powerful control regimes than stack-based ones. As an example of the trouble more powerful control regimes cause, consid-

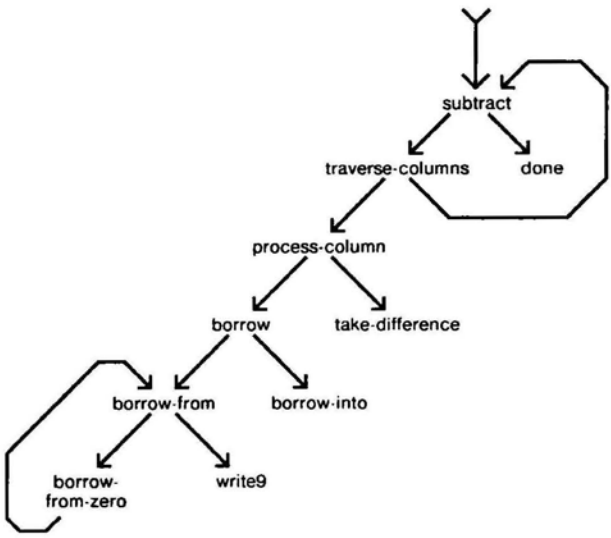


Figure 5.8. A goal–subgoal hierarchy.

er a simple one: Coroutines are a control structure that allows independent processes, each with their own stack. But this control structure increases the expressiveness of the language, which allows perfect prefix learning to generate absurd core procedures. To demonstrate this point, suppose the language allowed coroutines, and consider perfect prefix learning of simple borrowing. The instructional sequence would perhaps have pictures or blackboard demonstrations presenting the actions of borrowing in a sequence, such as:

	3	3	3
4 1 5	4 1 5	4 1 5	4 1 5
−2 9	−2 9	−2 9	−2 9
		6	1 6

Given that coroutines are allowed, one way to construe the first two actions, the new ones, is that they are a new coroutine. It happens that the example has this coroutine executing before the old one, but the learner need not take that as necessary. The core procedure could execute the coroutines interleaved, as in:

		3	3
4 1 5	4 1 5	4 1 5	4 1 5
−2 9	−2 9	−2 9	−2 9
	6	6	1 6

Because this core procedure can execute either interleaved or correctly, it seems an absurd prediction to make, a star bug. Yet with the addition of few minor constraints on sequencing the two coroutines, it meets the perfect prefix learning constraint and, I suppose, the others as well if the notion of subgraph can be defined for coroutines. In short, the use of a more powerful control regime allows perfect prefix learning to generate core procedures that it should not. Restricting the control regime to be a stack improves the descriptive adequacy of the theory by blocking the star bug. In addition, the stack constraint is in itself a restriction on acquisition and so increases explanatory adequacy as well.

SATISFACTION CONDITIONS, DELETION, AND THE APPLICATIVE CONSTRAINT

In the previous section, it was shown that the execution state should be a stack, which entails that the knowledge representation for core procedures has a goal–subgoal calling hierarchy. That is, a goal should have explicit subgoals that it can call. But this constraint leaves many issues unsettled. One open question is, when should a goal be popped from the stack? Certainly a goal is finished when all the subgoals have been tried, but are there occasions when it should exit

before trying all its subgoals? If there are, how should such information be represented? There are three reasonable possibilities:

1. Pop after *one* subgoal has been successfully executed.
2. Pop only after *all* applicable subgoals have been tried.
3. When to pop is controlled by a special information associated with each goal.

The first possibility cannot be the only exit convention used by the control structure because it will not allow expression of conjunctive goals, such as borrow, which has two subgoals that must both be executed. Consequently, it must be used along with some other exit convention, and goals must be typed to indicate which one applies. It is simplest to use it together with the second exit convention, a control regime equivalent to AND/OR graphs (Nilsson, 1971). So this control structure really amounts to giving goals either an AND or an OR type.

On the other hand, the second convention for when to pop goals can be used alone. Indeed, it is the one used by most production systems (i.e., those that use the recency and refractoriness conflict resolution principles (McDermott & Forgy, 1978) and do not erase goals from working memory). Goals do not need to be typed. Because it is simpler in that it does not require typing of goals, it is preferable to the AND/OR control regime.

The third option is a generalization of the other two. As it stands, it is just a catchall category and would have to be fully defined before it can be evaluated empirically. However, it is clearly more powerful and less constrained than the other two and should be considered only if the data force abandonment of the others (which they do, as it turns out). So, the three options, ranked in order of simplicity, are

1. AND: pop when all applicable subgoals have been tried.
2. AND/OR: goals have a binary type.
3. Otherwise.

The theme of this section is determining which of the three control structures optimizes the theory's adequacy. However, the route to resolving this question is in many ways more interesting than its answer. The preceding section uncovered some of the structure of the internal execution state by examining how repairs manipulated it. The details of how backup changed "short-term memory" revealed its structure. Here the task is similar. The exit conventions for goals are, in a sense, part of the "long-term memory" structure of goals. Repairs have been excluded from changing this structure by the core procedure immutability principle. To see into this "long-term" structure, one needs something that manipulates it or in some way depends on its form. Two will be used: perfect

prefix learning and deletion. So, part of the argumentation that resolves the exit convention controversy involves taking a stand on certain aspects of two formal devices for describing acquisition.

The AND Exit Convention Is Not Compatible with Assimilation

As a result of adopting a stack architecture in the previous section, a subset of the subtraction core procedures directly mirrors the structure of the teaching of subtraction. The segments of the subtraction curriculum are linearly ordered. It is plausible that an individual subject moves through some sequence of core procedures as he or she moves through the curriculum, assimilating new material on top of the old, with relatively little change to the old material. In order to capture this hypothesis formally as the assimilation constraint, the representation of knowledge must have a certain format. The argument that follows shows that capturing assimilation formally is incompatible and with the AND exit convention, the one used by production systems. It forces too much of the existing core procedure to be changed in order to assimilate the new material.

The problem with the AND exit convention is due to the cumbersome way that disjunctive goals must be expressed. To express the fact that two subgoals are mutually exclusive, one must put mutually exclusive "applicability" conditions on them. For example, to express the fact that there are two mutually exclusive ways to process column, depending on whether it is a two-digit column or a one-digit column, one would write

To process-column (X):
1. (blank? (bottom X)) $\Rightarrow$ (bring-down-top X)
2. (not (blank? (bottom X)) $\Rightarrow$ (take-difference X)

(In this example and the ones following it, a rule-oriented syntax has been adopted. Each subgoal is a rule, with its name and arguments on the right side. The conditions determining when the subgoal is applicable are separated from the rest of the subgoal to come a set of "applicability conditions," which are shown on the left side of the rule. The locally bound data flow that was argued for in the preceding section is implemented by giving goals arguments; in this case, X is a variable (argument) standing for the column being processed. Lisp function/argument syntax is used instead of the usual mathematical notation (i.e., f(x) is written (f x).). Nothing in the following argument depends on the adoption of a rule-oriented syntax as opposed to, for example, networks or schemata.)

Both rules must have applicability conditions in order that they by mutually exclusive. In the following problem:

37
− 4

the first rule must be prevented from applying to the units column, so its applicability condition is necessary. The second rule's applicability condition is necessary to prevent it from applying to the tens column. Because the AND exit convention tries to execute all subgoals, one can only get mutual exclusion by using mutually exclusive applicability conditions.

This implies that assimilating a new alternative method of accomplishing a goal involves rewriting the applicability conditions of the existing subgoals. If the applicability conditions are not changed, then the new subgoal will not turn out to be mutually exclusive of the old subgoal. For example, to assimilate a new method of processing columns, say one that handles columns whose top and bottom digits are equal, one would have to modify the previous goal to become

To process-column (*X*):
1. (= (top *X*)(bottom *X*)) ⇒ (write-zero-in-answer *X*)
2. (blank? (bottom *X*)) ⇒ (bring-down-top *X*)
3. *(and* (not (blank? (bottom *X*))
   (not (= (top X)(bottom X)) ⇒ (take-difference *X*)

Adding the new subgoal forced the applicability conditions of one of the existing subgoals to be changed (the italicized material was added). The essential point here is that the AND convention forces mutually exclusive alternatives to be highly interdependent. This lack of modularity defeats assimilation since assimilation is forced to modify existing material even though that material's function has not changed.

Despite the elegance of having just one exit convention and the simplicity of leaving goals unannotated, the AND exit condition must be abandoned. This implies that goals must be annotated with at least a single bit, discriminating between AND and OR, and perhaps with something rather complex.

This annotation is welcome in that it simplifies the backup repair. The function of the repair is to pop the goal stack back to the first goal that has some alternatives left to try. When goals are typed, it is trivial to tell whether a supergoal has any alternatives left. If it is an AND goal, by definition it does not. If it is an OR goal, then only one of its alternatives has been tried (because it normally pops after trying one subgoal), so all the rest must be open. If the representation language adds the syntactic constraint that all OR goals must have more than one subgoal, then backup's search becomes trivial: Pop the stack back to the first OR goal. In short, typing goals allows us to strengthen the locality constraints on repairs even more, reducing tailorability and leading perhaps to insight into the processes underlying local problem solving.

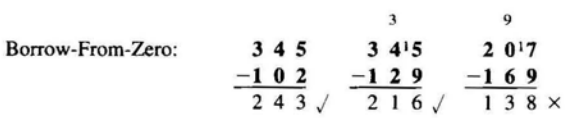
Finally, the AND/OR distinction allows the disjunction-free learning constraint to be formalized as a constraint on syntax rather than domain-dependent predicates, which is how disjunction must be represented under the AND exit convention.

Conjoined Rule Deletion

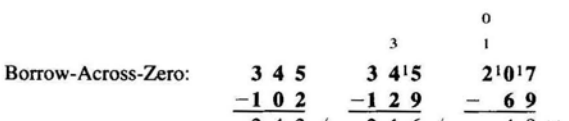
Perfect prefix learning is characterized by the constraint that all core procedures generated by it from a given prefix of the instructional sequence can answer all the exercises of that prefix correctly, without hitting impasses. If core procedures that are apparently the products of forgetting or partial learning were included in the output of perfect prefix learning, the constraint would be false. So, in order to increase the explanatory adequacy of the theory, they are generated by a separated operator. The issue is, what should that operator be? To motivate it empirically, consider the form of the following goal:

To borrow-from-zero (*X*), do all of:
1. (borrow-from (next-column *X*))
2. (write9 (top *X*))

This goal is an AND goal. When the first rule is deleted, a core procedure is generated that happens to survive application without reaching any impasses to become the bug Borrow-From-Zero:



This bug only changes the zero to nine during borrowing across zero. It doesn't borrow recursively to the left since rule 1 has been deleted. On the other hand, if rule 2 were deleted instead, it would be the writing of nine that is omitted, generating the bug borrow-across-zero



The basic idea is that rule deletion of any of the rules under the AND goal seems to generate core procedures that not only lead to bugs, but are core procedures that one cannot generate with perfect prefix learning because they do not reflect perfect learning of any known segment in the subtraction curriculum.

So, deleting rules under AND goals is good. But look what happens when rule deletion is applied under the following OR goal

To borrow-from (*X*), do one of:
1. (zero? (top *X*)) ⇒ (borrow-from-zero *X*)
2. ( ) ⇒ (decrement (top *X*))

Deleting the first rule generates a familiar core procedure, namely one that impasses whenever it is asked to borrow from a zero. Because the rule that handles borrowing from zero is missing, it tries to decrement the zero. (This happens despite the fact that rule deletion has left the goal borrow-from-zero intact; only the call to it has been deleted.) But there is no need to generate this core procedure via deletion since it is already generated by perfect prefix learning. Even worse things than redundant generation happen when the second rule or the OR goal is deleted. The following star bug is generated:

*Only-Borrow-From-Zero:	3 4 5	3 4¹5	9 2 0¹7
	−1 0 2	−1 2 9	−1 6 9
	2 4 3 ✓	2 2 6 ×	1 3 8 ×

This star bug misses all problems requiring borrowing because it never performs a decrement, despite the fact that it shows some sophistication in borrowing across zero in that it changes zeros to nines. The juxtaposition of this competency in borrowing across zero with missing knowledge about the simple case makes the behavior highly unlikely.

One possible deletion operator is to delete any rule (subgoal) of an AND goal. The rules of OR goals are not to be deleted. As the examples just given implied, this operator fits the data rather well. Note that it cannot be formulated without appealing crucially to the AND/OR distinction. Hence, this deletion operator cannot be used in a control structure that uses only the AND exit condition (i.e., it won't work with production systems). Its name reflects this dependence: conjoined rule deletion.

Satisfaction Conditions

This deletion operator is a little too unconstrained. Some of its deletions lead to star bugs. For example, the main loop of subtraction, the one that traverses columns can have the following goal structure:

To process-all-columns (*X*), do one of:
1. (not (blank? (next-column *X*))) ⇒ (column-traverse *X*)
2. ( ) ⇒ (process-column *X*)

To column-traverse (X), do all of:
1. (process-column X)
2. (process-all-columns (next-column X))

The second goal is an AND goal, so either of its subgoals can be deleted. Deleting rule 1 creates a core procedure which only answers the leftmost column in a problem. This is a star bug. Deleting rule 2 creates another stag bug, one that only does the units column.

Similar problems occur with other conjunctive goals. If the only goal types are AND and OR, borrowing must be represented with an explicit subgoal for taking the column difference after the modifications to the top row are made. This causes problems with the deletion operator. Specifically, when borrowing is represented with AND/OR control structure, as in:

To process-column (X), do one of:
1. (blank? (bottom X)) $\Rightarrow$ (bring-down-top X)
2. (less? (top X) (bottom X)) $\Rightarrow$ (borrow&take-difference X)
3. ( ) $\Rightarrow$ (take-difference X)

To borrow&take-difference (X), do all of:
1. (borrow-from (next-column X))
2. (add10 (top X))
3. (take-difference X)

deleting the third rule of the AND goal means borrow&take-difference will do all the setting up necessary to take the column difference, but will forget to actually take it. This leads to the following star bug:

		3	1 9
*Blank-With-Borrow:	3 4 5	3 4¹5	2 0¹7
	−1 0 2	−1 2 9	−1 6 9
	2 4 3 ✓	2 1 ×	3 ×

It is perhaps possible to put explicit constraints on conjoined rule deletion in the same way that deletion blocking principles were used in Brown and VanLehn (1980). This approach would probably collapse into a nest of unmotivated constraints. However, a second way to prevent overgeneration is to make the operator inapplicable by changing the type of the goal so that it is not an AND. That is, one changes the knowledge representation rather than the operator.

The proposed change is to generalize the binary AND/OR type to become "satisfaction conditions." The basic idea of an AND goal is to pop when *all* subgoals have been executed, whereas an OR goal pops when *one* subgoal has been executed. The idea of satisfaction conditions is to have a goal pop *when its satisfaction condition is true.* Subgoals of a goal are executed until either the goal's satisfaction condition becomes true or all the applicable subgoals have

been tried. (Note that this is not an iteration construct—an "until" loop—since a rule can only be executed once.) AND goals become goals with FALSE satisfaction conditions. Since subgoals are executed until the satisfaction condition becomes true (which it never does for the AND) or all the subgoals have been tried, giving a goal FALSE as its satisfaction condition means that it will always execute all its subgoals. Conversely, OR goals are given the satisfaction condition TRUE. The goal exits after just one subgoal is executed.

With this construction in the knowledge representation language, one is free to represent borrowing in the following way:

To process-column (*X*), do until (not (blank? (answer *X*))):
1. (blank? (bottom *X*)) ⇒ (bring-down-top *X*)
2. (less? (top *X*) (bottom *X*)) ⇒ (borrow *X*)
3. ( ) ⇒ (take-difference *X*)

To borrow (*X*), do until FALSE:
1. (add10 (top *X*))
2. (borrow-from (next-column *X*))

The AND goal borrow now consists of two subgoals. After they are both executed, control returns to process-column. Because process-column's satisfaction condition is not yet true—the column's answer is still blank—another subgoal is tried. Take-difference is chosen and executed, which fills in the column answer. Now the satisfaction condition is true, so the goal pops.

Given this encoding of borrowing, conjoined rule deletion does exactly the right thing when applied to borrow. Deleting the first rule generates a core procedure that hits an impasse which repair amplifies to generate several bugs, such as:

			^{3}		^{19}	
Smaller-From-Larger-With-Borrow:	345		345		207	
	−102		−129		−169	
	243	√	214	×	32	×
			^{3}		^{19}	
Zero-After-Borrow:	345		345		207	
	−102		−129		−169	
	243	√	210	×	30	×

(These bugs are generated because skipping the add10 operation causes an impasse when control returns to process the column that initiated the borrow. The top digit of that column is still larger than the bottom digit, so a repair is necessary. Different bugs result from different repairs.) Deleting the second rule generates the bug Borrow-No-Decrement:

Borrow-No-Decrement:	3 4 5		3 4^{1}5		2^{1}0^{1}7	
	−1 0 2		−1 2 9		−1 6 9	
	2 4 3	√	2 2 6	×	1 4 8	×

The point is that it is no longer possible to generate the star bug. Similarly, the star bugs associated with column traversal can be avoided by structuring the loop across columns as:

To process-all-columns (X), do until (blank? (next-column X)):
1. ( ) ⇒ (process-column X)
2. ( ) ⇒ (process-all-columns (next-column X))

By using a satisfaction condition formulation here, generation of the star bugs is avoided. From this example and the preceding ones, it is clear that augmenting the representation with satisfaction condition and using conjoined rule deletion creates an observationally and descriptively adequate treatment of perfect prefix learning and deletion.

Satisfaction conditions also play a crucial role in the formulation of empirically adequate critics. It turns out that one of the problems with critics, the so-called "blank answer critic" problem mentioned in Brown and VanLehn (1980), can be solved using satisfaction conditions. The details of this argument will be omitted here in favor of some comments of a different kind.

The Applicative Constraint

Having an operator that mutates the knowledge representation allows one to "see" the structure of the representation. An important use of this tool is to uncover one of the tacit constraints on data flow. One of the prominent facts about bugs is that none of them requires deletion of locative focus shifting functions. For example, if one knows about borrowing-from, one knows to borrow from a column to the left. No bug has been observed that forgets to move over before borrowing-from. This fact deserves explanation.

In all the illustrations so far, focus shifting functions such as (next-column X) have been embedded inside calls to actions, predicates, or subgoals. This is no accident. Suppose one did not embed them, but instead made them separate subgoals themselves, as in:

To borrow (X), do all of:
1. (add10 (top X))
2. (X ← (next X))
3. (borrow-from X)

where "←" means to change the binding (value) of the local variable X. A star bug could be generated by deleting rule 2. This star bug would borrow from the column that originates the borrow:

		14	9 16
*Borrow-From-Self:	3 4 5	3 4 5	2 0 7
	−1 0 2	−1 2 9	−1 6 9
	2 4 3 ✓	2 2 5 ×	1 3 7 ×

In order to avoid such star bugs, focus shifting functions must be embedded, so a constraint upon the knowledge representation is needed to make this explicit. About the strongest constraint one can impose is to stipulate that the language be applicative. That is, data flows by binding variables rather than by assignment. There are no side effects: A goal cannot change the values of another goal's variables, nor even it's own variables. The only way that information can flow "sideways" is by making observable changes to the external state, that is, by writing on the test page.

The applicative constraint is extremely strong, forcing data to flow vertically only in the goal hierarchy. The procedure can pass information down from goal to subgoal through binding the subgoal's arguments. Information flows upward from subgoal to goal by returning results. No counterexamples to the applicative constraint have been found in the subtraction domain.

The applicative constraint could have a profound effect on learning. It seems to make learning context-free. That is, learning a procedure becomes roughly equivalent to inducing a context-free grammar. The basic idea is that the applicative constraint together with the stack constraint force data flow and control flow to exactly parallel each other. To put in terms of grammars, the data flow *subcategorizes* the goals. This in turn makes it possible to induce the goal hierarchy from examples. Inducing hierarchy for procedures from examples has been an unsolved problem. Neves (1981) had to use hierarchical examples to get his procedure learner to build hierarchy. However, subtraction teachers do not always use such examples. Badre (1972) recovers hierarchy by assuming examples are accompanied by a written commentary; each instance of the same goal is assumed to be accompanied by the same verb (e.g., *borrow*). This is a somewhat better approximation to the kind of input that students actually receive, but again it rests on delicate and often violated assumptions. The applicative constraint cracks the problem by structuring the language in such a way that hierarchy can be learned via a context-free grammar induction algorithm (subject to the disjunction-free learning constraint, of course).

VARYING THE REPRESENTATION LANGUAGE IS THE KEY

As mentioned previously, the main methodological point of this chapter is that: *Getting the knowledge representation language "right" allows the model to be made simple and to obey strong principles.*

There have been five major changes in Repair Theory's representation language (so far). Each one had far-reaching effects on the simplicity and principledness of the model. In contrast, there has been only one change on the basic processes of the model, when perfect prefix learning was separated from rule deletion. There are several personal observations to make about this development.

Although it was intuitively obvious all along that "perfect learning" and "forgetting" could be separated, the need for changes in the representation language was never immediately obvious. The impact of knowledge representation was very subtle. Its structure was not at all apparent from the data, and intuition was no guide either. It was only by reimplementing the model to use the new representation that its effect could be evaluated.

The changes in the representation language were absolutely necessary for bringing the model into conformance with psychologically interesting principles. A typical scenario was to note a particularly ugly part of the model that could be simplified and made more elegant by changing the representation language; after the change was made, reflection upon the new elegance of the model uncovered a principle that the model now conformed to. Hard on the heels of that insight would often come several more ideas, usually of the form that changing a certain component process of the model to take advantage of the new construct in the representation language would allow it also to become simpler and to obey a new principle. In short, the evolution of the principles of the theory went hand in hand with the evolution of the representation language.

The process model and the deletion operator were implemented in a computer program—called the "workbench"—in such a way that, given a core procedure, all possible deletions and all possible repairs could be applied and evaluated automatically. The evaluation was carried out by having the parameterized model "take" a highly diagnostic subtraction test and submit its answers to DEBUGGY (Burton, 1981). DEBUGGY would diagnose the pseudo-student's answers to determine which bugs, if any, were exhibited. This program could be given a set of core procedures and left to run overnight. The morning's results revealed the observational adequacy of that set of core procedures. Although the initial investment in constructing this automatic theory evaluation system was high, it more than repaid that investment in facilitating the evaluation of subtle changes in the representation language as well as varying expressions of individual core procedures.

Although changing the representation language allows discovery of its impact on the theory, quite a bit more work was required to separate these effects from the context of their discovery in order to make well-formed arguments. To put it graphically, the journey that the model and I took through the space of representations was a single twisted path. A great deal of thought was required not only to uncover the critical junctions along that path, but to understand the *dimensions of the space*. The analysis sought to free the choices in alternative representational structures from their discovery context and present them in a neutral context, so they could be evaluated independent of each other. Once put in a neutral context, use of the medium-level of detail of Repair Theory as a rich structure of starting assumptions enabled competitive argumentation in most cases to settle the choice, establishing the particular set of choices that constitutes

the representation language at the end of the path as a global maximum, rather than as a local one, in the space of mental representations.

REFERENCES

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

Badre, N. A. *Computer learning from English* (memo ERL-M372). Berkeley: University of California, Berkeley, Electronic Research Laboratory, 1972.

Berwick, R. Cognitive efficiency, computational complexity and the evaluation of grammatical theories. *Linguistic Inquiry,* in press, *13.*

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. Repair Theory: A generative theory of bugs in procedural skills. *Cognitive Science,* 1980, *4,* 379–426.

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.

Brueckner, L. J. *Diagnostic and remedial teaching in arithmetic.* Philadelphia, Pa.: John C. Winston, 1930.

Bunderson, C. V. *Cognitive bugs and arithmetic skills: Their diagnosis and remediation* (interim tech. rep.). Provo, Utah: Wicat, April 1981.

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

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

Chomsky, N. *Aspects of the theory of syntax.* Cambridge, Mass.: MIT Press, 1965.

Chomsky, N. Rules and representations. *The Behavioral and Brain Sciences,* 1980, *3,* 1–63.

Cox, L. S. Diagnosing and remediating systematic errors in addition and subtraction computations. *The Arithmetic Teacher,* 1975, *22,* 151–157.

Dietterich, T. G., & Michalski, R. S. *Learning and generalization of structured descriptions: Evaluation criteria and comparative review of selected methods.* (Rep. 1007). Urbana: University of Illinois, Department of Computer Science, 1980.

Durnin, J. H., & Scandura, J. M. Algorithmic approach to assessing behavior potential: Comparison with item forms. In J. M. Scandura (Ed.), *Problem Solving: A structural/process approach with instructional implications.* New York: Academic Press, 1977.

Feldman, J. Some decidability results on grammatical inference and complexity. *Information and Control,* 1972, *20,* 244–262.

Gold, E. M. Language identification in the limit. *Information and Control,* 1967, *10,* 447–474.

Greeno, J., & Brown, J. S. *Theories of competence.* Paper presented at the Sloane conference in Boulder, Colo., 1981.

Hayes-Roth, F., & McDermott, J. Learning structured patterns from examples. *Proceedings of the Third International Joint Conference on Pattern Recognition,* Stanford, Calif., 1976, 419–423.

Keil, F. C. Constraints on knowledge and cognitive development. *Psychological Review,* 1981, *88,* 197–227.

Knobe, B., & Knobe, K. A method for inferring context-free grammars. *Information and Control,* 1976, *31,* 129–146.

Lankford, F. G. *Some computational strategies of seventh grade pupils.* Charlottesville: University of Virginia, 1972. (ERIC Document)

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.

Mitchell, T. M. *Version Spaces: An approach to concept learning* (Tech. Rep. 78–711). Stanford, Calif.: Stanford University, Department of Computer Science, 1978.

Neves, D. M. *Learning procedures from examples*. Unpublished doctoral dissertation, Department of Psychology, Carnegie-Mellon University, Pittsburgh, Pa., 1981.

Nilsson, N. J. *Problem-solving methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.

Norman, D. A. Categorization of Action Slips. *Psychological Review*, 1981, *88*, 1–15.

Pinker, S. Formal models of language learning. *Cognition*, 1979, *7*, 217–283.

Roberts, G. H. The failure strategies of third grade arithmetic pupils. *The Arithmetic Teacher*, 1968, *15*, 442–446.

VanLehn, K. *Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills* (Tech. Rep. CIS-11). Palo Alto, Calif.: Xerox Palo Alto Research Centers, 1981.

VanLehn, K., & Brown, J. S. Planning Nets: A representation for formalizing analogies and semantic models of procedural skills. In R. E. Snow, P. A. Federico, and W. E. Montague (Eds.), *Aptitude, learning and instruction: Cognitive process analyses*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1980.

Vere, S. A. Inductive learning of relational productions. In D. A. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed inference systems*. New York: Academic Press, 1978.

Winston, P. H. Learning structural descriptions from examples. In P. H. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975.

Young, R. M., & O'Shea, T. Errors in children's subtraction. *Cognitive Science*, 1981, *5*, 153–177.