# Part VIII: Semantic Memory

Semantic memory (SMem) in Soar is a mechanism that allows agents to deliberately store and retrieve facts about 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 objects (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])

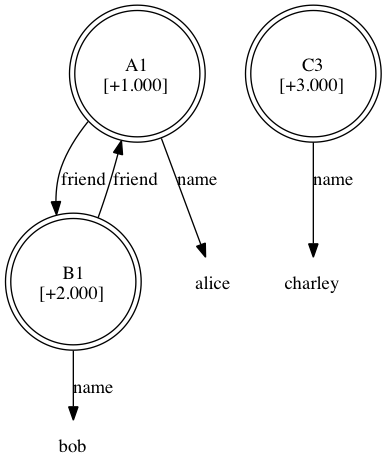
(@C3 ^name charley [+3.000])

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 we thus call them *long-term identifiers* (or LTIs). When printed, long-term identifiers are prefixed by the @ symbol and, when depicted, we use 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 visualizer (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, using rules, populates and maintains the *command* link and the architecture populates and cleans up the *result* link.

## 3. Agent Storage and Modification