AN INTRODUCTION TO SOAR PROGRAMMING

John Rieman, MRC-APU
15 March 1995

e-mail: John.Rieman@Colorado.edu

web: http://www.cs.colorado.edu/~rieman
[put into postscript from rtf, some page breaks put in, and moved to Europe by FER 26Nov96]

Abstract

This is a bare-bones introduction to the Soar programming language, reflecting my experience in learning the system during the last few months. It's written for anyone who's starting to program in Soar. I hope it will act as a "jump start" so the *Soar User's Manual* is easier to understand on a first reading. As such, this is intentionally not a complete presentation of the language, and it only touches on Soar theory.

The material begins with an overview. The second section is an ultra-simple but detailed example, with code supplied at the end of the document. The remaining sections cover a potpourri of topics, answering a lot of questions I asked while I was learning the language. I've suggested some exercises, which are versions of things I did to probe the system's behavior. The Soar manual itself gives a simple description of the commands for running Soar programs, so I haven't discussed that at all.

Applicability

The text and examples describe Soar version 6, nnpscm. A paragraph in the Environment section notes the difference between nnpscm Soar and earlier versions.

Contents

I. OVERVIEW OF SOAR

Soar: Theory, Language, Models, Programs, and System

The Soar theory describes cognition: how people think. It focuses especially on how they solve problems and how they learn. The basic theory was developed by Allen Newell, John Laird, and Paul Rosenbloom.

The Soar language is a specialized programming language, based on the Soar theory. It's designed to describe the knowledge people have about the world, and how that knowledge is mentally represented.

A Soar program, or model, is a description of the knowledge — the facts, skills, and concepts — that a person (or a machine) would need to think about and solve a specific problem, such as playing checkers or planning a plane trip.

The Soar system is a computer application. (An "application" is a piece of computer software, like a spreadsheet or a word processor.) The Soar system "runs" models written in the Soar language. This means that it uses built-in, theoretically based mechanisms to step through the way a person might think about and solve problems, if that person had the skills and concepts the model represents.

From this point on I'll often just refer to "Soar." I'll usually be talking about the language as run by the system.

The Business of Modelling

The Soar system by itself, without a model (a program), won't play checkers or plan a trip or do much of anything else, just as an empty spreadsheet won't perform any calculations. So questions like, "How would Soar play checkers?" should probably be recast as, "Is there a Soar model of playing checkers, and how does it work?"

In fact, there can be more than one Soar model of any activity. Each model will include knowledge about whatever the modeller thinks is relevant to the problem, such as the schedules of the airline or the rules of checkers. But another modeller might theorize that people rely on knowledge about different things, such as the quality of the airline's meals or the facial expressions of the checkers opponent. The way that Soar would use the knowledge in each model would be constrained by the Soar theory, language, and system. But the models could still be very different.

Many Soar models are written as part of a research effort to understand how people think about a problem. However, Soar is also used as an artificial intelligence system, which can control robots or other machines.

Productions: Soar's Way of Representing Knowledge

In programming language terms, Soar is a production system. "Productions" are if-then pairs. Here are some simple examples, in English:

> IF the alarm is ringing THEN get up.
> IF you're planning a vacation THEN consider the weather.
> IF it's a small, white, smoking cylinder THEN it's a cigarette.

In a Soar program, all knowledge about the world and what to do there is encoded as productions. They're written in the Soar language instead of English, but the difference doesn't matter for this discussion. When a Soar program runs under the Soar system, the THEN side of each production is acted on whenever the IF side matches the current state of the world. With the right productions, a Soar model can solve any problem or perform any mental task that a human could — at least, that's the theory.

The first two productions above have a special status in Soar. They represent "operators." Operators cause things to happen, either in the real world or in the imagination. Deciding on operators (i.e., "Now I'll consider the weather") and applying them ("OK, it will be June, so I think it will be warm, but there might be rain...") is a fundamental part of the Soar theory. But it's all done with productions.

Here's how a robot office worker might work, entirely programmed with productions. The robot arrives at the office in the morning and there are items in its in-box. A production says, IF there are documents in the in-box THEN pick up the first one. Another production says IF you're holding a document THEN read it. Just those two productions get the robot started. Now, if it has enough additional productions, it can also deal with each document. One production might be, IF this is an advertisement THEN throw it away. The robot can even have productions that say things like, IF you don't know how to deal with a document THEN ask your supervisor what to do.

Notice that all these productions can be "in the robot's head," ready to use, at all times. Only the ones that match the current state of the world will be applied. So, unlike more traditional computer programs, a production system doesn't have sequences of instructions that are followed, step by step, in a predetermined order.

Subgoaling: Soar's Special Approach to "Conflict Resolution"

Here are some productions that would help the office-worker model decide what to do at lunch time:

```
IF it's noon THEN it's lunchtime
IF it's lunchtime and it's Friday
        THEN skip lunch and exercise
IF it's lunchtime and it's payday
        THEN eat a big steak at Fred's Diner
```

Now, how should the office worker behave if payday falls on a Friday? The model would have what's technically called a "conflict resolution" problem. Two productions match the current conditions, but they suggest conflicting actions: skip lunch and exercise versus eat steak at Fred's. There are lots of ways this might be handled, most of which were tried in production systems before Soar. For example, the model might pick the production that was used least recently, or it might pick at random, or it might pick the production that had been most useful in the past.

Soar (the system and the theory) takes a unique and important approach to this situation. It considers the THEN side of all productions that could be applied under the current conditions. If there's an unresolved conflict over what to do, it suspends work on its current problem (which it can't continue with because of the question) and applies itself to resolving that conflict.

In the robot office-worker's example, the current problem is to control its behavior as the day at the office progresses. But that problem has produced a "tie" between two suggested

actions — eat and exercise — so resolving that tie becomes the new current problem. The old problem of controlling the day's activity isn't forgotten. It's just postponed until the eat/exercise issue is resolved.

The technique of setting the main problem aside in order to work on a lesser problem that's blocking progress is called "subgoaling." When the Soar system creates a subgoal, it changes the state of the world in its internal representation. Specifically, the world includes all the facts it included before, but it now also includes the fact that the current problem is to resolve the question that's blocking progress. Because this is part of the state of the world, it's something that the IF side of productions can match. So, if the model has a production that tells what to do if two lunch proposals are tied, that production will now be applicable. For example, the office worker might have this production:

IF resolving a tie between eating and exercise
THEN always prefer exercise.

That will resolve the eat/exercise issue, so the Soar system can forget about the subgoal and get back to the main problem of controlling its office activities. Of course, the next action is now specified, so the system can deal with whatever problem arises after that, such as controlling its exercise routine.

Subgoaling is a powerful technique, and Soar does this whenever it can't decide what else to do. (This is called "universal subgoaling.") Sometimes it can't decide because two or more suggested actions seem equally appropriate, as in the example. Other times no action at all is proposed, or an action has been decided on but no productions apply that tell Soar how to do it.

If Soar runs into further trouble while working on a subgoal, it will just create another subgoal — a sub-subgoal — and try to deal with that. It can create as many levels of subgoals as it needs.

Chunking: Learning Through Subgoaling

Like Soar, people often find themselves pausing to think through their options before taking action. Sometimes they have to think through several subgoals before they get the answer. For example, I might decide to go to Paris. I'd think: to get to Paris I'll have to fly, but that means I'll have to get airline tickets, so I'll have to call my travel agent, so what I need to do now is look up the agent's phone number. People can remember the end-products of these chains of reasoning, which will let them avoid the reasoning itself in the future. Next time I decide to take a trip to Paris, the first thing I'll do is reach for my phone book.

Soar has the same ability. It learns from its deliberations. Every time Soar resolves a problem in a subgoal, it remembers the solution. The next time the same problem occurs, that memory is available and Soar can avoid the time and trouble of working in the subgoal. It may even learn general principles from solving specific problems — for example, it might learn to reach for its phone book when planning any long trip, not just trips to Paris or other places it had thought about before.

The memory of past problem solving in Soar is stored as "chunks." Chunks are nothing more than productions produced by the Soar system instead of the programmer. The chunk learned by the robot office worker would say:

IF exercise and steak at Fred's Diner are both possible actions
THEN choose exercise

The chunk's IF side includes exactly those facts used for problem solving in the subgoal — it won't include other things that might be true at the time, such as the fact that it's noon and Friday and payday. The THEN side describes the result of the problem solving. And, although it may not be obvious from the English-language version above, the chunk will apply as soon as the two actions are considered, without subgoaling.

Once a chunk is formed, the Soar system treats it exactly as it treats the productions written by the programmer. In fact, the Soar theory holds that all knowledge, or at least all except a few very basic productions, is learned through chunking.

The Knowledge Level and Soar Input/Output

Chunking is the method Soar uses to learn from its deliberations. In a sense, however, the chunk learned by the robot office worker doesn't contain any knowledge that the robot didn't already have. It's just a restatement of existing knowledge, which will now apply exactly when it's needed. It would be a much different situation if the robot had learned from the radio that the Dodgers had won the ball game, 6 to 2. That would be knowledge that wasn't contained in the original model, in any form.

Allen Newell used the term "knowledge-level learning" to describe the learning of facts that couldn't possibly be derived from the knowledge already available. The final score of a ball game would be knowledge-level learning; the solution to the lunch dilemma would not be.

Soar models can be built that learn at the knowledge level, and the things they learn will be represented as chunks. These models need some way of interacting with the world outside of the computer, in order to acquire their new knowledge. The Soar system provides input and output routines for this. These make it possible to write productions that print things on the computer screen, such as "What was the final score of the Dodgers game?" Other productions can take input from the keyboard and feed that into Soar's temporary working memory. If the input is later used while in a subgoal, then Soar will learn a chunk that includes the input. The information in working memory will change as the external world changes, but the chunk will become part of the model's permanent memory.

A word of caution, however: Many Soar programs don't strictly follow the convention of supplying all new knowledge through the input routines. Often it's easier to program the input into the model itself, as Soar productions, and just pretend that it came from the outside. For example, a production might say:

IF you are listening to the radio at 11:30 a.m.
THEN you hear that the Dodgers won, 6 to 2

Conversely, many models send output to the screen about what they're thinking, so the programmer can track the model's behavior. This may make it look like a model has made decisions and acted on them, when it's really just thinking things over.

A careful reading of the source code should make it clear which productions are simulating the external world and which are really part of the cognitive model.

Review of the Overview

Here's a review of the points covered so far. Soar is a theory of cognition. Researchers write and run Soar models that simulate the thought processes people would use to solve problems and learn about different domains.

Four key characteristics of Soar are: (1) It is a production system, and things run smoothly as long as there are productions that unambiguously state what to do next. (2) Whenever it has a problem deciding what to do next, Soar sets itself the subgoal of resolving that problem. (3) Whenever Soar successfully resolves a problem by subgoaling, it remembers that solution as a chunk, so next time the problem arises there will be no need to subgoal. (4) To acquire completely new facts ("knowledge-level" learning), Soar has to use its input/output facilities to get information from the real world.

II. SOAR PROGRAMMING BASICS

A Soar model is not a trivial programming exercise. There is nothing like a Soar "Hello World" program that you can write in a couple of lines, or a simple while-loop that you can just fill in with your code. So, unlike C or Pascal or Basic, it's difficult to learn Soar by starting with a trivial program and incrementally adding features.

In addition to its overall complexity, Soar is difficult to learn because a lot of what the system does goes on "behind the scenes." There are critical mechanisms that don't show up in the typical trace format, and that can't be directly controlled by the code you've written. These mechanisms aren't a big secret — they're described (rather formally and cryptically) in the official *Soar User's Manual,* and they can be examined with various tracing and debugging tools as a model runs. But they're deep enough below the surface structure that visualizing them can be difficult when you first get started.

Finally, Soar is difficult to learn because it's usually presented as a theory of cognition, not as a programming system. That means you have to learn the theory, then translate the theoretical ideas into the mechanism. This is a little like the way college calculus is often taught: first you learn six weeks worth of definitions of limits and deltas and epsilons and infinite series, then you finally learn some simple algorithms that let you "do" calculus, without even thinking about the theoretical foundations.

The purpose of this section is to give you a programmer's fundamental understanding of how a running Soar program works, with as little theoretical baggage as possible. This will involve stepping through a program's behavior and describing what's happening, with special attention to the "hidden" parts of Soar.

Working Memory and Preferences

We'll focus again on the office-worker model. (The code for the model is supplied with this document, but I'll use pseudo-code in the text.) Soar has what's called a working memory that represents the current state of the world. For the office worker program, the working memory when the clock strikes noon on Friday (payday) might initially contain these items:

working
memory

state=top:
time=noon
weekday=Friday
day-type=payday

I've represented things roughly the way they are represented in the Soar system and language. Soar always works with attributes and values, such as attribute: time, value: noon. There are two special attributes in Soar: the state and the operator. Essentially all the other attributes are named by the modeller, similar to variable names in a traditional programming language. The memory is maintained as a tree structure, with the state as the root.

As Soar runs, it checks each of the productions that the programmer has written to see if the production's IF side matches the items in working memory. In the office worker simulation, the following productions (in pseudo-Soar code) were written by the programmer:

IF time=noon THEN agenda=lunch
IF agenda=lunch and weekday=Friday
THEN operator=exercise
IF agenda=lunch and day-type=payday
THEN operator=eat-steak-at-Fred's

Only the first of the productions matches the working memory just shown. The other two productions don't match, yet, because working memory doesn't contain agenda=lunch.

When a production's IF side matches to working memory, it "fires." In most production systems this would mean that its THEN side would be added to working memory. In Soar, however, things are more complicated. When a production fires, its THEN side is temporarily held in a sort of staging area, called preference memory. So for a moment, the Soar memory looks like this:

working *preference*
memory *memory*

state=top:
time=noon
weekday=Friday
day-type=payday
[agenda=lunch +]

The memory stays like this while the Soar system checks for any other productions that can fire on the current working memory; if it finds any, it puts their THEN sides into preference memory also. The firing of all productions that match working memory is called an elaboration cycle. In this example no other productions can fire. So Soar puts the new item into working memory:

working memory	*preference memory*
state=top:	
time=noon	
weekday=Friday	
day-type=payday	
agenda=lunch	[agenda=lunch +]

As shown above, the preference for agenda=lunch remains in preference memory even after working memory has been updated. In fact, the preferences for all the other items in working memory were also there, left over from earlier elaboration cycles. So, although it clutters the diagram, here's a more accurate representation of what memory looks like at this point:

working memory	*preference memory*
state=top:	
time=noon	[time=noon +]
weekday=Friday	[weekday=Friday +]
day-type=payday	[day-type=payday +]
agenda=lunch	[agenda=lunch +]

The "+" shown in each preference is the preference value, usually just called the preference. The + preference means lunch is an acceptable value for the agenda slot.

There is a whole range of preferences available, which can be explicitly specified in the THEN side of a Soar production (the + preference is implicit if nothing else is specified). At the end of an elaboration cycle, there will often be more than one value suggested by productions for a single working memory slot — for example, productions might fire and produce the following preferences:

working memory	*preference memory*
state=top:	
	[agenda=work +]
	[agenda=work >]
	[agenda=lunch +]

The Soar system would then examine the preferences to decide which value of agenda (if any) to put in working memory. In this case, it would choose work, because the ">" preference indicates that work is the best choice.

A similar situation could occur if agenda=work was already in working memory, and the firing of a production added new preferences for lunch:

	working memory	*preference memory*
state=top:		
	agenda=work	[agenda=work +] [agenda=lunch +] [agenda=lunch >]

In this case, agenda=work would be removed from working memory and agenda=lunch would be added — and all the preferences would remain.

These two examples reveal only a small part of a complicated scheme for arbitrating among preferences. One rule to be especially aware of is this: Nothing every gets moved from preference to working memory unless it has an acceptable (or a require) preference. The general effect of the other rules is reflected in the preference names:

~	prohibit (note that this preference is almost never used)
–	reject (and wipe out any other preferences)
+	acceptable
!	require (note that this preference is almost never used)
=	indifferent (one's as good as another)
&	parallel (allows two or more values for an attribute)
>	better (binary) or best (unary)
<	worse (binary) or worst (unary)
@	reconsider (applies only to operators)

Context Slots

Let's return to the running model of the office worker. Once working memory has agenda=lunch added, the IF sides of the other two productions also match working memory (the ones suggesting exercise and steak-at-Fred's). So those productions fire, producing:

	working memory	*preference memory*
state=top-state:		
	time=noon	[... +]
	weekday=Friday	[... +]
	day-type=payday	[... +]
	agenda=lunch	[... +]
		[operator=exercise +]
		[operator=eat-steak-at-Fred's +]

(Just to simplify the diagram, I've used "[... +]" to abbreviate the preferences for items already in working memory.)

Now, the rules for preference arbitration are applied to the preferences. Unfortunately, the rules don't say what to do if two acceptable values are proposed without any further preference to give best/better/worst or some other guidance, so Soar has reached an impasse.

Recall from the general introduction that Soar's response to any impasse is to subgoal. That's going to happen, but not quite yet. Here are the details.

Soar distinguishes between two kinds of impasses: those that involve operators and those that don't. Those that don't involve operators are, effectively, ignored: two + preferences for agenda would simply result in neither agenda value being in working memory.

But operators, which are the things that cause changes, are too important to ignore. So, when two or more operators have acceptable preferences, Soar puts copies of *both* acceptable operator preferences into working memory.

working memory	*preference memory*
state=top-state:	
time=noon	[... +]
weekday=Friday	[... +]
day-type=payday	[... +]
agenda=lunch	[... +]
[operator=exercise +]	[... +]
[operator=eat-steak-at-Fred's +]	[... +]

But notice: it is only the *preferences* that are copied into working memory. In fact, this is the way operators are always handled, whether or not there is an impasse. The acceptable preferences for all proposed operators are placed in working memory at the end of each elaboration cycle, and the decision as to which operator should actually be selected for working memory is postponed, until no more elaboration cycles can occur.

A production with an IF side specifying "operator=exercise" would not fire at this point, because no working memory value for the operator slot has yet been decided on. What might fire, however, is a production with [operator=eat-steak-at-Fred's +] on its left side — that is, a production that matched the preference. And that production, if it existed, might produce another operator preference, which could resolve the tie.

Operators are given special treatment because, as described earlier, they actually cause things to happen. The operator and the state (as we'll soon see, this can change too) are called the Soar context slots. These are is a sort of a higher form of working memory:

context slots	*working memory*	*preference memory*
state=top-state:		
operator=?		
	time=noon	[... +]
	weekday=Friday	[... +]
	day-type=payday	[... +]
	agenda=lunch	[... +]
	[operator=exercise +]	[... +]
	[operator=eat-steak-at-Fred's +]	[... +]

The Soar system tries to decide which operator to put into the context slot as soon as quiescence is achieved. Quiescence describes the condition in which elaboration cycles have stopped occurring — i.e., things have been moved from preference memory into working memory, but no productions match the revised contents of working memory. In terms of the Soar theory, the system has brought all immediately available knowledge to

bear on the problem of selecting the operator, and only now will it actually make the decision.

In fact, at quiescence, the system checks to see if either of the context slots (state or operator) need to be changed. This will mark a Soar decision cycle.

Before going into the details of how the tie between the two operators is resolved, it will be useful to look at a pseudocode description of the Soar system's basic behavior, incorporating the two cycles described.

```
Repeat forever:
/* decision cycles */

    While any productions match working memory:
    /* elaboration cycles */
        Fire all matching productions, putting results into
            preference memory.
        Arbitrate preferences and update contents of working
            memory for non-operator items.
        Copy all acceptable operator preferences into
            working memory.
    End while.

    /* quiescence */

    /* decision "phase" */
    Decide on and update context slots (states and operators).

End repeat.
```

So, to review the robot model with the above description in mind:

- In elaboration-cycle 1, the agenda=lunch production fired. There was no other contender for the agenda slot, so agenda=lunch was placed in working memory.
- In elaboration-cycle 2, the operator=exercise and operator=eat-steak productions fired. Copies of both operator preferences were put into working memory.
- No more productions fire, so we've reached quiescence.
- Now, it's time to do a decision cycle. In particular, there's an operator tie that needs to be resolved...

Impasses and Subgoaling

Which operator should go into the operator slot? Choosing the right operator is a central part of Soar's intelligent behavior, so an operator-tie impasse is a serious situation. Soar's reaction to an operator-tie impasse is to subgoal, which it does by creating a new state:

context slots	*working memory*	*preference memory*
state=top-state:		
operator=?		
	time=noon	[... +]
	weekday=Friday	[... +]
	day-type=payday	[... +]
	agenda=lunch	[... +]
	[operator=exercise, +]	[... +]
	[operator=eat-steak-at-Fred's +]	[... +]
state=substate-1:		
operator=?		
	problem=two-operators-tied	
	tied-item-1=exercise	
	tied-item-2=eat-steak-at-Fred's	
	superstate=top-state	

There are several things to notice about this situation. First, the substate contains knowledge of the reason for its own existence (an operator tie), as well as pointers to the operators that tied. Second, it contains a pointer to the state above, but almost none of the information from the state above. The pointer, however, gives access to that information. If the modeller wants to check the weekday with productions in the substate, he can write a production something like:

IF superstate.weekday=Friday THEN ...

A third point is that the top-state is still in working memory. In fact, it's in the same working memory that contains the substate. Soar hasn't set aside the top-state or the knowledge it contains. It has just run out of things to do there. So, as soon as the operator tie in the top state is resolved, work there can resume and the substate will automatically disappear. That will probably happen when the tie is resolved by work in the substate, but it could also happen if something occurs in the top-state, such as an announcement saying that payday has been cancelled.

We'll step through the substate activity to see how this all happens.

First, notice that deciding on a new state was a context decision. So, in terms of the elaboration-cycle/decision-cycle pattern, Soar has completed a decision cycle and is ready to start at the top of the loop again, checking for any productions that match the new contents of working memory. The production that the modeller has written to deal with this situation is:

```
IF      problem=two-operators-tied
        tied-item=exercise
        tied-item=eat-(anything)-(anywhere)
        superstate=top-state
THEN superstate.tied-item=exercise >
```

In other words, in this situation, put a best (>) preference on the exercise operator in the top state. The IF side of the production is an exact match to the working memory in the subgoal, so the production will fire, and memory will contain:

context slots	*working memory*	*preference memory*
state=top-state:		
operator=?		
	time=noon	[... +]
	weekday=Friday	[... +]
	day-type=payday	[... +]
	agenda=lunch	[... +]
	[operator=exercise, +]	[... +]
	[operator=eat-steak-at-Fred's +]	[... +]
		[operator=exercise >]
state=substate-1:		
	problem=two-operators-tied	
	tied-item-1=exercise	
	tied-item-2=eat-steak-at-Fred's	
	superstate=top-state	

Chunking

As described in the introductory overview, this is the point at which Soar forms chunks. As soon as the production firing in the substate reaches up and puts the > preference into the top-state, the Soar system will report, "Build: chunk-1" and the following chunk will be added to the list of productions that define this model's permanent knowledge:

```
IF      [operator=exercise +]
        [operator=eat-steak-at-Fred's +]
THEN operator=exercise >
```

This has the problem's cause on the IF side and the problem's solution (from the programmer's original production) on the THEN side. And, it is a production that will apply in the top-state as soon as the operator preferences are moved into working memory, without subgoaling.

Neat, huh? It all fits together!

Impasse Resolution and Subgoal Collapse

Now, once the > preference is added to the top state, no other production fires, so quiescence has been reached. Soar goes into another decision phase, considering first the context slots in the top state and working down. Specifically, it considers the three preferences for operator in the top state. The standard preference arbitration rules can now resolve the tie between eat and exercise, with the result that exercise is selected as the operator. It "goes into slot." That resolves the operator tie, so the substate (which depended on the tie) is eliminated. More accurately, the system eliminates the tie, and everything that depended on the tie gets garbage collected.

And that ends the decision cycle, so the next elaboration cycle can begin on the revised working memory:

context slots	*working memory*	*preference memory*
state=top-state:		
operator=exercise		[... >]
	time=noon	[... +]
	weekday=Friday	[... +]
	day-type=payday	[... +]
	agenda=lunch	[... +]
	[operator=exercise +]	[... +]
	[operator=eat-steak-at-Fred's +]	[... +]

Productions specifying "IF operator=exercise" will now fire, while those specifying "IF operator=eat-a-steak-at-Fred's" will not. The preferences for both operators are still in working memory. They'll stay there while the productions that put them in place continue to match working memory. (The > preference stays while the new chunk matches.) But it's the operator in slot that makes the difference. The next section explains why operators are especially important.

Operators and O-Support

As noted early in the introductory material, operators are a key concept. The Soar theory describes a cognitive architecture that deals with the world by applying operators to modify states. In fact, that's the S and O of Soar.

But the name "operator" as used in the Soar language and system is a little misleading. Unlike the human "operator" of a piece of machinery, a Soar operator doesn't actually do anything. It's just another item in memory, which might cause productions to fire. It is the productions that do the real work.

So, in the robot example, installing the "exercise" operator in the context slot won't, by itself, have any effect. The programmer will have to supply productions, such as:

```
IF operator=exercise
        THEN output=walk-around-office
IF time=1pm and operator=exercise
        THEN reconsider operator=exercise
```

From a programmer's point of view, however, a critical feature of operators isn't obvious in this example. Although operators can lead to output commands, they are more often used in subgoals where they just help control problem solving, without actually producing any action. The important programming fact about operators, especially when applied internally, is that they are "sticky." Equally important, things that go into working memory as a result of an operator being in the context slot are also "sticky."

To understand what's meant by "sticky" (it's a Soar-speak term, I didn't make it up), think about what will happen when the time of day changes to 12:01. As soon as that happens, the production that says:

```
IF time=noon THEN agenda=lunch
```

will no longer apply. As a result, the + preference that it put into preference memory will disappear. And with it, the working memory element, "agenda=lunch" will also be

removed. This is the start of a chain reaction that reaches right down to the point in the robot's behavior that we're looking at now: if the agenda isn't lunch, then productions proposing the eat-steak and the exercise operators no longer apply, so there's no tie to resolve, and... what should happen?

What happens is this: Because operators are sticky, the exercise operator stays in slot, even after the conditions that caused it to be placed there have changed. This gives the productions that actually "do" the operation time to fire. And it makes it the programmer's job to ensure that other productions test the state of the world and get the operator out of slot when it's no longer appropriate. That's what the "reconsider" production shown above does (reconsider is another preference, symbolized @).

Besides being sticky themselves, operators also pass some of their glue on to things placed in working memory as a result of the operation. If a production's IF side tests for the existence of a specific operator, then the items put into working memory as a result of that production will stay in memory after the conditions described on the IF side, including the operator being in slot, have changed.

For example: The production that sets the agenda to lunch if it's noon doesn't test any operator, so the agenda will be taken out of working memory as soon as it's no longer noon. (This is called, rather grandiosely, Soar's "truth maintenance system," or TMS.) But imagine that the production had been written this way:

```
IF time=noon and operator=set-agenda
    THEN agenda=lunch
```

In that case, a "set-agenda" operator would need to be in slot before the production would fire. But once "agenda=lunch" was placed in working memory, it would stay there, even after the time and operator had changed. until it was explicitly removed by some other production,

In review, then, the basic story is this: Things put into working memory by most productions will only stay in memory while the IF side of the production still matches. Those things are said to have "I-support" ("I" for "instantiation). But operators stay in slot even after the productions that proposed them are no longer valid. And things put into working memory by productions that test operators will stay until they're removed by some other production. Those working memory elements are said to have "O-support."

III. MORE ABOUT OPERATORS, PREFERENCES, AND SYNTAX

For anyone ready to actually write some Soar programs, here are some further details. All of this is in the *Soar User's Manual,* in much greater depth — but when I started working in Soar, I found that the manual actually gave too much information. So here are some highlights that I found useful.

Process Review

Here's a review of the process described in the previous section:

An Elaboration cycle:

- Soar I/O is done, getting input from other programs or interfaces, or sending information to them. (Input and output don't both happen exactly at the beginning of the cycle, but they are each performed within every elaboration cycle.)

- All productions that matched on the previous cycle but no longer match are withdrawn, which means the wme's they installed are withdrawn unless they had O-support. ("Wme's" are working-memory elements — things like "agenda=lunch.")
- All productions matching working memory fire, which means that preferences for the wme's on the productions' THEN sides are put into preference memory.
- The arbitration rules are applied to the preferences — except for the ^operator preferences — and the winning wme's are placed in working memory. If preferences don't resolve the contention (i.e., two acceptable agendas), then neither item is put into working memory.
- All acceptable preferences for operators are placed in working memory. That's acceptable preferences only — betters, worsts, etc. just stay in preference memory. No ties or other inconsistencies among operator preferences are resolved at this point.

Elaboration cycles repeat until no more productions fire. That's quiescence.

A Decision Phase (which happens as soon as elaboration cycles stop):

- The system starts at the top state and changes each "context" slot (state and operator) as required by new working memory elements:
  - operator reconsiders for the current operator
  - operator proposals for an open operator slot
  - impasses resolved that allow substates to be removed
- The system does this all the way down, looking at each state in the current stack of subgoals.
- If nothing changes, the system creates a new substate at the bottom of the stack, recognizing an impasse of one of these types:
  - state no-change (no operator to install)
  - operator tie (two or more equally preferred operators)
  - operator conflict (e.g. A > B and B > A)
  - operator constraint failure (e.g., A required and prohibited)
  - operator no-change (operator installed, nothing fires)

As soon as the decision phase completes, the system loops back into the elaboration-cycle phase.

Incidentally, notice that "conflict" describes a specific kind of impasse. It's not a generic term for any situation that could cause impasse. In Soar-speak, a tie impasse, like eat/exercise in the example, isn't a conflict. It's a tie.

Similarly, because Soar fires all matching productions (although their results initially go into preference memory) it is sometimes said that in Soar there "is no conflict-resolution." But the real point isn't that these things-that-would-be-called-conflicts don't get resolved. It's that they get resolved further downstream, using (for operators) all the relevant knowledge in the model, not just one simple set of rules.

Operators: Typical Use

Because they are the only way to make persistent changes to working memory, operators are the key to any Soar program. The typical programming scheme for using operators requires productions to do three things:

- propose the operator (IF condition-x THEN operator=foo)
- apply the operator (IF operator=foo THEN do things to memory)
- terminate the operator (IF not-x THEN reconsider operator=foo)

If you try to write a "simple" Soar program without using this scheme, you'll almost always run into problems with things being retracted from working memory before they should.

Here's how the three kinds of productions typically sequence:

1. A proposer with "stomach-state=hungry" on its IF side fires and puts [operator=eat +] into preference memory.

2. The acceptable preference for "eat" is moved into working memory.

3. Assuming no other operators have been proposed, the "eat" operator is installed in the operator context slot.

4. Now that "eat" is the operator, the operator-application productions can fire. They have "operator=eat" on their IF side and "stomach-state=hungry –" on their THEN side.

5. The preference-arbitration rules are applied to "stomach-state=hungry +", which was in memory to begin with, and "stomach-state=hungry –". The result is that "stomach-state=hungry" is removed from working memory. Now the production that proposed the "eat" operator is no longer applicable, so the + preference for that operator disappears.

6. A terminator production fires. It has the form, "IF stomach-state is not hungry THEN reconsider the eat operator."

7. At the next decision cycle, the eat operator is reconsidered, and since the preference that suggested it is gone, the operator is taken out of slot. Since "stomach-state=hungry –" was put into preference memory by a production that tested the operator, its effect will persist after the operator is gone, and the eat operator won't be proposed again.

Soar Language: Basic Syntax

Here's a typical production, written in the Soar language:

```
(sp suggest-lunch-agenda*friday          ; production name
   (state <s> ^day <d> ^agenda lunch)    ; IF side (LHS)
   (<d> ^name friday)                    ;
   -->
   (<s> ^operator <o>)                   ; THEN side (RHS)
   (<o> ^name eat-steak))                ;
```

The production's name is like a subroutine name: defined by convention for programmer readability, but not semantically meaningful to the system. Semicolons are comment

delimiters. LHS and RHS stand for left-hand side and right-hand side, which is the common terminology — not IF side and THEN side. This example doesn't show it, but items or entire lines on the LHS can be negated by putting a "–" in front of them. A negated item on the LHS means the production fires only if that line does not match current working memory.

See the *Soar User's Manual* for a full description of semantics.

Here, in Lisp-like syntax, is how you might imagine the knowledge after the production has fired:

```
(state   (day (name friday))
         (agenda lunch)
         (operator (name eat-steak))
)
```

In fact, the knowledge is represented in Soar something like this:

```
(S1 ^type state)
(S1 ^day D23)
(D23 ^name friday)
(S1 ^agenda lunch)
(S1 ^operator O7 +)
(O7 ^name eat-steak)
```

Preference Syntax: a Warning

A production with a RHS reading:

```
-->
(<s> ^agenda lunch))
```

puts an acceptable (+) preference into memory for agenda=lunch. That is, if you specify no preference, you get a +.

This RHS puts a best (>) preference into memory:

```
-->
(<s> ^agenda lunch >))
```

But — here's the warning — it does NOT put an acceptable preference in as well, and nothing, absolutely *nothing,* ever moves from preference memory to working memory unless it has an acceptable preference (or a require, which is almost never used).

So you will often need to write:

```
-->
(<s> ^agenda lunch + >))
```

This puts both preferences in memory. Beware that there are some tricky comma rules involving multiple preferences and multiple items — again, see the manual.

Accessing Higher States

When you're writing code that will apply in a subgoal, you often want to refer to items in higher states. There are two ways to do this. First, each state contains a pointer to the state above it, so you can always use that pointer, or a chain of those pointers:

```
(sp two-down*look-to-top
     (state <s> ^impasse tie ^attribute operator
                             ^superstate <ss>)
     (<ss> ^superstate <sss>)
     (<sss> ^time-at-the-top <t>)
     -->
     (<s> ^time-down-here <t>))
```

The other way to get access is to have two "root" calls on the IF side of the production:

```
(sp two-down*look-to-top
     (state <s1> ^impasse tie ^attribute operator)
     (state <s2> ^superstate nil ^time-at-the-top <t>)
     -->
     (<s> ^time-down-here <t>))
```

With both productions — indeed, with any production — it's critical that the "(state <sx> ^attribute-1 <val-1> ..." specifies enough unique attributes and values to distinguish the state you're looking for from any other state that might also be in working memory. The "^superstate nil" specification can only match the top state, but distinguishing substates may be trickier.

I-Support, O-Support, and the Details of Preference Persistence

Here's another review and more details on this critical topic. O-support is what preferences (and their associated working memory elements) have if they're installed by a production that tests something about an operator. I-support is what the preferences have if they're put in place by any other production.

Preferences and their wme's with I-support will drop out of working memory as soon as the production that suggested them no longer applies. That's the Truth Maintenance System. Those with O-support will persist until explicitly changed.

Now the further detail. The reject preference is special. If an O-supported reject preference is put into memory, it will cause the removal of *all* preferences for the attribute being un-preferred. As an example, first consider the I-supported case. Assume working memory holds the following (which could be either I- or O-supported):

working memory	*preference memory*
agenda=lunch	[agenda=lunch +]

Now an I-supported production adds:

[agenda=lunch –]

Then preference arbitration would result in:

working memory	*preference memory*
	[agenda=lunch +]
	[agenda=lunch –]

That is, the reject preference has caused the wme to be pulled out of working memory, but as long as the productions that produced the preferences still match working memory, both preferences will hang around.

But if the reject preference had been added by an O-supported production, then agenda=lunch would be pulled out of working memory, and both the + and the – preference would disappear as well:

working memory	*preference memory*

(Right, there's nothing there!) This gives the Soar system a chance to clean up preference memory, so things don't stay around long after they are meaningful.

Non-Operator Impasses: What <u>Really</u> Happens

When preference memory contains preferences for two non-operator items that can't be resolved by the standard preference arbitration rules, neither item will go into working memory. This is an "attribute impasse". And since the tied items aren't operators, Soar won't form a subgoal and try to resolve the impasse. (The logic is that only operators are important enough to justify subgoaling.)

But one other thing happens. An "impasse object" is put into working memory to flag the unresolved impasse. This object (working memory element) has the uncommon characteristic of having no direct connection to the current state, so locating it takes a bit of effort. However, a production of the form,

```
(sp monitor*impasse*agenda-attribute
     (impasse <i> ^attribute agenda)
     -->
     (write |oops - attribute impasse for agenda!|))
```

would fire on an impasse for the agenda attribute.

This kind of impasse is usually a programming error, which can be hard to find — your trace shows all the right productions firing, but then working memory doesn't contain what you expect it to contain. There's a general-purpose production in the default productions (described below, in the Environment section), which will warn you of attribute impasses that you hadn't expected.

IV. MORE ABOUT CHUNKING

The IF Side of Chunks

The process that decides what goes on the IF side of the chunk is called backtracing. The *Soar Users Manual* provides a lot of detail on how backtracing works. For the most part, its operation is just this: A chunk's IF side will test for the conditions that caused subgoaling (an operator tie, for example), and it will test for any other facts that are "touched" during the problem-solving that produces the chunk. In other words, if any production fires in the subgoal that leads to chunking, and if the results of that production are used in deciding what to return to the state above, then the tests from the IF side of that production will show up on the LHS of the chunk.

But there are some exceptions. One is that things created during the problem solving, then destroyed when the result is known, aren't tested in the chunk. So, if a Soar program is doing multiple-column addition, it might produce a chunk that said:

> IF adding 12 and 37 THEN the result is 49.

But it probably would not produce:

> IF adding 12 and 37 and the partial sums are 4 and 9
> THEN the result is 49.

Another tricky area to watch out for is negation. Chunks and productions can include negative conditions: IF it's not thursday THEN ... But the backtracing mechanism can only trace through productions that fired and produced things, not through productions that removed things and allowed negative conditions to hold.

A third area that's less-than-obvious involves variables versus constants. See the section on Knowledge Representation, below, for more on that.

The THEN Side of Chunks

In the office-worker example, the chunk coming out of the subgoal was used to decide between two operators. That's called a "control" chunk. In general, the different reasons for impasses will lead to different kinds of chunks being learned, where the "kind" of chunk is defined by it's THEN side. Here's a summary of common chunk types:

Cause of Impasse	Kind of Chunk Learned
Operator conflict or tie	Control
State no-change (no operator)	Operator proposal
Operator no-change (operator but no action)	Operator application or Operator termination or State elaboration

The last chunk, state elaboration, takes care of the situation where an operator is in slot and operator application chunks exist, but the facts needed to trigger those chunks haven't been recognized yet. For example, maybe the exercise operator is in slot, and there's a production that says, "If you can afford it, and the operator is exercise, then go to the

gym." What's needed is one more production, which an impasse and subgoaling could provide, saying: "If it's payday, you can afford to go to the gym."

Declarative versus Procedural Memories and Data Chunking

A feature that distinguishes Soar from most other cognitive architectures is that it has only one form of long-term memory: productions (including chunks). Soar's only form of declarative memory (memory for facts) is its working memory, which is expected to change as the world changes.

□

To store declarative facts in long-term memory, Soar has to incorporate them into chunks. For example, you might write a Soar program that hears the score of the Dodger's game, then learns that score. The chunk that should be formed would be something like:

> IF asked for the score of the Dodger's game
> THEN say it was 6 to 2.

The chunk wouldn't necessarily have to be in "ask-answer" format — it could be something like, IF I need to know the score THEN it was 6 to 2. But however it's phrased, there needs to be an IF side that contains the question and a THEN side that contains the answer.

This turns out to be very difficult to do in Soar, because of the way the chunking mechanism analyzes what was "touched" in solving problems within a subgoal. The chunks that get produced if you write what would appear to be a reasonable Soar model will usually have the form:

> IF asked what the Dodger's game score was
> and it was 6 to 2
> THEN say it was 6 to 2.

In other words, the chunks require the answer to be known before it can be recalled. Not cool.

There are solutions to this problem, but they're not pretty. One solution is to first learn the sort of circular, "recognition" chunk just shown. Then, when asked for the score, the model starts guessing (to itself): Was it 1 to 1? Was it 1 to 2? Was it ... Was it 6 to 2? At this point, if the programmer has coded things right, the chunk will fire. The model can recognize that it fired, and say to itself, "Yeah, that sounds right." Then it will report the answer. It only has to go through the guessing routine once, because it can learn a chunk in the correct form as a result.

Learning on/off, ^quiescence t, and Justifications

You can turn learning (chunking) on and off in Soar. (Type "help learn" at the Soar prompt to see the details.) The hard, theoretical line on this option is: Always program with learning on, because it's not Soar with learning off. The practical version of that line is: You'll probably want learning on eventually, so turn it on in the beginning, because programming with it off will just produce code that doesn't run when it's turned on.

In addition to the global switch for learning on/off, there are a couple of tricks that let you turn it on or off locally. One of these is the use of ^quiescence t. If you put ^quiescence t into the IF part of any production that fires in a subgoal, then chunks that would be learned as a result of the problem solving related to that production will not be learned.

Instead of “Built: chunk-x” the system will report “Built: justification-x”. A justification is just like a chunk, except it evaporates as soon as the reason for its formation goes away.

Why would you want to test ^quiescence t? Typically, it’s used in some deep sub-sub-substate, which the model only reaches as the result of a particular line of problem solving. It may be useful to know that an impasse has been reached in a particular problem-solving episode, but not appropriate to remember that this problem will always produce that impasse.

That’s pretty abstract. Here’s an example: The task is to recall the score of the Dodger’s game, but the model hasn’t heard the score yet. It tries several routes to recall, then impasses into a “give-up” space. It needs to give up now, but it shouldn’t chunk that result, because that would prevent it from trying again later, after it might have learned the score. If the programmer knows this would be the case, then a ^quiescence t flag is a reasonable approach.

V. KNOWLEDGE REPRESENTATION ISSUES

Soar provides a very simple structure and not much guidance for knowledge representation. However, different ways of representing the same knowledge can have significant effects on a model’s behavior. There are also some conventions that have been established in the Soar research community.

The Effect of Constants and Variables on Chunking

When Soar builds chunks, it tries to be smart about it. The IF side of a chunk doesn’t include things that (the Soar system thinks) don’t matter, and the THEN side includes exactly the action that does matter. In addition, the IF and THEN sides may be more general than the situation in which the chunk was learned.

For example, a Soar model might be in the situation where it was trying to turn on the blue light, and there were operators proposed for pressing either the red or the blue button. Soar could subgoal to resolve the tie, and as a result it might produce either of the following chunks:

Specific Chunk:
IF the objective is to turn on the blue light
and one operator is press the blue button
and one operator is press the red button
THEN press the blue button.

General Chunk:
IF the objective is to turn on the <x-color> light
and one operator is press the <x-color> button
and one operator is press the <not-x-color> button
THEN press the <x-color> button.

Which chunk is learned will depend on how the knowledge is represented in the top state and in the production that’s used in the subgoal. In general, if the production in the subgoal tests “red” and “blue” then the specific chunk will be learned. But if the knowledge is represented in a way that allows equality of color to be tested, without actually specifying the color itself, then a more general chunk can be learned.

Here's a production that might propose the operators in the top state. If there were more than one button-color defined on the state, then the production would fire once for each color, proposing an operator for each:

```
(sp topstate*propose-operators
     (state <s> ^superstate nil)
     (<s> ^color <c>)
     -->
     (<s> ^operator <o>)
     (<o> ^action press-button ^color <c>))
```

Here's a production that could apply in the subgoal to resolve the tie:

```
(sp substate*resolve-button-tie
     (state <s> ^superstate <ss> ^impasse tie
                ^attribute operator
                ^item <op1> ^item <op2>)
     (<ss> ^objective <ob>)
     (<ob> ^color <x>)
     (<op1> ^action press-button ^color <x>)
     (<op2> ^action press-button -^color <x>)
     -->
     (<ss> ^operator <op1> > <op2>))
```

Which chunk would these productions produce? That depends on how the world is represented in the top state. The following definition would produce a color-specific chunk, because the constants get "touched":

```
(sp topstate*describe-world
     (state <s> ^superstate nil)
     -->
     (<s> ^color red + & ^color blue + &)
     (<s> ^objective <ob>)
     (<ob> ^color blue))
```

But this definition would produce a color-general chunk, because the constants are one level removed from what the productions touch:

```
(sp topstate*describe-world
     (state <s> ^superstate nil)
     -->
     (<s> ^color <c1> + & ^color <c2> + &)
     (<c1> ^name red)
     (<c2> ^name blue)
     (<s> ^objective <ob>)
     (<ob> ^color <c2>))
```

Problem Spaces

The term "problem space" used a lot in Soar theory and programming. A problem space, loosely defined, is the knowledge needed to deal with a specific class of problems. A person might have a checkers problem space, or a take-a-long-trip problem space. There could be more detailed problem spaces within those general areas, such as a make-airline-

reservations problem space. This idea is important enough that earlier versions of the Soar system specifically chose a problem space, just like the system now chooses an operator.

Soar programmers usually organize productions into problem spaces within their source code, and they often name each problem space and put it on the IF side of the production. In the office-worker example, the main problem-space might be named "work-in-the-office-world." But when the issue of what to do at lunch came up, the programmer might have identified that as a "decide-between-lunch-options" problem space and written two productions to apply in the subgoal instead of one:

```
IF problem is that two operators are suggested
        and they are both lunchtime operators
THEN the problem-space is decide-between-lunch-options.

IF problem-space is decide-between-lunch-options
        and one option is exercise instead of eating
        and one option is eat a steak at Fred's Diner
THEN exercise is best
```

Besides making for clearer code, this scheme is also useful in situations where two or more problem spaces might be proposed — a decide-between-lunch-options space, a give-up-and-quite-the-job space, etc. If these have different preferences, then the preference arbitration rules can pick the right one. In other words, the problem spaces can represent strategies for solving a problem, and there can be an order among those strategies.

Even though problem spaces are a theoretically grounded concept, it's probably unreasonable to believe that people actually organize their knowledge into such totally segregated blocks. But it remain a useful programming convention.

Attributes for ^name and ^problem-space

The ^name attribute is used in various built-in or externally supplied trace routines for Soar. Operators, states, problem-spaces, and goals are often given names, and those will appear as the code is run with default trace settings. Outside of the trace functions, ^name has no special significance. Any working Soar model could have ^name replaced throughout with ^elephant, or vice versa, and it would run the same. (That's not the case for ^operator or ^state, obviously.)

The ^problem-space attribute has a similar status. In earlier versions of Soar (see "Pre-nnpscm Soar," below), ^problem-space was a reserved word, like ^operator and ^state. Code converted from earlier Soar will have ^problem-spaces, and most of the problem-spaces will have ^names. But in the current version of Soar, ^problem-space, while theoretically meaningful, is just another programmer-defined attribute name.

VI. PRODUCTION SCHEMAS

When you first look at someone else's Soar code, you'll probably see lots of productions without any obvious structure except for comments. There aren't any clear structural things like variable definitions or while-loops or subroutine calls. However, Soar productions do fall into categories, and it helps to have some idea of what those categories are.

Here are some of the main kinds of productions you're likely to see:

- state initialization
  - fires immediately after the state or substate is first created by the architecture, and often create a problem-space name.
- state elaboration
  - fires after the state has been initialized, adding I-supported elements to working memory that are "obvious" considering what's already there (i.e., IF noon THEN agenda=lunch)
- operator proposal
  - suggests an operator that would be appropriate to the situation.
- operator comparison, also called control
  - these produce better/best/worst or other desirability preferences, to help the decision phase select among proposed operators.
- operator application: external
  - with an operator in slot, this sends commands through the Soar I/O interface, to produce actions in the real world.
- operator application: internal
  - with an operator in slot, this produces O-supported state changes.
- operator termination
  - produces a reconsider preference for the current operator when its work is done, so it will go out of slot.
- return result (subgoal termination)
  - within a subgoal, pass some result up to a higher state, resolving the impasse and thereby eliminating the subgoal.
- task goal termination
  - with an operator in slot, this produces O-supported state changes.
- simulation
  - produce results that simulate the effect of changes in the real world. (these productions, which should be explicitly identified, aren't really part of the cognitive model, but it's often easier to use them than to write a totally separate simulation of the world in C.)

You may also see productions that combine two or more of these functions, such as initializing and elaborating a state, or initializing the space and proposing an operator.

VII. ENVIRONMENT

The Default Productions

The Soar system comes with a set of productions called the "Defaults." These provide a core set of very basic problem-solving routines that can do something semi-intelligent with each of the kinds of impasse that can arise in Soar. By "semi-intelligent," I mean they can do things like subgoal and pop out of goals in a way that's likely to uncover other productions which will resolve the impasse. Of course, if the programmer hasn't written those other productions, the system will just run out of places to look and stall. (Actually, even stalling is better than Soar's behavior without the defaults, which is to subgoal almost indefinitely.)

Some of the default routines go beyond simple impasse resolution to produce complex, AI-style search behavior. These defaults in particular, and the other defaults to a lesser extent, require the programmer to use specific conventions for knowledge representation in the rest of the program.

You can write Soar programs that run without using any of the default productions. This means you never have to load those productions into the Soar system, and you don't have to try to figure out what they're doing when they fire along with the productions you've written. But you will probably find yourself duplicating some of the simpler default productions.

SDE: Soar Development Environment

SDE is a gnu-emacs based environment that simplifies writing, running, and debugging Soar programs. Just a few of the useful things you can do with SDE are:

- view your Soar code in one window, Soar's standard output in another window, and various debugging printouts in a third window.
- mouse-click on a production or block of code in the editor and load it into Soar with a couple of keystrokes
- mouse-click on an item's id in the Soar trace and have the full description of that item printed (instead of typing, e.g., "p S1")
- mouse-click on an item in a trace and have the item's preferences printed (instead of typing, e.g., "preferences S1 agenda")
- mouse-click on a production name and disable that production (instead of typing, e.g., "excise top*lunch-operator*propose")

SDE does take some time to learn, especially if you're not already an emacs user. But if you're running Soar under Unix and you intend to write more than a trivial model, it's probably worth the effort.

Pre-nnpscm Soar

The examples and discussion in this document describe "nnpscm" Soar. (nnpscm stands for "new new problem-space computational model" — a bit of trivia that you might use to fill the next conversational lull...) In Soar systems that preceded the nnpscm version, there were context slots for Goals, Problem-Spaces, States, and Operators. Decisions for States, Problem-Spaces, and Operators were all handled very much like decisions for Operators are handled now. That is, if there was an impasse, then the system would produce a subgoal and attempt to resolve the impasse there.

If you ever need to read non-nnpscm code, keep in mind that the context slot called "State" in nnpscm was called "Goal" in non-nnpscm. That meant that subgoaling created a subgoal, instead of a substate... which makes a lot of sense, when you think about it.

Another feature of SDE is its ability to convert pre-nnpscm code to nnpscm.

ACKNOWLEDGEMENTS

If this material makes sense to the reader, it's largely because Richard Young did an excellent job of explaining the concepts to me. The Soar Tutorial presented by Frank Ritter and Richard Young at the Euro-Soar workshop in Leiden was also a big help in getting started, and Mike Hucka's support for SDE provided a powerful tool for investigating the hidden parts of Soar. Any mistakes and misconceptions are, of course, entirely my responsibility.

BIBLIOGRAPHY

Laird, J.E., Congdon, C.B., Altmann, E., and Doorenbos, R. (1993) *Soar User's Manual* (Version 6). (See on-line information, below, for sources.)

Laird, J., Newell, A., and Rosenbloom, P. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33, 1-64.

Newell, A. (1990). *Unified Theories of Cognition*. The William James Lectures. Cambridge, MA: Harvard University Press.

Rosenbloom, P.S., Laird, J.E., Newell, A. (1991) A preliminary analysis of the SOAR architecture as a basis for general intelligence. *Artificial Intelligence, 47,* 289-326.

Rosenbloom, P.S., Laird, J.E., Newell, A. (1933). *The Soar Papers: Readings on Integrated Intelligence*. MIT Press.

Soar on-line information:

On the world-wide-web:
http://www.isi.edu/soar/soar.html
http://www.isi.edu/soar/soar-archive-homepage.html/

Anonymous ftp to:
centro.soar.cs.cmu.edu, in directory /afs/cs/project/soar/public/
(this is where to get updated Soar and SDE code)

EXERCISES

These are really simple. But, hey — anybody can suggest hard Soar exercises! ("Write a Soar model of a fighter pilot," for example.)

1. Run the office-worker code. Watch preference memory and working memory to see how things happen. Here are some suggestions as to how:

- Type "r 1" (no quotes) at the Soar prompt to run 1 elaboration cycle.
- Type "preferences s1 time" to look at preferences for ^time.
- Type "wm s1" and "p s1" to see different representations of top-state working memory. Look for the ^action of the operators, after they're proposed.

- Type “ms” to see which productions are about to fire. Type “matches sub*lunch-operator-tie*resolve” to see what parts of that production match or don’t match working memory.

Notice that you can’t see things sitting in preference memory before they go into working memory, because “r 1” does an elaboration cycle complete with preference arbitration and memory update.

2. Add another production, just like top*lunch-agenda*set, except have it set the agenda to “work”. (If you cut and paste, be sure to change the production name — in fact, don’t change the name at first, and see what happens...)

Now step through the model again (type “init-soar” and “excise-chunks” first). What happens when the two different agendas are proposed? Can you find the preferences? Can you find the impasse object? (Try using “wm 1”, “wm 2”, ... to look at everything in working memory.)

3. Investigate what happens if there’s only one operator proposal.

4. Are you getting tired of typing “init-soar” and “excise-chunks”? Put a line at the top of the source code that says: (excise-task). That will wipe out the old model every time you reload the source code.

5. Investigate what happens if there are no operator proposals.

6. Go back to the original model and write productions to apply and terminate the exercise operator. Syntax for reconsider is:

```
(<s> ^operator <o> @)
```

where <o> has already been identified in the IF side of the production. But watch out: just reconsidering the operator won’t be enough. You’ll have to make sure the operator proposer no longer matches.

Further hint: remember that you can’t just write over things already in memory; you have to put a reject preference on what’s there, then put in an acceptable preference for something new. So the THEN side of one of your productions might be (depending on what you decide to change):

```
(<s> ^time noon -)
(<s> ^time 1pm)) ; “+” after 1pm is implicit
```

7. Here’s an example of a trick that’s often useful. Write a production that proposes [number=1 + =] and [number=2 + =] and ... up to [number=9 + =]. The “=“ is an “indifferent” preference, so the Soar system will pick one value for number and put it in working memory. Now initialize a subtotal=0 value in working memory and use an operator that does this:

```
IF      number=<x>
        subtotal=<y>
THEN
        subtotal=<y> –
        subtotal=(sum-of <y> and <x>)
        number=<x> –
```

This will add each number to the subtotal and reject the number, which kicks it out of working memory. But as soon as the number is gone, another of the indifferently preferred numbers will go into working memory, and the operator will get applied again.

Try it. Get it to kick the operator out of slot and output the answer.

8. You get the idea. Now start on the fighter-pilot model.

SAMPLE CODE

```
;;; The office-worker robot -- lunchtime knowledge
;;; John Rieman, 15 March 95

;;; For Soar 6, nnpscm. (tested under 6.3.5)

;;; -------------------------------------------------
;;; the topstate knowledge

(sp top*office-world*init
    (state <s> ^superstate nil)
    -->
    (<s> ^time noon)
    (<s> ^weekday friday)
    (<s> ^daytype payday))

(sp top*lunch-agenda*set
    (state <s> ^time noon)
    -->
    (<s> ^agenda lunch))

(sp top*eat-steak-operator*propose
    (state <s> ^agenda lunch ^daytype payday)
    -->
    (<s> ^operator <o>)
    (<o> ^action eat ^food steak ^place Freds))

(sp top*exercise-operator*propose
    (state <s> ^agenda lunch ^weekday friday)
    -->
    (<s> ^operator <o>)
    (<o> ^action exercise))

;;; not coded: productions to apply and terminate the
;;; exercise operator.

;;; -------------------------------------------------
;;; here's knowledge for the substate

(sp sub*lunch-operator-tie*resolve
    (state <s> ^superstate <ss> ^impasse tie
          ^attribute operator ^item <o1> ^item <o2>)
    (<o1> ^action exercise)
    (<o2> ^action eat ^food <f> ^place <p>)
    -->
    (<ss> ^operator <o1> >))

;;; -------------------------------------------------
;;; Here's a trace, followed by the chunk learned

#|

Soar> load "office-worker.soar"
```

```
Loading office-worker.soar
*****

Soar> watch :firings on

Soar> d 3

     0: ==>S: S1
Firing top*office-world*init

Firing top*lunch-agenda*set

Firing top*eat-steak-operator*propose

Firing top*exercise-operator*propose
     1:    ==>S: S2 (operator tie)
Firing sub*lunch-operator-tie*resolve

Build: chunk-1
     2:    O: O2
     3:    ==>S: S3 (operator no-change)

Soar> list-chunks
(sp chunk-1
  :chunk
  (state <s1> ^operator <o2> + ^operator <o1> +)
  (<o2> ^action exercise)
  (<o1> ^place freds ^food steak ^action eat)
-->
  (<s1> ^operator <o2> >))


Soar>

|#
```