# Part VIII: Semantic Memory

Semantic memory (SMem) in Soar is a mechanism that allows agents to deliberately store and retrieve objects that are persistent; this information supplements what is contained in short-term working memory (as well as rules, episodic memory, etc).

## 1. The Semantic Store

Before we delve into how an agent can use semantic memory, let’s see an example of preloading knowledge and viewing the contents of the memory.

First, open the Soar Debugger. Then, execute the following command (this can be loaded from a source file just as any other Soar command):

smem --add {

(<a> ^name alice ^friend <b>)

(<b> ^name bob ^friend <a>)

(<c> ^name charley)

}

After executing this command, three identifiers (represented above as <a>, <b>, and <c>) will be added to semantic memory, as well as the associated name and friend augmentations. This command is more generally useful to preload the contents of large knowledge bases in Soar.

We can view the contents of semantic memory using the following command:

smem --print

Which will output the following result:

(@A1 ^friend @B1 ^name alice [+1.000])

(@B1 ^friend @A1 ^name bob [+2.000])

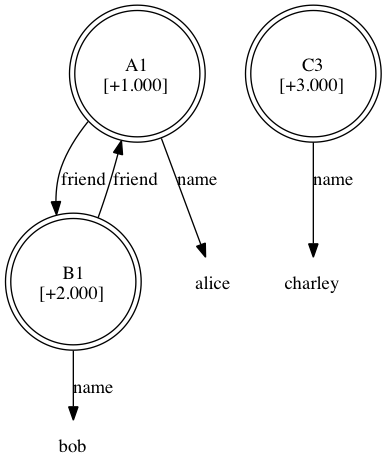
(@C3 ^name charley [+3.000])

Note first that the variables from the *smem --add* command have been instantiated as specific identifiers (<a> as @A1, <b> as @B1, and <c> as @C3). Additionally, the identifiers have the at sign (@) as a prefix. All identifiers in semantic memory are persistent, and thus we call them *long-term identifiers* (or LTIs). When printed, long-term identifiers are prefixed by the @ symbol and, when depicted, are shown using a double circle. The number in square brackets is the bias value of the object, which we shall return to when discussing retrievals. Finally, unlike working memory and rules, the knowledge in semantic memory need not be connected, nor linked directly or indirectly, to a state.

To pictorially view the contents of semantic memory, we combine the *command-to-file* command, which takes the output of any command in Soar and redirects it to a file, with the *smem --viz* command, which outputs the contents of semantic memory in Graphviz format. For example, execute the following command:

command-to-file smem.gv smem --viz

The result will be a new file in Soar’s current working directory (get this by executing the *pwd* command) named smem.gv. You can open this in any text editor to see the source code, but more usefully, open it with any Graphviz renderer (see <http://graphviz.org> for more detail) to produce the following diagram:

Now that we have seen the contents of semantic memory, you can confirm that none of this knowledge is present in any of Soar’s other memories. For instance, execute the following commands to print the contents of Soar’s working and procedural memories:

print --depth 100 <s>

print

You notice that the contents of the semantic store can be completely independent of the other memories, though, as discussed later, an agent can access and modify the store over time.

We are now done with this example and wish to clear the semantic store. However, long-term identifiers are persistent and can permeate other memories (such as through chunking). Therefore, in order to clear the store, we also need to clear all other memories. To do this we issue a special command:

smem --init

The agent is now reinitialized, as you can verify by printing the contents of working memory, procedural memory, and now semantic memory.

## 2. Agent Interaction

Agents interact with semantic memory via special structures in working memory. Soar automatically creates an *smem* link on each state, and each *smem* link has specialized substructure: a *command* link for agent-initiated actions and a *result* link for feedback from semantic memory. For instance, issue the following command:

print --depth 10 <s>

If you read the output carefully you will notice a WME that can be generally represented as (state ^smem <smem>) and two additional WMEs that can be represented as (<smem> ^command <cmd>) and (<smem> ^result <r>).

As described in the following sections, the agent, via rules, populates and maintains the *command* link and the architecture populates and cleans up the *result* link.

For the agent to interact with semantic memory, this mechanism must be enabled. By default, all learning mechanisms in Soar are disabled. To enable semantic memory, issue the following command:

smem --set learning on

## 3. Agent Storage and Modification

An agent stores an object to semantic memory by issuing a *store* command. The syntax of a store command is (<cmd> ^store <id>) where <cmd> is the *command* link of a state and <id> is an identifier.

An agent can issue multiple store commands simultaneously, and the commands are processed at the end of the phase in which they are issued. A *store* command is guaranteed to succeed and the response from the architecture will be a success WME: (<r> ^success <id>), where <r> is the *result* link of the state on which the *store* command was issued and <id> was the value of the *store* command.

A *store* command stores the identifier that is the result of the command, as well as any augmentations of that identifier. The command is not recursive. If the identifier to be stored was not long-term, it is changed in place to a long-term identifier. If it was already in semantic memory, the augmentations of the long-term identifier in semantic memory are overridden.

Let’s see an example. Source the following rules into the Soar Debugger.

sp {propose\*init

(state <s> ^superstate nil

-^name)

-->

(<s> ^operator <op> +)

(<op> ^name init)

}

sp {apply\*init

(state <s> ^operator.name init

^smem.command <cmd>)

-->

(<s> ^name friends)

(<cmd> ^store <a> <b> <c>)

(<a> ^name alice ^friend <b>)

(<b> ^name bob ^friend <a>)

(<c> ^name charley)

}

sp {propose\*mod

(state <s> ^name friends

^smem.command <cmd>)

(<cmd> ^store <a> <b> <c>)

(<a> ^name alice)

(<b> ^name bob)

(<c> ^name charley)

-->

(<s> ^operator <op> +)

(<op> ^name mod)

}

sp {apply\*mod

(state <s> ^operator.name mod

^smem.command <cmd>)

(<cmd> ^store <a> <b> <c>)

(<a> ^name alice)

(<b> ^name bob)

(<c> ^name charley)

-->

(<a> ^name alice -)

(<a> ^name anna

^friend <c>)

(<cmd> ^store <b> -)

(<cmd> ^store <c> -)

}

Now click the “Step” button to run till the decision phase and notice that the *init* operator is selected. Now, click the “Watch 5” button and then the “Run 1 -p” button to watch as the operator is applied. Below is part of the trace that should be produced. If you do not see this part of this trace in your run, be sure that you enabled semantic memory (see section above).

--- apply phase ---

--- Firing Productions (PE) For State At Depth 1 ---

Firing apply\*init

-->

(C3 ^name charley + :O)

(B1 ^friend A1 + :O)

(B1 ^name bob + :O)

(A1 ^friend B1 + :O)

(A1 ^name alice + :O)

(C2 ^store C3 + :O)

(C2 ^store B1 + :O)

(C2 ^store A1 + :O)

(S1 ^name friends + :O)

--- Change Working Memory (PE) ---

=>WM: (25: C3 ^name charley)

=>WM: (24: B1 ^friend A1)

=>WM: (23: B1 ^name bob)

=>WM: (22: A1 ^friend B1)

=>WM: (21: A1 ^name alice)

=>WM: (20: C2 ^store A1)

=>WM: (19: C2 ^store B1)

=>WM: (18: C2 ^store C3)

=>WM: (17: S1 ^name friends)

--- Change Working Memory (PE) ---

=>WM: (28: R3 ^success @A1)

=>WM: (27: R3 ^success @B1)

=>WM: (26: R3 ^success @C3)

Notice that the *apply\*init* rule fired and added 3 *store* commands to working memory, where the identifiers to be stored are, initially, not long-term (and whose augmentations mirror the contents of the *smem --add* command in part 1 of this tutorial). Then, at the end of the elaboration phase, semantic memory processed the command, converted the identifiers to long-term, and added status for each command.

Now, try printing the contents of semantic memory using the *smem --print* command. You will see that semantic memory now has the same contents as after using the *smem --add* command in part 1.

Application of the next operator modifies the contents of semantic memory by overriding the contents of an existing long-term identifier (@A1). Click the “Step” button to select the next operator (*mod*) and then click the “Run 1 -p" button to apply the operator:

Firing apply\*mod

-->

(C2 ^store @C3 - :O)

(C2 ^store @B1 - :O)

(@A1 ^friend @C3 + :O)

(@A1 ^name anna + :O)

(@A1 ^name alice - :O)

--- Change Working Memory (PE) ---

=>WM: (33: @A1 ^name anna)

=>WM: (32: @A1 ^friend @C3)

<=WM: (21: @A1 ^name alice)

<=WM: (18: C2 ^store @C3)

<=WM: (19: C2 ^store @B1)

--- Change Working Memory (PE) ---

<=WM: (26: R3 ^success @C3)

<=WM: (27: R3 ^success @B1)

You will notice in the trace that the store commands for @B1 and @C3 are removed by the application rule, and that augmentations of @A1 are removed and added. Then, at the end of the elaboration phase, semantic memory cleans up the status information for the old *store* commands.

Now, print the contents of semantic memory using the *smem --print* command:

(@A1 ^friend @B1 @C3 ^name anna [+4.000])

(@B1 ^friend @A1 ^name bob [+2.000])

(@C3 ^name charley [+3.000])

Notice that the augmentations of @A1 have indeed changed in semantic memory to reflect the new *store* command, while @B1 and @C3 remain unchanged.

## 4. Non-Cue-Based Retrieval

The first way an agent can retrieve knowledge from semantic memory is called a non-cue-based retrieval: the agent requests from semantic memory all of the augmentations of a known long-term identifier. The syntax of the command is (<cmd> ^retrieve <lti>) where <lti> is a long-term identifier.

As an example, add the following two rules to our agent from part 3 of this tutorial:

sp {propose\*ncb

(state <s> ^name friends

^smem.command <cmd>)

(<cmd> ^store <a>)

(<a> ^name anna

^friend <f>)

-->

(<s> ^operator <op> + =)

(<op> ^name ncb

^friend <f>)

}

sp {apply\*ncb

(state <s> ^operator <op>

^smem.command <cmd>)

(<op> ^name ncb

^friend <f>)

(<cmd> ^store <a>)

(<f> ^name <f-name>)

-->

(<cmd> ^store <a> -

^retrieve <f>)

(<f> ^name <f-name> -)

}

These rules propose to retrieve all the information about one of @A1’s two friends (selected randomly), and removes the friend’s name from working memory.

Unlike *store* commands, all retrievals are processed during the agent’s output phase and only one retrieval can be requested per state per decision.

So now click the “Step” button and notice that one of the two *ncb* operators is selected. Click “Run 1 -p" to see the application rule create a *retrieve* command, requesting information about one of the two friends, as well as remove that friend’s name from working memory. Then click the “Run 1 -p" button again to proceed through the output phase. Finally print the contents of the *smem* link (*print --depth 10 s2*):

(S2 ^command C2 ^result R3)

(C2 ^retrieve @C3)

(@C3 ^name charley)

(R3 ^retrieved @C3 ^success @C3)

We see that semantic memory has retrieved and added to working memory the name of the friend, as well as indicated status for this command (*success*). Note that had the *retrieve* command been issued with an identifier that was not long-term, the status would have been *failure* and there would be no *retrieved* structure. Note also that retrieved knowledge is limited to the augmentations of the long-term identifier and so, like the *store* command, the *retrieve* command is not recursive.