6 Goal Reconstruction: How Teton Blends Situated Action and Planned Action

Kurt VanLehn
William Ball
Learning Research and Development Center and Computer Science Department, University of Pittsburgh

When you purchase a programming language, what you actually receive is a program (an interpreter or a compiler) that causes text written in the programming language to control the actions of the computer. When you buy an expert system shell or an AI programming environment, you get not only an interpreter but a variety of other programs as well. For instance, you often receive inference engines, data base management tools, graphics packages and libraries of utility programs. When you obtain a symbolic architecture, you receive an interpreter and some extra capabilities, most of which are not found even in the most advanced expert system shells. For instance, in the Pittsburgh architectures—Soar (Rosenbloom, Newell, & Laird, chap. 4 in this volume), Prodigy (Carbonell, Knoblock, & Minton, chap. 9 in this volume) and Theo (Mitchell et al., chap. 12 in this volume)—the extra capabilities include a kind of dynamic optimization. Programs automatically get faster as they run. From a purely pragmatic view, a symbolic architecture is just an expert system shell with some novel features added. Even architectures that are intended to model human cognition resemble augmented expert system shells. If one removed the automatic learning from ACT* (Anderson, 1983), it would be indistinguishable from many expert system shells on the market, because they too have a semantic net database and a production system programming language. On the other hand, the current state of the art is merely a stage in the development of much more powerful architectures. Architectures may evolve to the point where they are no longer programmed but instead acquire expertise through training and experience in much the same way that humans do. However, it is fair to say that we do not yet have such general problem solvers. Currently, an architecture is a programming language with some powerful, unusual extra capabilities.

This chapter discusses an extra capability that few architectures have, even though it is both useful from a programming point of view and arguably a good approximation to a human capabilities. People can reconstruct goal structures and other aspects of their internal state that have been forgotten. For instance, suppose one is interrupted in the middle of solving a difficult problem by a long involved phone call. When the phone call is over, one can eventually pick up the problem solving where one left off. This capability is called *goal reconstruction.* Because goal reconstruction requires no special training to acquire it and it does not have to be acquired separately for each a new problem solving procedure one learns, goal reconstruction is arguably a fundamental, task-general capability of human problem solvers. Goal reconstruction is also a useful capability even for an artificial problem solver. It permits recovery from interruptions of the problem solving by processes that modify the body of procedural knowledge, such as an inferential learning process or a programmer debugging the procedural knowledge. In short, goal reconstruction is both a fundamental human capability and a useful capability for AI architectures.

Goal reconstruction is part of the larger process of maintaining a goal structure. Our analysis of goal reconstruction is based on the insight that goal maintenance is a special case of the notorious frame problem in AI. The frame problem is to keep a model of the world up to date as actions take place in the world. Sometimes actions have unexpected and wide-ranging effects, which may make it difficult to calculate how much of the model needs changing in order to reflect the change wrought by the action on the real world. Of course, if the agent can see the world, then perceptual processing can be partially substituted for the cognitive processing that calculate updates to the model. At first glance, the frame problem has nothing to do with goal maintenance. Goals are not usually though of as being a part of the real world, so literally speaking, maintenance of goals is not maintenance of an internal model of the external world. However, the agent's knowledge, when viewed as a disembodied logical system, can be applied to the external world in order to generate a virtual or ideal goal structure. For the sake of the analogy, we can pretend that this Platonic goal structure is "in the real world".[1] Now it is clear that maintaining the agent's internal goal structures is exactly a frame problem: it must manipulate its internal goal structures so that they accurately reflect changes in the external, Platonic goal structures. As always in the frame problem, perceptual processing can be substituted, at least in principle, for internal calculations. This chapter discusses computa-

[1]Many AI problem solvers assume that high level descriptions, such as "block-1 supports block-2," are part of the real world. In fact, a robot would have to infer such relationships with the aid of a sophisticated vision system. Goals are also produced by inferences. So it is not such a great leap to consider goals as well as "block-1 supports block-2" relationships as being "in the real world".

tional mechanisms for implementing this "in principle" tradeoff between perceptual and cognitive maintenance of goals.

THREE PROBLEMS TO BE SOLVED BY GOAL RECONSTRUCTION

Goal reconstruction is a solution to three problems in cognitive theory. Two of the problems stem from inadequacies in current accounts of human working memory for goals. The third problem is that current accounts of problem solving overemphasize planning and plan-following, because much of human behavior is *situated* as opposed to being *planned*. This section contains a discussion of each problem in turn.

The first problem has to do with working memory capacity for goals. People cannot remember arbitrarily large goal structures for arbitrarily long times. For instance, a telephone call often causes one to forget one's place in a problem. An early approach to modeling this human trait was to assume that goals were held in a capacity limited memory, called STM or working memory. For instance, Newell and Simon (1972, p. 808) claimed that "STM holds about five to seven symbols, but only about two can be retained for one task while another unrelated task is performed." Because working memory holds both goals and intermediate results, and these can accumulate quickly while problem solving, it is difficult to perform significant computations when working memory is strongly capacity limited. Thus, it was assumed that people use the external world as a storage place for temporary results while problem solving, and this makes it just like a working memory. For instance, Newell and Simon say (p. 801) that the operative "STM should be defined, not as internal memory, but as the combination of (a) the internal STM (as measured by the usual psychological tests) and (b) the part of the visual display that is in the subject's foveal view." Thus, instead of trying to remember an intermediate result, such as $T = 0$, the person writes it down on a worksheet. Things are not so simple for goals, however, because people do not usually write goals on their worksheets. Anderson (1983, p. 161) showed how to reconstruct Tower of Hanoi goals using task-specific knowledge about the puzzle, but he did not present a general capability. Thus, goal reconstruction has been thought for some time to be important as a way of increasing the effective capacity of working memory, although a general model of goal reconstruction was never developed.

Another problem in cognitive theory involves the access characteristics of goal memory. In most models of human goal storage, goals are held in a last-in-first-out goal stack (Laird, Newell, & Rosenbloom, 1987; Newell & Simon, 1972; VanLehn, 1990). That is, when a person is done with a goal and needs to select a new goal to work on, the only goals that can be selected are those that were most recently created and are not yet accomplished. This restriction is called the LIFO (last–in–first–out) convention. Consider, for instance, a cognitive procedure with the following goal structure:

Top goal
  Subgoal 1
    Sub-subgoal A
    Sub-subgoal B
  Subgoal 2
    Sub-subgoal C
    Sub-subgoal D

Suppose all these goals are conjunctively related, so that achieving the top goal means that all the subgoals must be achieved. Suppose further that the lowest goals, the ones with letters as their names, correspond to physical actions that an experimenter could observe the subject doing. Let us see what kinds of goal selection orders are allowed by the LIFO restriction. Suppose the top goal constructs subgoals 1 and 2 at the same time. Subgoal 2 is selected, and constructs goals C and D at the same time. Goal C is selected. After it is finished by performing some physical action, the subject must choose either goal 1 or goal D, as these two have been constructed but not yet executed. The LIFO restriction implies that goal D must be chosen, as it is younger. Thus, in a LIFO architecture, the experimenter would never see actions in the sequence CADB, as this interleaves subgoals of goals 1 and 2. Of the 24 possible permutations of the four primitive goals, only 8 can be generated by a LIFO architecture.

Intuitively, the LIFO restriction is quite implausible. It essentially says that there are some subgoals that one can recall but cannot select. In the example above, one can recall subgoal 1 (since it will be selected later) and yet one cannot select it because subgoal D is younger. For instance, suppose the top goal is "do evening chores" and subgoals 1 and 2 are, respectively, "clean breakfast dishes" and "prepare dinner." A LIFO restriction would mean that one would have to clean all the breakfast dishes before starting the dinner preparation, or vice-versa. On this analysis, many people violate the LIFO restriction nightly.

The problem with the evening–chores example is that we do not really know what the goal structures of the subjects are. There are other goal structures than the one above that would allow a LIFO architecture to interleave dish–cleaning actions with dinner-preparing actions.

There are, however, good examples of the LIFO constraint being violated. We discovered eight elementary school students (from a sample of 26) who executed subtraction procedures in a non-LIFO order (VanLehn, Ball, & Kowalski, 1989). The goal structure of subtraction procedures is quite well understood (VanLehn, 1990), and there is no reason to believe that these students' goal structures were any different from their peers'. If the eight students did have one of the standard goal structures, then the sequence of physical actions they made could only be accomplished by violating the LIFO constraint.

Moreover, there were strong regularities in the 8 students' actions that make it highly unlikely that their behavior is due to working memory failures wherein a basically LIFO goal storage mechanism "accidentally" marks the wrong goal as most recent. This source of non-LIFO execution should appear as random "point" mutations to the standard execution sequence and it should also be fairly infrequent. This was not what the eight students did. They generally had two or more stable execution orders, some of which could only be generated by a non–LIFO architecture. For instance, one student had three stable orders:

1. The standard order. Columns are processed right-to-left, and the borrowing for one column is finished before the next column is begun.
2. Horizontal order. All the borrowing in the problem is done on a right-to-left horizontal pass across the columns. Then the columns are answered on a second horizontal pass, which may be either right-to-left or left-to-right.
3. Vertical order. Columns are processed in right-to-left order. However, borrows are not completed before moving on to the next column. Instead, all marks in column, including any marks caused borrows from earlier columns, are done together.

The student used the standard order on four problems, the horizontal order on four problems, the vertical order on three problems, and a blend of the horizontal and vertical orders on two problems. The systematicity of her behavior makes it implausible that her non-LIFO orderings are based on working memory failure.[2]

These eight students provide clear examples of violations of the LIFO constraint. They allow us to conclude what was intuitively obvious all along: people can select any goal for execution that they can recall. Whether or not it is sensible to make a non–LIFO choice is, of course, task specific. The reason the LIFO constraint has survived as long as it has in models of the architecture is due to the structures of the task domains, which generally require or encourage a LIFO selection of goals. Subtraction, which is not one the task domains typically studied in the architecture literature, does not have this LIFO property.

This work shows that the *operative* working memory is non–LIFO, but as Newell, Simon, Anderson, and others have pointed out, the operative working memory is implemented in part by visual perception. It could still be the case that

[2]One might think that this subject has three distinct subtraction procedures, one for each order. However, this would not explain her ability to blend the horizontal and vertical orders, as she does on two problems. For more discussion of this and other challenges to the conclusions, see VanLehn, Ball, and Kowalski (1989).

internal working memory is LIFO and that the non–LIFO aspects of the subtraction subjects' behavior is due to the way they infer or reconstruct goals from what they see. This led us to investigate the process of goal reconstruction.

A third problem in cognitive theory comes from recent work in robotics and ethnomethodology. Several investigators have worried that real-time, adaptive control of behavior does not allow for interleaving planning and plan following. Instead, people just act. As Agre and Chapman (1987, p. 268) put it, "Rather than relying on reasoning to intervene between perception and action, we believe activity mostly derives from very simple sorts of machinery interacting with the immediate situation. This machinery exploits regularities in its interaction with the world to engage in complex, apparently planful activity without requiring explicit models of the world." This belief that action is derived by cursory examination of the situation rather than reasoning is often called the situated action paradigm (Suchman, 1987).

It would be wrong to think that the proponents of situated action claim people's mental apparatus makes it impossible for them to plan their actions. As Agre and Chapman (1987, p. 272) put it, "We do *not* believe that the human central system has no state. Our point is simply that state is less necessary and less important than is often assumed." Currently developed computational models of situated action (Agre & Chapman, 1987; Brooks, chap. 8 in this volume) are claimed to be interesting architectures *for robotics* and not literal models of human cognitive capabilities. These architectures have so little internal state that they cannot model simple tasks, such as counting or mental multiplication, that humans can easily perform. Even mundane tasks, which are intended to be the forte of these architectures, sometimes cannot be done in a purely situated way. For instance, one of us once had a job washing glassware in a medical laboratory. The procedure was to wash the glassware 6 times in tap water then 6 times in distilled water. Since one cannot tell by looking at a piece of glassware how many times it has been washed, the Pengi architecture (Agre & Chapman, 1987) cannot solve this task.

Suchman (1987) does take situated action as an account of human behavior, so her position is more complex than her robotics colleagues. Suchman points out that people *do* plan, as for example, when they study a river rapids in order to plot a course for their canoe. However, these plans "are constituent as an artifact of our *reasoning about* action, not as the generative *mechanism of action.*" (Suchman, 1987, p. 39, emphasis original) Suchman is mostly concerned with plans derived as post-hoc explanations of behavior, so her book does not contain a clear statement about the causal entailments of plans made in advance of an action. Her choice of a canoeing example suggests that she does believe that advance planning can effect actions, albeit indirectly: planning to paddle to the left around a boulder in the rapids is one factor involved in causing the ultimate action of paddling to the left of the boulder. Suchman's major point, however, is that advance planning is rare, and even when it does occur, "plans are best viewed as a weak resource for what is primarily *ad hoc* activity" (bid, p. ix).

The situated action position is certainly partially right, because current models of the human problem solving have emphasized planned action rather than situated action. In part this is due to their historical roots, which lie in studies of people working with puzzles, mathematical problems and other tasks where planned actions are probably more common than situated actions. The problem for cognitive modeling is to develop an architecture that can easily and seamlessly oscillate between planned action and situated action, since both occur in human behavior and we are often not even aware, even in retrospection, of the transitions between them (Suchman, 1987).

We believe that goal reconstruction is exactly what is needed for this seamless oscillation between situated and planned action. We describe an architecture that can operate with almost no internal state by rapidly reconstructing whatever goals are necessary in the current situation. We demonstrate that these goal reconstruction processes are formally identical to processes for perceptual parsing of the situation, so goal reconstruction can be thought of as high-level perception. This nicely captures the principal intuition of the situated action paradigm, which is that much action is guided by perception. On the other hand, when goals can be recalled or when they *must* be recalled, the architecture can do that as well. So it can develop plans in memory and follow them. Moreover, this sort of planned activity blends seamlessly into situated activity.

In short, goal reconstruction is claimed to be a solution to these three problems in cognitive modeling: How do people access more goals than they can reliably store in memory? How do people implement a non-LIFO goal store? How do people blend situated and planned action?

RECONSTRUCTION IN SEVERAL PROBLEM-SPACE ARCHITECTURES

Goal reconstruction depends strongly on interpretation of visual scenes, so it would seem that any model of goal reconstruction should include at least a rudimentary model of perception. However, it is convenient to start the discussion by ducking the question of perception entirely. In this section, an initial mechanism for goal reconstruction is developed. In the next section, the initial model is augmented with a rudimentary model of perceptual processing.

A standard way to avoid modeling perception (and motor control as well—but that is irrelevant to this paper) is to use a problem space. In order to model a given task, the theorist specifies a set of primitive predicates and a way of composing them into descriptions of a problem state. In the model, the *current problem state,* which is one of these compositions of primitive predicates, represents that which the person infers from perceiving the real problem state. Thus, the problem space technique avoids the perception issue by postulating the output from the perceptual interpretation processes without describing the processes themselves.

There are many ways to implement a problem space. This section argues that goal reconstruction is simple to implement in any of the implementations of problem spaces. However, in order to make the argument easier to follow, an implementation of goal reconstruction will be described for a particular implementation of problem spaces. This implementation depends crucially on a Truth Maintenance System or TMS. Although this is a standard piece of technology in AI, it was developed fairly recently (deKleer, 1986; Doyle, 1979), so not all readers may be familiar with it. The first subsection describes a TMS-based implementation of problem spaces and how a TMS works. The second subsection presents a simple implementation of goal reconstruction. The third subsection argues that goal reconstruction is simple to add to other implementations of problem spaces.

Modeling State Changes with a TMS

The implementation of problem spaces presented here is the one pioneered by Strips (Fikes, Hart, & Nilsson, 1972). The TMS–based implementation of Strips problem spaces was developed more recently and is used by Prodigy (Carbonell et al., chap. 9 in this volume) and other problem solvers.

A state is represented by a set of literals in a first order logic. A literal is just a single predicate which may or may not be negated. Thus, *on (block56,block2)* and *not (clear(top(block2)))* are both literals. Literals that are used to represent states have no variables in them. They have only constants, such as *block56,* and functions of constants, such as *top(block2).* We use the Prolog convention of capitalizing variables. Constants, functions and predicates are written in lower case.

Perception (reading the state) is modeled by matching expressions against the set of literals that represents the current problem state. To find out what block is on *block2,* the expression on *(X,block2)* is compared to all the literals in the current state until one is found that matches (unifies) with it. Matching causes the variable x to be matched to a constant, say *block56,* thus answering the question of which block is on *block2.*

Action is represented by adding and deleting literals from the current state, thus creating a new state. A generic action is called an operator, and its generic effects are represented by a list of literals to be added to the current state (the add–list) and a list of literals to be deleted (the del–list).

In order to allow this economical description of actions to model complicated state changes, rules are used to maintain logical relationships that hold in all states. For instance, suppose the problem space uses a literal *indirectly-on (X,Y)* that means that X is directly on top of Y (i.e., *on (X,Y))* or X is on top of something that is indirectly-on Y. Two rules can be used to provide a formal recursive definition of *indirectly-on:*

1. *If on (X,Y) then indirectly-on (X,Y).*
2. *If there is a Z such that on (X,Z) and indirectly-on (Z,Y), then indirectly-on (X,Y).*

Given these rules, the operators need only mention their effects on the on literals. They do not have to mention *indirectly–on* literals in their add–lists and del–lists since the effects on those can be calculated with the two rules above. For the sake of discussion, let us distinguish *primitive* literals from *derived* literals. A primitive literal is one that is added directly to the problem space by an operator's execution because a generic version of it appeared in the operator's add–list. A derived literal is one that is added by the execution of a rule.

Although it is clear that the rules provide the knowledge that is required for omitting derived literals from add-list and del-lists, it is not as simple as it might seem to get the system to use this knowledge effectively. There are two basic methods. The simpler one, which was used by Strips, is to create a new empty state, add all the primitive literals specified by the operator's add-list and copy all the primitive literals from the old state that are not mentioned by the operator's del–list. Now the new state has all the primitive literals that it should have. The derived literals are added by repeatedly firing the rules until no new derived literals are inferred. Many of these derived literals will be equal to ones in the preceding state. For instance, if block A is on block B in the old state, and the action does not effect that, then *indirectly–on (A,B)* is true in both the old state and the new state. Thus, this method of modeling action amounts to *reconstructing* problem states.[3]

The other method of modeling, which is used by Prodigy (Carbonell et al., chap. 9 in this volume), achieves exactly the same result, but is more efficient because it substitutes cheap copying and removal operations for expensive re-derivation operations. The copying and removal operations use a TMS. The basic idea is to copy all the literals in the old state, including the derived ones, then remove all the literals that should be removed and add all the literals that should be added. The trick is to remove only the right literals. This happens in two stages. First, all the primitive literals that are explicitly mentioned in the operator's del–list are retracted. Second, the TMS retracts derived literals whose derivations depend on retracted primitive literals. In order to do this, the derivations of the literals have to be remembered.[4] If any of the primitive literals in the

[3]Although this description uses forward chaining, most problem solvers use backwards chaining. Instead of drawing all possible inferences as soon as the state is created, backward chaining makes inferences only when the problem solver poses a query, such as *indirectly-on (X,block2).* The important point is that in the reconstruction method, only the primitive literals are copied from the old state to the new state. Exactly when the derived literals are inferred does not matter.

[4]A derivation is a proof tree whose leaves are primitive literals. Thus, if rule 1 is used to derive C from A and B, and rule 2 is used to derive E from C and D, then the derivation for E is the tree (E(C(AB))D).

derivation are retracted, then the derived literal is also retracted. This retraction process is guaranteed to retract all and only the appropriate derived literals.

Next, the TMS–based system adds the literals from the operator's add-list and runs rules until quiescence. There is a trick that is used to speed this part of the process up. It is often the case that one of the derived literals that was retracted during the first phase is rederived during the second phase. Since the system has to remember derivations anyway, it is can save work by looking up the derivations that depended on this literal and calculate which ones can now be reinstated because the literal has been reasserted. Reinstating old retracted literals can be computationally cheaper than reinferring them. This trick is called *un–outing* (de Kleer, 1968; Doyle, 1979).

An example of a TMS–based state maintenance may be helpful. Suppose that whenever a block is supported directly or indirectly by the table, then X is stable. This is expressed by the following rule

If indirectly–on(X,table) then stable (X).

Suppose that in the old state, block56 is indirectly on the table and stable because it is on block2, which is directly on the table. That is, *on (block56,block2)* and *on(block2,table)* imply that *indirectly–on (block56,table)* by the rules listed earlier, and this implies *stable (block56).* Suppose that an operator applies, and moves block56 off block2 and onto the table. The del-list of the operator will retract *on(block56,block2)* and the TMS will thus retract *indirectly–on (block56,table)* and *stable (block56)* because their derivations depended on the retracted literal. Eventually, all the appropriate literals will be removed from the state. Now the TMS adds literals from the add-list, including *on (block56,table).* The rules are run, and they infer *indirectly–on (block56,table).* The TMS notices that this literal is identical to one in the old state. It uses the un–outing mechanism to reinstate *stable (block56)* immediately without referring to the inference rule. It knows that this is appropriate because the derivation of *stable (block56),* which has to be saved anyway for retraction to work correctly, indicates that *stable (block56)* depends only on *indirectly–on (block56,table).*

Goal Reconstruction in a Problem-Space Architecture

Although both the reconstruction and TMS methods of implementing state change are widely used in AI (see Charniak and McDermott's (1986) textbook, section 7.3), it is rarely recognized that they can also be used for maintaining the problem solver's goal structures. This section sketches a problem solver, similar to Amord (de Kleer, Doyle, Steele, & Sussman, 1977), that uses a TMS to maintain its goal structures.

A goal is usually defined to be a description (i.e., logical expression) of a state that is desired. A goal is satisfied if the current state matches the description. A pending goal is a logical expression that does not match the current state. Suppose problem solving starts with an initial state that contains a pending goal, which is represented by wrapping the pseudo-connective *pending–goal* around a logical expression. Thus, the goal of holding block37 in one's hand can be represented by *pending–goal (holding(block37))*.

When pending goals are represented this way, the rule mechanism mentioned above can be used to calculate what kinds of actions are appropriate for the given goals. This is most easily demonstrated with an example. The following rules indicate how to achieve a *holding(X)* goal given that the hand can only hold one block at a time.

> *If pending-goal (holding(X)) and not (holding(Y)),*
> *then executable (pick-up(X)).*
>
> *If pending-goal (holding(X)) and holding (Y) and not (X=Y),*
> *then pending-goal (not(holding(Y))).*
>
> *If pending-goal (not(holding(Z))) and holding (Z)*
> *then executable (put-down(Z)).*

These rules use pseudo-literals of the form executable (Op), where Op is an operator, in order to indicate that the specified operator is an appropriate action. In this case, if the initial state is the literals

> *holding (block6)*
> *pending-goal (holding(block37))*
> *not (block37=block6)*

then the second and third rules will add the following derived literals:

> *pending-goal (not(holding(block6)))*
> *executable (put-down(block6))*

This represents the process of deciding that putting down the block being held is a good idea given the current goal and the current situation. Much more complicated reasoning can also be represented.

When an action is finally taken, some or all of the *pending–goal* literals must be retracted because their predicates will now be true and only unsatisfied goals are represented with *pending–goal* literals. Both the reconstruction methods and the TMS method work just fine for updating *pending–goal* literals. Let us consider reconstruction first. In order to model the state change caused by executing the operation *put–down (block6)*, reconstruction adds the literal *not (holding(block6))* to the new state because that literal is mentioned in the operator's add-list. Then it copies over primitive literals from the old state that are not mentioned in the del-list. This adds *pending–goal (holding(block37))* and *not (block37=block6)* to the new state. Notice that the old derived literal, *pending-goal (not(holding(block6)))*, is not copied over. Now the reconstruction method runs the rules, which adds to the state the literal *executable (pick-up(block37))*.

This demonstrates how the reconstruction method works to maintain goal structures. Essentially, it starts over from the top level goal, which is the only one that is a primitive literal, and rederives as much of the goal structure as is still relevant. In this very simple illustration, no old goals were reconstructed. Usually, many old goals will be reconstructed.

The TMS method can be used in order to avoid some of the computation of reconstruction. In order to use it, the derivation of a goal must be stored with the goal. For instance, with *executable(put–down(block6))* the system associates the tree

```
executable (put-down(block6))
   pending-goal (not(holding(block6)))
      pending-goal (holding(block37))
      holding (block6)
      not (block37=block6)
   holding (block6)
```

which records its derivation via the rules listed earlier. In order to update the state, the TMS method first retracts *holding(block6)* because it is mentioned in the operator's del-list. Since this literal occurs in the derivations of both *pending–goal (not(holding(block6)))* and *executable (put–down(block6))*, those two literals are retracted as well. Next the system adds *not (holding(block6))* because that literal is mentioned by the operator's add-list. The rules run, and the literal *executable (pick-up(block37))* is derived. This demonstrates how the TMS method can maintain goal structures.

The TMS method is more memory intensive than the reconstruction method. It requires that the problem solver remember all the derived literals from the old state and moreover, it should remember the derivations of each literal as well. What would happen if memory failed? If a literal was completely forgotten and the literal was going to be retracted anyway, then it does not matter that it was forgotten. On the other hand, if the literal was not going to be retracted, then it must still be derivable from literals in the new state, so the rules will end up deriving it. So forgetting a literal does no harm. What if the literal is not forgotten, but its derivation is, or worse yet, only part of its derivation is forgotten? The problem solver must somehow detect this and treat the whole literal as if it were forgotten. If it can do that, then the literal and its derivation will be reconstructed if necessary and retracted (via forgetting) otherwise. Although the TMS-based method requires memory storage for the derivations, it is quite robust because it can easily reconstruct forgotten derivations.

From this perspective, the TMS method and the reconstruction methods are just two ends of a continuum. If all the derived literals can be recalled, then the faster TMS method is used. If none of the derived literals can be recalled, then the slower reconstruction method is used. If only some of the derived literals are recalled, then TMS-based retraction and reconstruction are used jointly to produce the appropriate goal structure. This is a seamless combination of reconstruction and recall.

Notice that the primitive literals cannot be forgotten. If they are, then the whole scheme falls apart, since there are no rules for deriving them from other literals.[5] However, problem spaces are usually designed so that the primitive literals model unforgettable information. For instance, the literals that describe the current state are usually chosen to correspond to perceptually available information (e.g., which block is my hand holding?), and are thus unforgettable. The top level goals are assumed to be either perceptually available (e.g., from instructions written on a card handed to the subject) or very familiar. Literals that reside unchanged in all states (e.g., *not (block37=block6))* correspond to common sense or well-learned facts.

The next step in the argument is to show that this mechanism for maintaining goals solves the three problems mentioned in the introduction: capacity limitations on goal storage, non–LIFO access to goals, and blending situated and planned action.

The capacity problem is that many computations seem to require more goals than the human short-term memory system has room to store. At first glance, it seems that this mechanism completely solves the capacity problem. As long as the top level goal is held in long term memory, any other goals that are forgotten can be reconstructed. However, if we take an extremely simple model of the short–term store, such as a buffer with seven cells, then it is possible for the goal reconstruction mechanism to fail. As goals are reconstructed, the rules generate literals that may be needed a few moments later by other rules. If more than 7 of these literals are generated, then some may be lost from memory before being used. It is important to note, however, that although the number of literals requiring storage in STM may be large, they do not have to be stored there for very long. It is well known that the number of chunks that can be recalled from STM varies inversely with the delay between storage and recall. The simple buffer model does not reflect this, although more complex buffer models could.

A standard model with the appropriate decay properties is based on spreading activation. In this model, gaining access to a goal requires that the goal exist in memory and that its activation level be above some threshold. In order to recall an old goal, activation can be spread up through the derivation trees starting with the literals that stand for perceptual chunks (which are presumably highly active as they are the current focus of visual attention) and the top–level goal. This corresponds to normal retrieval of an old goal. If the goal is inaccessible via spreading activation, then it can be reconstructed. Some elements of its derivation tree will be accessible (in the worst case, only the leaves can be retrieved).

[5]Actually, some literals can be both primitive and derived because they appear in both the add-list of operators and the conclusions of rules. These literals could be reconstructed if forgotten, at least in principle. They will continue to be ignored in order to simplify the discussion.

These trigger rules whose execution creates new literals that are copies of the forgotten ones. In most spreading activation theories (e.g., ACT*—Anderson, 1983), newly created elements are given high activation. Although activation decays rather rapidly at first, as long as the goal reconstruction process occurs rapidly and without interruption, it should be possible to reconstruct large numbers of goals. Thus, the initial impression that a TMS-based mechanism solves the goal capacity problem is actually correct, although there appear to be some subtle interactions with the operation of the underlying memory system.

The second problem mentioned in the introduction is that people sometimes execute goals that are not the most recently created pending goal. In LIFO architectures, this is not possible. The architecture sketched in this section is not necessarily a LIFO architecture. If the rules are run to quiescence, then they will find all pending goals whose preconditions are met and mark them as executable. The problem solver is free to choose any of them for execution.

The last problem mentioned in the introduction is to find a way to blend situated action and planned action. In a single-agent world, the above mechanism suffices. If all goals are forgotten during a state change, then the agent can be said to have no state, so it is a situated-action agent. The calculations that are performed by goal reconstruction would have to be performed by any agent possessing the same knowledge, and the above mechanism makes it seem that the intermediate results of these calculations must be stored as literals. However, one can replace the rules by gates in a combinatorial logic and the literals by wires connecting gates. Seen this way, the goal reconstruction calculation requires no more state than Agre and Chapman's (1987) Pengi or Brook's (chap. 8 in this volume) subsumption architecture. So in a world where the only source of state change is the agent itself, the TMS–based goal maintenance mechanism seamlessly blends situated and planned action.

If the world has multiple sources of state changes, then the agent must supplement the add-list and del-list with perceptual operations. These have the same effects as the lists do, in that they cause addition and retraction of literals. The TMS propagates these through the goal structure in the usual way. Thus if another agent helps our agent by satisfying one of our agent's pending goals, then perception will add a literal to the state, and the TMS will ultimately retract that goal. Similarly, if a hostile agent undoes a goal that our agent previously accomplished, then the un-outing mechanism of the TMS will quickly reinstate the goal.[6] Thus, the agent will behave adaptively in a changing world where not all of the changes are under its control. The TMS-based mechanism is adequate for blending situated and planned action even in a multiple agent world.

[6]This TMS-based goal maintenance mechanism does not model the process of deciding which executable action to execute. This is called *action arbitration* by Agre and Chapman (1987) and Brooks (chap. 8 in this volume). Their systems seem to use some ad hoc priority-based system to do action arbitration. Presumably, such a system could be used here as well, or a more complex system, like the goal preferences of Prodigy (Carbonell et al., chap. 9 in this volume), could be used instead.

In addition to solving the three problems mentioned in the introduction, the TMS-based goal maintenance system bears striking similarities to the overall human memory system. As noted earlier, when people are distracted from a task by a long telephone call, they have the ability to reconstruct the goals and other internal state that they have forgotten. On the other hand, if they are not distracted, then they do not require similarly extended periods of time for reconstructing their state after each action. Moreover, there seems to be no sharp boundary between human retrieval and reconstruction. The seamless combination of TMS and reconstruction methods also has the same lack of a sharp boundary. It is well known that human recall is facilitated by making the perceptual environment at recall similar to the perceptual environment at storage. This is consistent with the TMS/reconstruction combination, where the accuracy and availability of the whole state depends strongly on the accuracy and availability of the primitive literals. In short, the combined TMS-reconstruction method of state updating is qualitatively similar to human memory, at least as far as problem solving is concerned.

Goal Reconstruction in Precondition-Based Problem Solvers

The simplicity of the TMS-based goal maintenance system is due to its use of rules for reasoning about goals. Although this is elegant and allows certain issues to be presented clearly, rule–based representations of planning knowledge can be awkward and redundant, especially for conjunctive goals. A more widely used technique represents that knowledge in the operators themselves as a set of *preconditions* on the operator. This is the representation used by Strips (Fikes, Hart, & Nilsson, 1972) and its many descendants. The goal reconstruction capability of the Amord–style problem solver can also be implemented in a Strips-style problem solver. The next few paragraphs demonstrate this.

An example will help in comparing Amord–style and Strips–style problem solvers. In a Strips–style problem solver, the *pick–up* operator could be represented as:

Name: pick–up
Arguments: X
Preconditions: not(holding(Y))
Add–list: holding(X)
Del–list: on(X,Z)

This representation replaces the rules listed earlier for the Amord-style reasoning about goals. For instance, one of the rules mentioned earlier says

If pending–goal (holding(X)) and holding(Y) and not(X=Y),
then pending–goal (not(holding(Y))).

Let us use the Strips representation to do the example mentioned earlier, where the goal is *holding(block37)*. The system searches for an operator whose add–list matches the goal. In this case, *pick–up* is found. Next, the preconditions are checked. In this case, the precondition is false, because *holding(block6)* is true. Whenever a precondition is not satisfied, the system makes it into a subgoal. Thus, the system makes *not(holding(block6))* a goal. Clearly, this Strips–style reasoning has achieved the same effect as the inference rule from the Amord-style problem solver. The knowledge representation, however, is more parsimonious.

Most Strips-style problem solvers do not use a TMS for maintaining their goal structures. In fact, we do not know of any that use a TMS. Instead, they use a *goal tree*, or more frequently, just a part of the tree arranged in a *goal stack*. Although less elegant than the TMS-based method, goal maintenance with a goal tree has all the same properties. The next few paragraphs are a point–by–point comparison of the goal tree and TMS methods of maintaining goals.

In TMS-based maintenance of goals, the derivation of each goal must be stored so that retraction and un-outing can function correctly. Instead of a derivation, a tree-based goal maintainer uses the goal tree itself. Instead of a TMS data structure indicating that *pending–goal (not(holding(block6)))* was derived from *pending–goal (holding(block37))* and other literals, the goal tree has a data structure indicating that *pending–goal (not(holding(block6)))* is a subgoal of *pending–goal (holding(block37))* caused by an unsatisfied precondition.

In TMS-based maintenance of goals, executing an operator first causes all satisfied goals to be retracted. In tree-based maintenance, the exact process of finding satisfied goals seems to vary from one problem solver to the next. However, the gist of the method is to check goals in the tree and see if the goal's literal is now in the current state. If it is, then that goal is satisfied and all its subgoals are now irrelevant. These goals are marked appropriately or removed from the tree. If a goal stack is used instead of a goal tree, this phase can be accomplished by popping the stack.

After the satisfied goals have been dealt with, the TMS method infers new goals using its inference rules, whereas the tree–based method infers new goals using the operators' preconditions.

One of the advantages of the TMS–based method of goal maintenance for modeling humans is the seamless integration of reconstruction and recall. The same advantage can be obtained with a goal tree. If parts of the goal tree are forgotten, they can be reconstructed by starting at an ancestor goal and using the usual precondition-based subgoal creation method. If subgoals are created that are equal to ones that have not been forgotten, then the new tree can be attached at this point to the tree rooted at the recalled subgoal. The goal indexing mechanism used in GPS (Ernst & Newell, 1969) and most of its successors will cause this reattachment (which is equivalent to un-outing) to happen automatically. Thus, in most cases no new mechanisms need to be added to the system in order to achieve a qualitative similarity to human behavior.

This section has shown that goal reconstruction is a capability that can be easily added to a problem–solving architecture. Moreover, the goal maintenance mechanism shifts seamlessly from recall to reconstruction of goals, which makes its performance qualitatively similar to human behavior.

Goal Reconstruction in Procedure-Following Systems

In our vocabulary, a goal serves two purposes. It is both a description of a desired state of the world, and it is a part of the control structure of the problem solver. For some tasks, the description of desired states mention features that cannot be detected by unaided perception. If a person has already washed a piece of medical glassware 6 times in tap water, then washing it 6 times in distilled water will achieve a state of cleanliness that is not distinguishably different from its current state. The only simple description of the goal to be achieved is just the procedure for achieving it: wash the glassware 6 times in distilled water. A large number of tasks have this property. After a house is built or a tax form is filled out, the results look to the visible eye like a house or a tax form. But the quality of the house or the tax accounting can vary widely depending on which procedures were followed in achieving it. Properly cured concrete looks exactly the same as an improperly cured concrete. The only way to know if the properly–cured–concrete goal has been met is destructive testing (which partly undoes the goal of having properly cured concrete) or checking that the proper curing procedure has been followed. Many goals in human culture have the property that they are partly specified by the visible state to be achieved and partly specified by the procedures that should be followed in achieving them.

Although most problem–solving architectures can only accept goals that are specified by descriptions of the desired state, Sierra is one that is specifically designed to follow procedures (VanLehn, 1987, 1990). Recently, it has been augmented with the ability to accept goals specified as desired states. The resulting architecture, called Teton, is documented in an appendix to this chapter.

When goals are specified by procedures, reconstruction of goals becomes more complicated. Teton can handle some cases (but not all) with a fairly simple mechanism. Teton uses a Strips–like operator representation for procedural knowledge. In addition to the usual slots for preconditions and so forth, operators can have a *shortcut condition.* This condition is checked just before executing an operator. If it is true, then the operator is not executed but its goal is marked "satisfied" anyway. For example, in order to reconstruct the goals of the following partially completed multiplication problem,

$$\begin{array}{r} 336 \\ \times\ 208 \\ \hline 2682 \\ 7200 \end{array}$$

Teton would run the multiplication procedure which causes an operator, call it *Single–digit–multiply,* to be instantiated for each of the three digits in the multiplier, 208. Suppose the operator has a shortcut condition that is true if the partial product row to be filled already has some digits in it and there is something written underneath that row (i.e., another partial product row or a bar). In the case of the units digit instantiation of *Single–digit–multiply,* the shortcut condition is true, so the operation is marked completed. However, the shortcut conditions are false in the case of the tens digit, so execution resumes with that operation. Thus, Teton reconstructs goals then judiciously takes "shortcuts" instead of executing some of them.

As a quick check on the plausibility of this type of processing, we took a protocol from a subject who was asked to complete the partially solved problem shown above. She said:

> Alright. Since there are two columns done [referring to the partial product rows], I know that the first digit on the right hand side of the bottom number has been multiplied. Um. I would start the, um, since the second column is a zero, somebody has filled in the zero. I would now go to the third digit on the bottom column and do all the multiplication involved there. Two times six is twelve, two times three is six plus one is seven, two times three is six and then I would do the addition starting from the right hand side, and get the answer.

The first sentence corresponds to taking the shortcut on the *Single–digit–multiply* of the units digits of the multiplier. The second sentence corresponds to the execution of the *Single–digit–multiply* of the tens digit. The rest of the protocol corresponds to execution of *Single–digit–multiply* for the hundreds digit. This protocol corresponds quite well with the type of goal reconstruction used by Teton.

Shortcut conditions are task-specific knowledge about how to reconstruct specific goal trees. Sometimes people may have to learn shortcut conditions, and sometimes they may be able to deduce them from general principles in the midst of reconstructing a goal.

There is another type of task–specific knowledge about goal reconstruction that people sometimes use. If one can anticipate forgetting some goals, say because the phone is ringing and one intends to answer it, then one can take steps now that will make reconstruction much easier to do later. For instance, if one is interrupted by a ringing phone in the middle of adding up a long column of figures, one can write the subtotal down and mark the last number added in. This will enable reconstruction later. Teton does not handle this sort of knowledge. It would be a fascinating behavior to simulate, because the agent must have a crude model of forgetting in order to plan ways to prevent forgetting from happening. It also must be able to tell what aspects of its state are worth saving, so it must understand its capabilities for goal reconstruction.

As usual in cognitive modeling, we can model the most common cases but the other cases are orders of magnitude harder to model. Goal reconstruction is easily modeled when goals are descriptions of visible aspects of the state, as in the case of the Amord–style problem solvers and the Strips–style problem solvers. When goals are partially procedures, then the shortcut conditions of Teton can handle some of the cases. However, the remaining cases of goal reconstruction present tricky problems that are likely to resist modeling for some time.

ARITHMETIC LEARNING: AN APPLICATION REQUIRING PERCEPTION

The preceding account of goal reconstruction ignored perceptual processes and assumed that their output was available in the form of literals in the current problem state. Part of the novelty of situated action is the claim that perceptual processing handles most of the load in guiding activity. In this chapter we discuss how to integrate perception and problem solving in such a way that goal reconstruction retains all the good properties that it had when problem solving was based on problem spaces.

This investigation grew out of a study of how people learn arithmetic, algebra equation solving and other written procedures. There is fairly good evidence that students pay close attention to the visual syntax of the written expressions and may even induce visual features into their procedures that the teacher did not intend them to learn (VanLehn, 1986, 1990). This reliance of visual features is the key to explaining many otherwise mysterious phenomena, as the following example illustrates. When students are introduced to borrowing, teachers usually use the simplest subtraction problems they can—ones with just two columns. Here is a borrowing problem that has been solved in the manner taught in many American textbooks:

$$\begin{array}{rrr} & 2 & 14 \\ & \cancel{3} & \cancel{4} \\ - & 1 & 8 \\ \hline & 1 & 6 \end{array}$$

Some students notice that the decrement action takes place in the leftmost column of the problem, and induce that all such actions should take place in the leftmost column. This leads them to make errors like the following one:

$$\begin{array}{rrrr} & 1 & & \\ & \cancel{2} & & 18 \\ & \cancel{3} & 2 & \cancel{8} \\ - & & 1 & 9 \\ \hline & 2 & 1 & 9 \end{array}$$

Early versions of Sierra simply postulated a problem space that includes leftmost and other relations that students induce. It would leave out relations that did not seem to play any role in their learning. Thus, the initial state would contain the literal *leftmost (column3)* but it would not contain the literals *rightmost (column1)* or *hundreds (column3)*. Although this allowed Sierra to explain the systematic errors of thousands of students, it also pushed the mystery of learning back one more level. Sierra explained how procedures are learned, but what explains how the problem spaces are learned? This comment is not meant to denigrate the accomplishment—all models of learning bottom out on some kind of assumptions about prior knowledge, and most models of procedure acquisition bottom out on the problem space, just as early versions of Sierra did.

In the case of mathematics, it is particularly important to explain problem spaces rather than assume them. The problem space embeds knowledge of mathematical notation, which is something that students learn (and mislearn!) in school. Whereas someone might naturally think of a column of 3 wooden blocks as something that is important enough to see and record in the problem space as a composite structure, such as *stack(block37, block6, block13)*, the habit of seeing a subtraction problem as columns instead of rows is something that has to be learned in school.

As a first step in determining how people acquire mathematical problem spaces, and knowledge of notational syntax in particular, it is wise to determine what the representation of that knowledge is like. This makes it easier to formulate learning models for notational knowledge.

Pursuit of these goals led us to the problem of devising a representation of notational knowledge that could be nicely integrated with mathematical problem solving. The first part of this section discuss some constraints on the representation of notational/perceptual knowledge. These were uncovered by trying simple approaches and discovering that they were inadequate. The second part of the section presents a system that seems to meet all the constraints. Moreover, its structure sheds some light on the distinction between situated and planned activity.

The Need for Global Parsing

The first attempt at representing notational/perceptual knowledge was to assume that task-specific terms in the problem space were defined by task-general terms using standard first-order logic. Thus, *column (X)* is defined to be a sequence of three vertically aligned cells, and *cell (Y)* is defined to be a digit, a blank or a digit that has been scratched out and written over. The *column (X)* definition might be represented formally as:

column (X) ::= *part-of(X,C1) & part-of(X,C2) & part-of (X,C3) &*
cell (C1) & cell (C2) & cell (C3) &
sequence (X) & first (X, C1) & last (X,C3) & middle (X, C2) &

ordered (X,C1,C2) & ordered (X,C2,C3) & ordered (X,C1,C3) &
adjacent (C1,C2) & adjacent (C2,C3)

Learning mathematical notation is assumed to consist of learning definitions like this one. There are a variety of machine learning algorithms sufficient for learning such concepts from examples and a given set of primitive concepts (e.g., VanLehn, 1987; Vere, 1975; Winston, 1975). In this case, the given concepts are perceptual primitives, such as *adjacent (X,Y)*. This is not the large loophole that one might imagine because the set of perceptual primitives needed for mathematical symbol manipulation is surprisingly small. For instance, one vocabulary sufficient for arithmetic and algebra required only ten predicates (see p. 183, VanLehn, 1983). A much more complex vocabulary would be needed for, say, high school geometry or mechanical drafting.

It might seem that the major difficulty in this approach to explaining the acquisition of mathematical problem spaces would be determining how people acquire concepts such as *column*. In fact, this approach failed utterly before even getting to that stage. Even when definitions are constructed by hand, it proved impossible to find definitions that would perform like people do. For Sierra, the visual world was represented as a Cartesian plane with characters centered at particular x-y coordinates. One problem was to get a definition of "algebraic formula" that is true of "2 + 3" when it stands alone in the plane, but to be false of "2 + 3" when it is embedded in "2 + 3x." Another problem is that adjacent (3,x) should be true of (a) below and false of (b) despite the fact that the two symbols are closer in (b) than in (a):

```
a.  3 x = y
b.  3 y = x
      x = y/3
```

The problem here is that an interpretation of a subset of some mathematical symbols is acceptable only if it participates in a global interpretation which includes all the symbols. This is analogous to many English words, such as "run," which can be interpreted either as a noun or a verb depending on the global interpretation of the sentence it is a part of. Compare "I'm not going to run today" with "I had a good run today." In the analysis of both English and mathematical syntax, better techniques are based on context–free grammars or something like context–free grammars.

Grammatical Definitions of Task-Specific Problem Representations

In order to use context-free grammars as a representation for knowledge of mathematical notation, a few augmentations to the standard formalisms were

TABLE 6.1
A Simplified Grammar for Arithmetic Notation

1. Problem	←	Sign ColumnS	; horizontal
2. Sign	←	+	
3. Sign	←	−	
4. Sign	←	×	
5. ColumnS	←	Column ColumnS	; horizontal
6. ColumnS	←	Column	
7. Column	←	Cell Cell Cell	; vertical
8. Cell	←	Digit	
9. Cell	←	Blank	
10. Digit	←	1	
11. Digit	←	2	
. . .			

needed. For instance, because mathematical notation is two-dimensional, rules need to indicate whether their constituents are arranged horizontally, vertically or diagonally. Table 6.1 shows a simplified grammar for arithmetic problems.

This grammatical formalism accomplishes what the first-order logical definitions of terms could not. It can properly parse arithmetic and algebraic expressionse. Unfortunately, a very nasty problem was encountered when Sierra's problem space machinery was replaced with a parser for this formalism.

The problem occurs when states change. For Sierra, state changes are always due to writing a new symbol on the visual page. When this happens, there is usually not much change in the parse tree.[7] Filling a column's answer in a subtraction problem only affects one small part of the parse tree—that which concerns the particular blank cell that is filled by the new symbol. Sometimes, however, writing a single symbol has effects on other parts of the tree. Writing

[7]A parse tree is a record of the derivation or parsing of a particular sentence, or in this case, of a particular mathematical expression. A parse tree for the vertical form of 2 + 1 when parsed by the grammar of table 6.1 would be:

```
Problem—Derived via rule 1 from:
  Sign—Derived via rule 2 from:
    + (perceived)
  ColumnS—Derived via rule 6 from:
    Column—Derived via rule 7 from:
      Cell—Derived via rule 8 from:
        Digit—Derived via rule 11 from:
          2 (perceived)
      Cell—Derived via rule 8 from:
        Digit—Derived via rule 10 from:
          1 (perceived)
      Cell—Derived via rule 9 from:
        Blank (perceived)
```

where indenting represents the hierarchical relationships in the tree.

one symbol changes "2 + 3" into "2 + 3x," which changes the interpretation of the 3. In order to allow for arbitrary changes in the state, Sierra ignores the old state's parse tree and constructs a new one for the current state. This has the unfortunate side-effect of making obsolete most of the goals held in Sierra's working memory because most goals have arguments that mention nodes in the parse tree. When the visual scene is parsed anew, a whole new parse tree is produced, but the goals continue to mention nodes from the old parse tree. By parsing the current state, Sierra makes obsolete all the goals with objects as arguments.

Several years ago, this seemed like a nasty technical problem with no important theoretical implication. It was circumvented with some subtraction-specific hacks and banished to appendix 8 of the first author's dissertation (VanLehn, 1983).

Annotated Grammars: Another Version of Situated Action

In the intervening years, the situated action paradigm has begun exploring the idea that people rarely plan by building up stacks or trees of pending goals. Instead, they parse the situation so as to "see" possibilities for actions. Thus, goals are not held in memory, but perceived in the situation.

In order to better understand the implications of the situated action view, we implemented an architecture, called *Rocky*. Instead of a procedure, Rocky has a grammar that is just like the one used by Sierra to represent knowledge of mathematical notation except that it has a few extra annotations. For instance, the rule for parsing a column, rule 7 in table 3-1, is annotated to indicate the numerical relationship among its the cells in the column:

7. *Sub-column* $\rightarrow$ *Digit*$_1$ *Digit*$_2$ *Digit*$_3$; *vertical*
   where: *Digit*$_3$ = *Digit*$_1$ − *Digit*$_2$

We call this kind of knowledge representation an *annotated grammar*. With proper interpretation, it seems quite likely that an annotated grammar can generate actions and solve problems just as well as a procedure.

By getting rid of goals, the annotated grammars approach solves the problem of goals becoming obsolete. Each time the state changes, a new parse tree is constructed and nodes that are capable of having actions taken on them are marked as executable. The resulting parse tree quite literally wears the possibilities for action on its sleeve. Thus, an annotated grammar not only parses the visual plane, it also does all the reasoning that would normally be done by the rules mentioned earlier that compute with literals named *goal* and *executable*.

Annotated grammars seem to implement what Suchman (1987) had in mind when she said, "We generally do not anticipate alternative courses of action, or their consequences, until *some* course of action is already under way. It is frequently only on acting in a present situation that its possibilities become clear." (Suchman, 1987, p. 52, original emphasis)

Unfortunately, the annotated grammars approach ran into grave difficulties when we tried to implement some of the less visually oriented mathematical procedures. For instance, consider a common procedure for solving multiplication problems, which involves skipping zeros in the multiplier, as in the following problem:

$$\begin{array}{r}
2345 \\
\times\ \underline{1204} \\
9380 \\
469000 \\
\underline{+2345000} \\
2823380
\end{array}$$

There are four multiplier digits, but only three partial products. In order to properly pair off the multiplier digits and the partial products, an annotated grammar must encode what amounts to a right–to–left traversal of the multiplier digits. Similarly, it is difficult to differentiate the zeros that are inserted in order to vertically align the partial procedures from the zeros produced by multiplications (see the second partial product above). Counting or some other kind of iteration is needed in order to determine these mapping from the visual plane. This cannot be done in the representation for grammars used by Rocky. Although the representation could perhaps be augmented, this would go against the situated action paradigm, which tries to obtain action without explicit execution of procedural knowledge, such as an iteration across a string of digits.

The underlying problem is that the only way to properly understand some problem states is to know how they were derived, and this historical information is sometimes not present in the perceptual information. In the task of washing medical glassware, one cannot tell by looking at a piece of glassware how many times it has been washed. An annotated grammar cannot perform this task.

In retrospect, it appears that Rocky's version of situated action is too extreme. It tries to keep *no* historical information about the problem solving and instead work only with what it can infer from the current situation. This is a rather implausible hypothesis, for surely a person in the middle of a problem would recall and use information about immediately preceding actions and decisions if such historical information were useful. As argued earlier, most architectures based on problem spaces have this property (or could have it given a few simple augmentations). They recall goals when they can and reconstruct them otherwise. Their reconstruction proceeds from primitive literals, which often represent outputs from perceptual processing. Somehow this useful and psychologically plausible property has been lost in the attempt to deepen the model of perception so as to allow for task-specific knowledge about mathematical notation.

A TMS–Based Parser

Let us temporarily abandon the parsimony of unifying procedures and grammars and return to the old assumption that procedures and grammars are two distinct bodies of knowledge. This means that there are two types of internal state, a parse tree and a goal structure. The parse tree nodes correspond to the objects that would exist in the current problem state if a problem space approach were being used.

This means we must solve the updating problem wherein all goals that refer to parse nodes become obsolete with each state change because the parse trees for different states share no nodes. What we would like is an updating technique that will allow parse trees from consecutive states to share as many nodes as possible. Only parse nodes for parts of the visual plane that are "really new" would be built. However, the definition of "really new" depends on the task.

A solution that we think will work (it has only been partially implemented) is based on the same TMS-reconstruction method that was used successfully with regular problem spaces. The key idea is to note that parsing a visual scene is a special kind of inference, where grammar rules correspond to inference rules and parse nodes correspond to literals. A TMS is used to retract only those literals (parse nodes) that are changed, directly or indirectly, by the writing of a new symbol on the visual plane. In order to make this idea work for mathematical notation, however, we must be very careful about the representation of blank space in the visual plane.

As a running example, consider the change from "2 + 3" to "2 + 3x." The status of the 3 should be changed, but the parse node for the whole formula should stay the same. Suppose that the grammar is just

sum → *term* + *term*	*; horizontal*
term → *term term*	*; horizontal*
term → *2*	
term → *3*	
term → *x*	

Let parse nodes be represented by unary ground literals. The predicate is the category of the constituent and the argument is a region. For concreteness, let a region be represented by four numbers in square brackets, corresponding to the left, top, right and bottom boundaries of the region. Thus, term ([5,23,25,13]) represents a term occupying a certain region. With these definitions, the first grammar rule becomes the following inference rule.

If there are three regions, R1, R2 and R3 such that
term(R1) & plus(R2) & term(R3) &
right-boundary(R1) = left-boundary(R2) &
right-boundary(R2) = left-boundary(R3) &
region C is the union of regions R1, R2 and R3,
then sum(C).

The visual plane is represented by primitive literals and the grammar (inference) rules create derived literals.

In order to make the TMS–reconstruction method work, literals that mean the same thing, relative to the task, should be syntactically equal. Recall that reconstruction continues to run inference rules until no new literals are produced. "New" is defined relative to syntactic equality. If a literal is produced that is equal to an existing literal, then we say that a new derivation was found for an old literal; a new literal was not produced. Equality of literals depends crucially upon the definition of regions. Let us define the top boundary of a region to be halfway between that region and the next region in the positive y–direction. If there is no such region, then the boundary is set at infinity, which is represented by "*." Define the bottom, left and right boundaries similarly. Thus, "+" in the expression "2 + 3" would be represented by the literal *plus ([35,*,45,*])* because the top and bottom boundaries are at infinity.

With this definition, the literal *sum([*,*,*,*])* represents either "2 + 3" or "2 + 3x" written alone on a page. This makes the two terms syntactically equal, which is just what we want. A goal whose argument refers to "2 + 3" will not be made obsolete by stage change. Both before and after the state change, the goal's argument will be *sum ([*,*,*,*])*.

Let us see how the TMS handles the state change from "2 + 3" to "2 + 3x." The parse tree for "2 + 3" consists of the following literals:

```
1. sum ([*,*,*,*])
2.   term ([*,*,35,*])
3.     two ([*,*,35,*])
4.   plus ([35,*,45,*])
5.   term ([45,*,*,*])
6.     three ([45,*,*,*])
```

When the writing operator puts an "x" in region [55,*,*,*], it must retract primitive literals whose regions have been overlaid and assert new literals with smaller regions. In this case, the literal on line 6 above must be retracted and a new literal *three ([45,*,55,*])* is asserted. Retracting the literal on line 6 causes the TMS to retract the literals on lines 5 and 1, since their derivation depends on the literal of line 6. However, the addition of the new literals for "x" and "3" causes reconstruction, which leads ultimately to a new parse tree, which is:

```
1. sum ([*,*,*,*])
2.   term ([*,*,35,*])
3.     two ([*,*,35,*])
4.   plus ([35,*,45,*])
5.   term ([45,*,*,*])
```

7. *term ([45,*,55,*])*
8. *three ([45,*,55,*])*
9. *term ([55,*,*,*])*
10. *x([55,*,*,*])*

The literal on line 5 has been reconstructed. Although it has a different derivation now, it occupies the same region as before, so it is equal to the old version. The un-outing mechanism of the TMS will detect this and cause the literal on line 1 to be reinstated.[8]

It appears that the updating problem has at least been solved. By using a TMS–based method, only parse nodes that are truly different are changed. This means that only goals whose arguments have really changed must be reconstructed.

Moreover, by using the TMS-based method of updating, we obtain the same seamless blend of recall and reconstruction that characterizes human recall behavior. If parts of the parse tree are forgotten, then the TMS-based updating method will simply reconstruct them without even "noticing" that they were forgotten.

Summary: When Is Reasoning Really Perceptual?

In this section, we have descended into the ugly details of mathematical notation in order to find out what would happen if the problem space approximation was dispensed with and something more like real perception was modeled. It turned out to be much more difficult than it first appeared. There were two interacting sources of difficulty. The first was the fact that mathematical notation cannot be defined locally, but only by finding the most globally coherent parse of the visual plane.

The second difficulty occurs when updating the state after an operation is executed. This problem, which includes the frame problem of AI, can be solved in the problem space framework using Strips operators and a TMS. However, it is more difficult when perception is modeled. The global coherence of a perceptual parse means that the individual parts of the parse depend on each other in subtle ways. A change to one small piece of the visual plane can ripple through the parse and change large amounts of it. After a noble but ill-fated attempt at ducking the problem (the annotated grammars approach), a method was found for representing mathematical notation so that the propagation of changes died

[8]The old parse tree, which treats "2 + 3" as a sum, is still available, but now it has *sum([*,*,*,55])* as its root instead of *sum([*,*,*,*])*. This literal does not participate in a parse that covers all the symbols. In order to avoid generating it, the inference mechanism should only produce literals that participate in the derivation of a literal whose argument is [*,*,*,*]. This restriction would be simple to implement in a backwards chaining control structure; a forwards chainer would require a filter.

out quickly. This allowed perceptual parsing to be updated by roughly the same TMS-based method that successfully updates state changes when problem spaces are used.

Stepping back still further, one sees that the two computations, one supposedly procedural and the other supposedly perceptual, are nearly identical. The perceptual calculation updates a "state," which is a set of existing objects and their relationships to each others. The procedural calculation updates a "goal structure," which is a set of desired things and their relationships to each other. In order to obtain a reasonable solution to the frame problem, the same TMS-based method is used to update both the state and the goal structure. This method also yields robustness to forgetting, even the kind of massive forgetting caused by answering long telephone calls.

From a computational point of view, nearly the only way to tell that one calculation is perceptual and the other is procedural is to read the English names of the predicates, which is something that only a human observer can do. The situated action theorist would probably call the whole calculation perceptual. Traditional problem-solving theorists would call the whole calculation problem solving. Planning theorists would call it planning or perhaps reactive planning. As far as we can see, what you call it does not change what it is and does. As with many of the great binary distinctions in AI (e.g., procedural vs. declarative, logic vs. knowledge engineering), the distinction between situated action and planned action may turn out to be too ill-defined to be useful.

DISCUSSION: MULTIPLE LEVELS AND EXTRA CAPABILITIES

Two claims are made in this chapter. One claim is that goal reconstruction solves at least three problems: allowing intelligent problem solving within a limited capacity store for goals, providing non-LIFO access to goals, and creating a seamless blend of situated and planned action. The other claim is that most current problem-solving architectures already have the capability to do simple goal reconstruction or could easily add that capability with a few changes. These are primarily computational claims, although we have indicated at several points the similarities of goal reconstruction and human cognition, and particularly the way a TMS–based goal maintenance system mimics the way human memory blends recall and reconstruction. Most of this section discusses the psychological status of goal reconstruction, but first we present one further claim.

Goal reconstruction is useful in building AI systems. This claim is based on our experience with our newest problem-solving architecture, Cascade. Cascade is a simplified version of Teton. The major simplification is that it can only represent monotonic state changes (i.e., all the operators have empty del-lists). While constructing an expert system in Cascade for solving physics problems, we discovered that goal reconstruction was quite useful during debugging. The usual cycle during debugging is to try a computation, detect a mistake, find the buggy piece of knowledge, correct it, and redo the computation. Goal reconstruction makes redoing the computation much faster because the problem solver can begin more–or–less from where it left off. We are currently adding a learning engine to Cascade that will act roughly like a programmer would in debugging the knowledge base. We suspect that goal reconstruction will aid the learning engine just as it aided the programmer. If our experience generalizes, then there are some unexpected practical benefits to adding the little bit of extra code to problem-solving architecture that allows it to reconstruct goals.

It is time to address the psychological status of goal reconstruction. Is it a part of the real human cognitive architecture? Newell (1990) and Pylyshyn (1984) define the cognitive architecture to be those parts of cognition that are innate, subject-universal (i.e., common to all subjects) and cognitively impenetrable. We think that goal reconstruction is subject-universal, but neither innate nor cognitively impenetrable. For instance, instructions to the subject can probably cause them to modify the way they do goal reconstruction, which would imply that goal reconstruction is cognitively penetrable and hence not a feature of the true cognitive architecture, according to Pylyshyn (1984). Thus, computational architectures such as Teton that have goal reconstruction built into them are not good models of the cognitive architecture. A better computational model would represent goal reconstruction as knowledge—a program in the model's library.

However, there are problems with modeling goal reconstruction as a cognitive procedure that has the same form as a procedure for arithmetic or physics. A procedure for goal reconstruction would have to take two inputs, the perceptual situation and the task's procedure (e.g., multiplication), and produce a goal structure as output. This procedure would not only have to be a meta-level procedure, because it reads other procedures and produces goal structures, but it would have to duplicate most of the functionality of the architecture's interpreter. The goal reconstruction procedure would essentially be a copy of the interpreter with a few extra lines of code added. This position is not only unparsimonious, but nearly self-contradictory. How could a person learn a procedure that is a copy of their architecture when the architecture is not open to introspection? In short, there are grave technological and developmental problems with the position that goal reconstruction should be modeled as a cognitive procedure rather than a feature of the architecture.

The fact is that cognitive modelers are not free to set the architecture/program boundary anywhere they want. Even the Soar group, with its emphasis on aligning Soar's architecture with the human cognitive architecture, finds it convenient to provide a selection problem space as part of the bare, "innate" Soar. In its format, the selection problem space is identical to problem spaces for acquired capabilities, such as a solution procedure for a puzzle, but the selection problem space is considered to be a model of a capability that is innate, subject–universal and cognitively impenetrable.

Rather than label Soar, Teton and other architectures as failures, let us reconsider the research object proposed by Newell and Pylyshyn, which is to develop an computational architecture that models all and only the human capabilities that are innate, subject universal and cognitively impenetrable.

First, not everyone cares about innateness, universality and penetrability. More typically, learning theorists begin by defining a set of tasks that they intend to explain. For instance, Anderson (1983) chose memory tasks (mostly), Berwick (1985) chose English syntactic analysis tasks and we chose problem solving tasks. In order to explain the observed learning behaviors, the theories assume specific *prior* cognitive capabilities. These are processes and structures that are assumed to exist at the time the tasks' acquisition begins. For instance, one of Anderson's theory's prior capabilities is a semantic network with specific functions for spreading activation and strengthening connections. Berwick's theory assumes a fixed parser as one of its prior capabilities. We assume that goal reconstruction is a prior capability. Although all these theorists seem to believe in the subject–universality of their prior capabilities, none have addressed cognitive penetrability and their claims about innateness are made tentatively if at all. This is quite reasonable. The objective of their investigations is an explanation of human behavior in the chosen task domains. Assuming that a prior capability is innate or impenetrable adds little to the explanatory adequacy of their theories. Logically, an explanation for some acquisitional behavior does not have to involve ascriptions of innateness and penetrability, but only assumptions about what capabilities existed prior to the observation period.

If the cognitive theorist expresses the learning theory as a computer model, it often takes the form of an architecture and some programs. Some of the theory's prior capabilities are expressed as programs and some are features of the architecture. There is no logical reasoning why the prior capabilities must be part of the architecture alone. Indeed, it is hard enough to formulate a detailed computational model without being saddled with this superfluous restriction. What matters is developing a scientifically adequate explanation of the phenomena, and that does not entail any particular alignment of prior capabilities with distinctions inherent in the modeling technology.

Cognitive modeling has produced relatively isolated computer-based models, which, as Newell (1973) points out, leaves psychology with no unified theory of cognition. It seems to us that there are three approaches to a unified theory:

1. Reduce all the models to the lowest common denominator. A model of the lowest-level cognitive processes is selected (or developed) and models of higher level processes are (re-)implemented on top of them. ACT* is an example of such a unified theory of cognition. As Anderson discovered, actually implementing a model of a higher level process on top of a model of lower-level processes is

technologically difficult, to put it mildly. Even if it could be done, the model would produce unusably complex "explanations" of high level human behavior. In order to achieve integration, this approach sacrifices the explanatory adequacy of the higher level models.

2. Develop models at different levels, and indicate explicitly how they relate to each other. This approach seems to characterize Anderson's recent computational models (Anderson, 1989). Grapes and Pups are high-level architectures that omit spreading activation and other memory mechanisms, but are intended to be homomorphic to ACT* in all other respects. Exactly how these higher level architectures map onto ACT* is not made fully explicit, although it should be if the ensemble is to quality as a unified theory of cognition.

3. Develop models at different levels, where each level is a copy of the one below it. This approach seems to characterize the Soar work (Newell, 1990). Soar has been used as a model of lower level processes, such as stimulus response compatibility and transcription typing (John, 1988; Rosenbloom & Newell, 1987), where its cycle times correspond roughly to the frequency of updates to human memory. Soar has also been used for modeling computer configuration, algorithm design and other higher level problem solving tasks (Rosenbloom, Laird, McDermott, Newell, & Orchiuch, 1985), where its cycle times correspond to seconds or minutes of real time. In principle, the primitives provided by the authors of these higher level models could be replaced by Soar programs that are similar to those used in the modeling of the lower level processes.

We think that the second approach is the best. It allows models of higher level processes to be expressed in any way that optimizes the clarity and productivity of the explanations. The third approach forces the theorist to use the same architecture for both low level and high level models, and that seems analogous to forcing the quantum physicist and the biologist to use the same mathematics for their models. In principle it could be done, but the clarity of the models would be sacrificed.

In summary, Teton and similar architectures should not be viewed as claims about innateness, universality or penetrability. They should be viewed as part of a model that explains problem solving and skill acquisition. The model contains assumptions about what capabilities are possessed by subjects prior to training. Some, but not all, of those capabilities are modeled by features of the architecture. The others are modeled by pre–existing programs. Eventually, this model should be related via explicit mappings to models of lower–level processes, notably memory, attention, perception and motor control.

The remainder of this chapter contains another explanation of the psychological status of goal reconstruction. We claim that goal reconstruction is a *prior capability* of problem solving, which means that all subjects possess this capability prior to learning the given problem solving procedure. One way to see what this means is by seeing what other prior capabilities would be needed in a model of skill acquisition. The following sections list capabilities that, in our estimation, are prior capabilities for the tasks usually studied in the problem-solving literature: physics, blocks world, Tower of Hanoi, algebra, eight puzzle, etc.

Goal Reconstruction

The key property that makes goal reconstruction a candidate for a prior capability is that it does not have to be learned, or at least that it does not have to be learned each time a new procedure is learned. To demonstrate this, consider a gedanken experiment. Suppose we train subjects in an entirely novel procedure, being careful never to interrupt them while they are executing the procedure. When they have mastered it, we perform the telephone test: we interrupt them in the middle of solving a problem, have them engage in an interference task sufficient to wipe out goal memory, then have them resume their original task. Presumably, they would all be able to reconstruct their internal state for this procedure, even though they had never done reconstruction on this procedure before. This gedanken experiment shows that their reconstructive capability was acquired prior to the acquisition of the procedure.

As the discussion earlier in the chapter indicated, some cases of goal reconstruction seem to require task-specific knowledge. These kinds of goal reconstruction would have to be acquired along with the task's procedure. Although we claim that some goal reconstruction is a prior capability, we are not claiming that *all* goal reconstruction is due to prior capabilities. Teton's architecture embeds specific claims about what kinds of goal reconstruction are prior and what kinds would have to be learned.

Explanation of Worked Examples

Another capability that seems to come "for free" when one learns a procedure is explanation of worked examples. A worked example is a problem that has been solved in such a way that a partial trace of the solution process is available. Math and physics textbooks have many worked examples. Usually, the textbooks print only the results of visible actions of the procedure, the actions that the students would write if they were solving the procedure. The invisible actions, such as deciding which goal or strategy to pursue, are usually left out. Often the exact nature of the visible actions is underspecified, too. For instance, the textbook might print an algebraic equation but not say what operation was used to produce it. *Explaining* a worked example entails producing all the information that is necessary for solving the problem but has been left out of the printed material.

There is ample evidence that people can explain worked examples even when the procedure they are using to explain the example is new to them (Chi, Bassok, Lewis, Reimann, & Glaser, 1989; Pirolli & Bielaczyc, 1989). This indicates that the ability to explain a worked example is a prior capability. That is, after one has learned a procedure well enough to execute it, then one can automatically explain examples with it as well. The converse may also be true (Chi et al., 1989).

It could be objected that explaining an example is exactly the same as solving the example's problem. This is true only of simple cases. In more complicated cases, the example might not use exactly the same order of steps as the subject would use. It might produce intermediate steps that the subject would not, or use less efficient strategies than the subject would. The subjects most control their own processing so as to reproduce the same steps as the example. So example explaining really is a different process than interpreting a procedure. Thus, it should be viewed as a distinct prior capability.

Impasse Handling

When people are executing a procedure, even a fairly well-known procedure, they sometimes get stuck. For instance, if you normally make a white sauce using butter, flour and milk, and you discover, after mixing the butter and flour together and cooking them for a while, that you are out of milk, then you are at an impasse. People seem to have a fairly standard set of capabilities for handling impasses. For instance, one standard so–called *repair strategy* is substitution (Brown & VanLehn, 1980). In the case of the white sauce procedure, the cook might substitute for the milk something that is liquid, edible, mildly flavored and otherwise quite similar to the milk. Another repair is backing up. In the case of the white sauce, one might back up to the procedure that required the sauce (e.g., your favorite moussaka recipe) and reconsider the need for the sauce.

Repair strategies seem to be somewhat independent of the impasse and the procedure that they are applied to (VanLehn, 1990). For instance, the two white sauce repair strategies, substitution and backing up, are also applied by arithmetic students to arithmetic procedures (VanLehn, 1990). This illustrates the claim that people have a stock of general purpose repair strategies that can be adapted for use with any procedure's impasses. The impasse-repair process is a prior capability because it does not have to be learned as each new procedure is learned.

Rule Acquisition Events

It is often conjectured that human problem solvers can interrupt their procedure, reason about the procedure and its efficacy, make a modification to the procedure, and resume execution of the modified procedure. In early work, the existence of these rule acquisition events was inferred from changes in the person's problem solving behavior (e.g., Anzai & Simon, 1979; Neches, 1987). Recent fine-grained protocol analyses have shown that people tend to pause and/or make unusual verbal comments during a rule acquisition event (Siegler & Jenkins, 1989; VanLehn, 1991, 1989). For instance, in one 90-minute protocol (VanLehn, 1991) there were 11 rule acquisition events of which 10 were accompanied by either long pauses, reflective comments (e.g., "It's just like moving four, isn't it?") or negative comments (e.g., "Wrong . . . this is the problem and . . ."). These detailed analyses support the hypothesis that people can reason about and modify their procedures even in the midst of using them.

We are currently developing detailed simulations of rule acquisition events taken from protocols of students learning college physics. It is already clear that the subjects have a large variety of rule acquisition methods that they use to analyze and modify their procedures. For instance, a particularly powerful and common method is *explanation-based learning of correctness* (VanLehn, Ball & Kowalski, 1990). When subjects try to explain a worked example and their knowledge of the target procedure is incomplete, then they will sometimes be unable to complete an explanation of the example. There will be segments of the example's solution that cannot be parsed by the student's procedure. One rule acquisition method is to invent new rules that will complete the example's explanation (Ali, 1989; Danyluk, 1989; Fawcett, 1989; Pazzani, 1988; Schank & Leake, 1989; VanLehn, 1987; Wilkins, 1988). In general, there are combinatorially many ways to build a syntactically correct an explanation (Nowlan, 1987). Rather than exhaustively searching for the semantically correct completion, the physics students seem to specialize an overly general rule that they would not normally use. For instance, one student could not explain where a certain minus sign came from in a physics equation. She eventually formed an explanation after noting that the quantity bearing the minus sign came from a vector whose x–projection lay along the negative x-axis. She said, "The reason the negative is there is because the X component is in the negative direction on the X axis." Apparently this subject used the overly general rule that mathematical manipulations conserve negations. On this reasoning, the negation in her equation had to come, ultimately, from some existing negation, such as the negative part of the X–axis. This kind of explanation completion is halfway between syntactic explanation completion (e.g., VanLehn, 1987), where completions are chosen based on their size or other structural characteristics, and explanation-based learning (Mitchell, Keller, & Kedar-Cabelli, 1986), where new rules are created by specializing general rules that are used in an explanation. In explanation-based learning of correctness, the subject normally avoids certain rules because they are known to be overly general. However, such rules will be used to bridge an impasse, and if this ultimately results in a correct solution, a specialization of the overly general rule is kept as a new, hopefully correct rule.

Explanation-based learning of correctness is just one of many rule acquisition methods that seem to be used by people in order to improve their understanding of a task domain. Since they are used in the course of acquiring a procedure, they must have existed before the procedure. Thus, they are a prior capability.

Conclusions

This list has illustrated just a few prior capabilities that a theory of skill acquisition would need to assume. Some capabilities, such as goal reconstruction and explanation of worked examples, are best modeled as features of the architecture. Other prior capabilities, such as reading and writing English, are best modeled as procedural knowledge. Still other capabilities, such as repairing impasses, are best modeled as a mixture of architectural features and procedural knowledge.

Whether a capability is modeled as procedural knowledge or a feature of the architecture is independent of whether it is a prior capability or acquired during the observation period. Indeed, we see no logical problems with hypothesizing that some features of the architecture are acquired. (Although we *do* see interesting technical challenges in developing a learning mechanism that modifies the architecture.)

Having distinguished prior capabilities from architectures, both computational and cognitive, we hope we have clarified the main psychological claim of the chapter, which is simply that goal reconstruction is a prior capability for classical problem solving and skill acquisition.

APPENDIX: TETON

Teton is a von Neuman machine, so it has two kinds of memory. The *knowledge base* is a large, slowly changing memory that holds general knowledge, such as procedures for solving problems, inference rules and general facts. The *working memory* is a rapidly changing memory that holds information produced in the course of a computation. Like all von Neuman machines, Teton has an built-in *execution cycle* that interprets procedural knowledge stored in its knowledge base. The execution cycle consists of (a) deciding what to do, based on the current states of the working memory and the knowledge base, and (b) doing what it decided to do. The execution cycle is an algorithm that treats the information in the working memory and the knowledge base as formatted data. The format of the data is called the *representation language*.

This description of Teton has, so far, said nothing that would distinguish it from any other von Neuman machine. To define Teton per se, the following three sections will describe, respectively, its representation language, its execution cycle and its memories.

Knowledge Representation

Teton's representation language is appropriate for procedural knowledge, but clumsy at best for representing declarative knowledge. For instance, it is simple to represent addition and subtraction algorithms, but it is difficult to represent that addition and subtraction are inverses. This is *not* intended to be a claim that the mind has only clumsy ways to represent declarative knowledge. It means only that we have not investigated tasks where declarative knowledge has a major influence, so we have not yet included a language appropriate for representing declarative knowledge.

In working memory, the main unit of information is the *goal*. A goal serves many purposes. It can represent an action that has already been completed, or an action that is planned but not yet begun, or an action that is in progress. A goal has slots for indicating a state to be achieved, an operation, the state resulting from the operation, subgoals created by the operation, the supergoal of this goal, the time that the goal was created, and so on.

In the knowledge base, there are two kinds of knowledge: *operators* and *selection rules*. Operators have the following parts:

1. A goal type, which indicates what kinds of goals this operator is appropriate for. This description usually has variables that must be instantiated before the operator can be executed.
2. A set of preconditions. If all these predicates hold of the current state of working memory, then the operator can be executed. If not, then the architecture will automatically create subgoals for each of the unsatisfied preconditions. Operators may have an empty set of preconditions.
3. A body, which describes what is to happen when the operator is executed. If the operator is a primitive, the body describes the changes that will occur to the situation and/or the rest of working memory. If the operator is non-primitive (i.e., a macro–operator), the body describes what subgoals the operator will create when it is executed.
4. A shortcut condition, which is true if the operator can be assumed to be completed.

Teton's operators allow both deliberate subgoaling and operator subgoaling. The execution of the body of an operator can create subgoals (deliberate subgoaling), and the architecture will create subgoals if an operator's preconditions are unsatisfied (operator subgoaling).

Selection rules are the other type of knowledge in Teton's knowledge base. They are used for selecting a goal to work on and for selecting an operator to use for achieving the selected goal. There are three types of selection rules. *Consideration* rules indicate that a goal or operator should be considered. These rules are consulted first. They usually produce a large set of items. *Rejection* rules are consulted next, and cause some of the item to be removed from the set of items under consideration. *Preference* rules are consulted last. They partially order the set of items under consideration. Normally, one item will be preferred over all the others. It is the one selected. Teton's selection rule mechanism is similar to the ones used by Soar (Rosenbloom et al., chap. 4, in this volume) and Prodigy (Carbonell et al., chap. 9, in this volume). All three systems use this type of mechanism because it makes it easy to implement the acquisition of strategic knowledge: just add new selection rules.[9]

The Execution Cycle

The main loop of Teton's interpreter is shown in Table 6.2. Most of it is quite standard: Goals are selected by goal selection rules. Operators are selected by operator selection rules. Unsatisfied preconditions cause subgoaling. Execution of macro-operators causes subgoaling. Execution of primitive operators causes state changes. However, there are two facilities, impasses and shortcut conditions, that are not standard and deserved some explanation.

Whenever the architecture needs to select a goal or operation, it enumerates all possible candidates, filters this set with the rejection–type selection rules, then rank orders the set with the remaining selection rules. If one choice is better than all the others, then Teton takes it. However, if the selection rules fail to uniquely specify a choice (e.g., they reject all possibilities, or they cannot decide between two possibilities), then an *impasse* occurs. As in Soar (Rosenbloom et al., chap. 4 in this volume) and Sierra (VanLehn, 1987; VanLehn, 1990), an impasse causes the architecture to automatically create a new goal, which is to resolve the impasse. Typically, such resolve-impasse goals are tackled by task-general knowledge. For instance, one of Sierra's methods is: If the selection rules

TABLE 6.2
The Main Loop of Teton's Interpreter

1. Select a goal from working memory using the goal selection rules. If there is no unique selection exists, then create an impasse goal describing that and select it.
2. If the selected goal has an operation selected for it already, then skip the next step.
3. Select an operation (a partially instantiated operator) for the current goal using the operator selection rules. If there is no unique operation, then create an impasse goal describing that, make it a subgoal of the selected goal, select it, and repeat this step.
4. If the selected operation has unsatisfied preconditions, then create a new goal for each such precondition and link it to the selected goal as a subgoal. Leave the selected goal marked "pending," and return to step 1.
5. If the selected operation has a shortcut condition and it is true, or it has subgols and they are all completed, then mark the selected goal "completed" and return to step 1.
6. If the operation is primitive, then execute the operation, mark the selected goal "completed", and return to step 1.
7. Otherwise, the operation is non-primitive, so execute the operation and return to step 1. Execution will cause new subgoals to be created and linked to the seleted goal as subgoals.

cannot decide among several possible candidates, then choose one randomly. Another popular impasse-resolving method is: If the selection rules rejected all operations for the current goal, then mark the goal as accompanied even though it is not. This causes the architecture to "skip" planned actions that it does not know how to accomplish. Brown and VanLehn (1980) exhibited a collection of such impasse-resolving methods (called "repairs") and showed how they could explain the acquisition of many students' bugs (procedural misconceptions).

Shortcut conditions play an important role when Teton reconstructs goals that have been forgotten (i.e., deleted from working memory). In order to recover from such working memory failures, Teton has to reconstruct some of the goals it once had. It is assumed that there is some top-level goal that is not forgotten. The remaining goals are reconstructed by simply executing the procedural knowledge with the interpreter of Table 6.2. However, when the situation corresponds to a half–completed problem, some of the goals created are superfluous because they have already been achieved. In such cases, the appropriate shortcut conditions are true, and goals are marked "completed" before any attempt is made to execute them.

One mechanism that is common in other architectures is missing in Teton. Teton goals need not be selected in last-in-first-out (LIFO) order. For instance, if there are two pending goals, A and B, and A is selected and leads to a subgoal C, then a LIFO restriction would rule out selecting goal B since C is more recently created. Most architectures, including Soar and Grapes, place a LIFO restriction on goal selection, but Teton does not. In the case just mentioned, it allows either B or C to be selected.

Memories

As mentioned earlier, Teton has two memory stores, the knowledge base and the working memory. Working memory is composed of four distinct memories:

1. The *main working memory* is the one that holds the goals and other data structures generated by the execution cycle.
2. The *situation* holds a representation of the external environment. Its contents model the subjects' interpretation of what they see, which is task-specific, like a problem space's current state. For instance, an arithmetic problem is represented as a grid of rows and columns in the situation, whereas an algebra equation is represented as a tree.
3. The *scratchpad* is just like the situation, except that the contents represent something that the subject is imagining, rather than actually seeing. For instance, some subjects imagine the result of a move during problem solving before actually making the move in the real world. In order to model such events, Teton distinguishes the situation from the imagination.

4. The *buffer* is a limited capacity store for items that have simple verbal encodings, such as numbers

The latter two memories are a novelty in computational models of the architecture, so they are worth a little explanation. They are designed as simple versions of the two slave memories described by Baddeley (1986) and called the *articulatory loop* and the *visio-spatial scratchpad*. According to Baddeley, the articulatory loop consists of a passive storage medium, called the phonological store, and a mechanism for "rehearsing" its contents (analogously to a dynamic RAM). The phonological store can hold a phonological code for about 2 or 3 seconds (Zhang & Simon, 1985). If it is not rehearsed in that time, it becomes inaccessible. The time required to rehearse a code is linearly related to the time required to read the equivalent lexical item. Thus a person can store a given list of stimulus items if the time required to rehearse them once is less than 2 or 3 seconds. This accounts for the often-cited finding that untrained subjects can store and immediately recall about 7 plus or minus 2 chunks (Miller, 1956). Because rehearsal can go on relatively independently of most cognitive tasks (Baddeley, 1986), the articulatory loop acts like a short term store with a capacity of a few phonologically encoded chunks. Teton uses this much simpler model, and allows N chunks to be stored in the articulatory loop, where N is a parameter of the architecture. Typically, the articulatory loop is used for temporary storage of numbers.

The visual–spatial scratchpad contains the same kind of items as the situation does, but it is meant to model a scene that the subject is imagining, rather than the real world. Teton's version of the scratchpad is only used for one purpose, which is looking ahead during problem solving in order to project the consequences of contemplated moves. Consequently, Teton supports only a simple model of the scratchpad. There is a switch in the architecture, which can be set by a primitive operation to either "normal" or "imaginary." When the switch is thrown from "normal" to "imaginary," the scratchpad is initialized with a copy of the items in the current situation. Thereafter, all reading and writing operations that would normally access the situation access the scratchpad instead. The volatility of the scratchpad is modeled, again quite crudely, by counting the number of operations applied to it. After a threshold is crossed (the threshold is a parameter of the model), the contents of the scratchpad become inaccessible.

This facility was used to simulate look-ahead search in the Tower of Hanoi, which plays a crucial role in Anzai and Simon's (1979) account of strategy acquisition. In the course of developing a similar account of strategy acquisition, we discovered that learning the more advanced versions of the disk subgoaling strategy would require looking ahead 12 moves in the scratchpad. Not only is this implausible, but setting the stability parameter of the scratchpad to 13 caused learning of earlier versions of the strategy to go awry. This led us to look for methods of strategy acquisition that did not use the scratchpad. We found not one but several, along with good support for them in the protocol data (VanLehn, in press, 1989).

ACKNOWLEDGMENTS

This work was supported by the Personnel and Training Research Program, Psychological Sciences Division, Office of the Naval Research, under contract N00014-88-K-0086 and by the Information Sciences Division, Office of Naval Research, under contract N-00014-K-0678. Reproduction in whole or in part is permitted for any purpose of the United States Government. Approved for public release; distribution unlimited.

REFERENCES

Agre, P. E., & Chapman, D. (1987). Pengi: An implementation of a theory of activity. In K. Forbus & H. Shrobe (Ed.), *Proceedings of the Sixth National Conference on Artificial Intelligence*. Los Altos, CA: Morgan Kaufman.

Ali, K. M. (1989). Augmenting domain theory for explanation-based generalisation. In A. Segre (Ed.), *Proceedings of the Sixth International Workshop on Machine Learning*. Los Altos, CA: Morgan Kaufman.

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

Anderson, J. R. (1989). A theory of the origins of human knowledge. *Artificial Intelligence, 40*, 313–352.

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

Baddeley, A. (1986). *Working memory*. Oxford, UK: Clarendon Press.

Berwick, R. (1985). *The acquisition of syntactic knowledge*. Cambridge, MA: MIT Press.

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

Charniak, E., & McDermott, D. (1986). *Introduction to artificial intelligence*. Reading, MA: Addison-Wesley.

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, P., & Glaser, R. (1989). Self explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13*, 145–182.

Danyluk, A. P. (1989). Finding new rules for incomplete theories: Explicit biases for induction with contextual information. In A. Segre (Ed.), *Proceedings of the Sixth International Workshop on Machine Learning*. Los Altos, CA: Morgan Kaufman.

de Kleer, J. (1986). An assumption-based truth maintenance system. *Artificial Intelligence, 28*, 127–162.

de Kleer, J., Doyle, J., Steele, G. L., & Sussman, G. J. (1977). Amord: Explicit control of reasoning. *Sigplan Notices, 12*(8), 116–125.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence, 12*(3), 231–272.

Ernst, G. W., & Newell, A. (1969). *GPS: A case study in generality and problem solving*. New York: Academic Press.

Fawcett, T. E. (1989). Learning from plausible explanations. In A. Segre (Ed.), *Proceedings of the Sixth International Workshop on Machine Learning*. Los Altos, CA: Morgan Kaufman.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence, 3*, 251–288.

in the protocol data (VanLehn, in

ENTS

ınd Training Research Program,
: Naval Research, under contract
iences Division, Office of Naval
production in whole or in part is
Government. Approved for public

S

tation of a theory of activity. In K. Forbus
Conference on Artificial Intelligence. Los

ınation-based generalisation. In A. Segre
p on Machine Learning. Los Altos, CA:

'ambridge, MA: Harvard.
ın knowledge. *Artificial Intelligence, 40,*

ing by doing. *Psychological Review, 86,*

ırendon Press.
lge. Cambridge, MA: MIT Press.
. generative theory of bugs in procedural

artificial intelligence. Reading, MA: Ad-

laser, R. (1989). Self explanations: How
oblems. *Cognitive Science, 13,* 145–182.
theories: Explicit biases for induction with
s of the Sixth International Workshop on

ıance system. *Artificial Intelligence, 28,*

. J. (1977). Amord: Explicit control of

Intelligence, 12(3), 231–272.
in generality and problem solving. New

ms. In A. Segre (Ed.), *Proceedings of the*
Los Altos, CA: Morgan Kaufman.
ing and executing generalized robot plans.

John, B. (1988). *Contributions to engineering models of human-computer interaction.* Doctoral dissertation, Dept. of Psychology, Carnegie Mellon University.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33,* 1–64.

Miller, G. A. (1956). The magic number seven plus or minus two: Some limits on our capacity for processing information. *Psychological Review, 63,* 81–97.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1*(1), 47–80.

Neches, R. (1987). Learning through incremental refinement of procedures. In D. Klahr, P. Langley, & R. Neches (Ed.), *Production systems models of learning and development.* Cambridge, MA: MIT Press.

Newell, A. (1973). You can't play 20 questions with nature and win. In W. G. Chase (Ed.), *Visual information processing.* New York: Academic.

Newell, A. (1990). *Universal theories of cognition.* Cambridge, MA: Harvard University Press.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

Nowlan, S. (1987). *Parse completion: A study of an inductive domain* (Technical Report PCG11). Department of Psychology, Carnegie Mellon University.

Pazzani, M. (1988). Integrated learning with incorrect and incomplete theories. In J. Laird (Ed.), *Proceedings of the Fifth International Workshop on Machine Learning.* Los Altos, CA: Morgan Kaufman.

Pirolli, P., & Bielaczyc, K. (1989). Empirical analyses of self-explanation and transfer in learning to program. In G. Ohlson & E. Smith (Ed.), *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society.* Hillsdale, NJ:Lawrence Erlbaum Associates.

Pylyshyn, Z. W. (1984). *Computation and cognition: Toward a foundation for cognitive science.* Cambridge, MA: MIT Press.

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., & Orchiuch, E. (1985). R1-Soar: An experiment in knowledge-intensive programming in a problem solving architecture. *Pattern Analysis and Machine Intelligence, 7,* 561–567.

Rosenbloom, P., & Newell, A. (1987). Learning by chunking: A production system model of practice. In D. Klahr, P. Langley, & R. Neches (Ed.), *Production system models of learning and development.* Cambridge, MA: MIT Press.

Schank, R. C., & Leake, D. B. (1989). Creativity and learning in a case-based explainer. *Artificial Intelligence, 40,* 353–386.

Siegler, R. S., & Jenkins, E. A. (1989). *How children discover new strategies.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Suchman, L. A. (1987). *Plans and situated actions: The problem of human-machine communication.* New York: Cambridge University Press.

VanLehn, K. (1983). *Felicity conditions for human skill acquisition: Validating an AI-based theory* (Tech. Report CIS-21). Xerox Palo Alto Research Center. Out of print, but available as publication number 9018167 from University Microfilms, 300 North Zeeb Road, Ann Arbor, MI 49106.

VanLehn, K. (1986). Arithmetic procedures are induced from examples. In J. Hiebert (Ed.), *Conceptual and procedural knowledge: The case of mathematics.* Hillsdale, NJ: Lawrence Erlbaum Associates.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence, 31*(1), 1–40.

VanLehn, K. (1989). Learning events in the acquisition of three skills. In G. Ohlson & E. Smith (Ed.), *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society.* Hillsdale, NJ: Lawrence Erlbaum Associates.

VanLehn, K. (1990). *Mind bugs: The origins of procedural misconceptions.* Cambridge, MA: MIT Press.

VanLehn, K. (1991). Rule acquisition events in the discovery of problem solving strategies. *Cognitive Science, 15*(1), 1–47.

VanLehn, K., Ball, W., & Kowalski, B. (1989). Non-LIFO execution of cognitive procedures. *Cognitive Science, 13,* 415–465.

VanLehn, K., Ball, W., & Kowalski, B. (1990). Explanation-based learning of correctness: Towards a model of the self-explanation effect. In *Proceedings of the 12th Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Vere, S. (1975). Induction of concepts in the predicate calculus. In *Proceedings of the Fourth IJCAI*. Los Altos, CA: Kaufmann.

Wilkins, D. C. (1988). Knowledge base refinement using apprenticeship learning techniques. In R. Smith & T. M. Mitchell (Ed.), *Proceedings of the Seventh National Conference on Artificial Intelligence*. Los Altos, CA: Morgan-Kaufman.

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

Zhang, G., & Simon, H. A. (1985). STM capacity for Chinese words and idioms: Chunking and acoustical loop hypotheses. *Memory and Cognition, 13*(3), 193–201.