4_running_our_first_model

April 2, 2021

Open in Colab

1 Running our first model

Code from the previous chapter:

[1]:
```
# uncomment the line below to install pyactr
# !pip3 install pyactr

import pyactr as actr
```

[2]:
```
actr.chunktype("word", "form, meaning, category, number, synfunction")
actr.chunktype("goal_lexeme", "task, category, number")

carLexeme = actr.chunkstring(string="""
    isa word
    form car
    meaning '[[car]]'
    category noun
    number sg
    synfunction subject
""")

agreement = actr.ACTRModel()
dm = agreement.decmem
dm.add(carLexeme)
```

[3]:
```
agreement.productionstring(name="retrieve", string="""
    =g>
    isa goal_lexeme
    category verb
    task agree
    ?retrieval>
    buffer empty
    ==>
    =g>
    isa goal_lexeme
    task trigger_agreement
    category verb
    +retrieval>
    isa word
    category noun
    synfunction subject
""")

agreement.productionstring(name="agree", string="""
    =g>
    isa goal_lexeme
    task trigger_agreement
    category verb
    =retrieval>
    isa word
    category noun
    synfunction subject
    number =x
    ==>
    =g>
    isa goal_lexeme
    category verb
    number =x
    task done
""")

agreement.productionstring(name="done", string="""
    =g>
    isa goal_lexeme
    task done
    ==>
    ~g>
""")
```

[3]:
```
{'=g': goal_lexeme(category= , number= , task= done)}
==>
{'~g': None}
```

To run the agreement model, we just have to add an appropriate chunk to the goal buffer. Recall that ACT-R conceptualizes higher cognition as fundamentally goal-driven: if there is no goal, no productions will fire and the mind will not change state.

We add a goal chunk below. First, we declare our `goal_lexeme` type (line 1). Then, we add one such chunk to the goal buffer (lines 2-6). Chunks are always added to buffers / modules using the method `add`. We check that the chunk has been added to the goal buffer by printing its contents (line 7). Note that the number specification on line 8 is empty.

[4]:
```
agreement.goal.add(actr.chunkstring(string="""
    isa goal_lexeme
    task agree
    category verb
    """))
agreement.goal
```

[4]:
```
{goal_lexeme(category= verb, number= , task= agree)}
```

We can now run the model by invoking the `simulation` method (with no arguments), as shown in line 1 below. This takes the model specification and initializes various parameters as dictated by the model (e.g., simulation start time). We can then execute one run of the simulation, as shown on line 2.

[5]:
```
agreement_sim = agreement.simulation()
agreement_sim.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: retrieve')
(0.05, 'PROCEDURAL', 'RULE FIRED: retrieve')
(0.05, 'g', 'MODIFIED')
(0.05, 'retrieval', 'START RETRIEVAL')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'retrieval', 'CLEARED')
(0.1, 'retrieval', 'RETRIEVED: word(category= noun, form= car, meaning= [[car]],
number= sg, synfunction= subject)')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: agree')
(0.15, 'PROCEDURAL', 'RULE FIRED: agree')
(0.15, 'g', 'MODIFIED')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'RULE SELECTED: done')
(0.2, 'PROCEDURAL', 'RULE FIRED: done')
(0.2, 'g', 'CLEARED')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'NO RULE FOUND')
```

The output of the `run()` command is the temporal trace of our model simulation. Each line specifies three elements:

1. the simulation time (in seconds);
2. the module (name in upper-case letters) or buffer (name in lower-case letters) that is affected;
3. a description of what is happening to the module or buffer. By default, every cognitive step in the model takes 50 ms, i.e., 0.05 seconds; this is the ACT-R default time for an elementary cognitive operation.

The first line of our temporal trace states that conflict resolution is taking place in the procedural memory module, i.e., the module where all the production rules reside. This happens at simulation time 0. The main function of 'conflict resolution' is to examine the current state of the mind (basically, the state of the buffers in our model) and to determine if any production rule can apply, i.e., to check if the current state of the mind satisfies the preconditions of any production rule.

Note how ACT-R once again combines serial and parallel components to capture actual cognitive behavior. Checking if the current state of the mind satisfies the preconditions of any rule is a massively parallel process: all rules are simultaneously and very quickly (instantaneously) checked. But rule firing is serial: at any given point in the cognitive process, only one rule can fire / apply. This is similar to the interaction between the parallel computations in the declarative memory module (all chunks are simultaneously checked against a pattern / cue) and the serial way in which retrieval cues can be placed in the retrieval buffer (one at a time).

'Conflict resolution' is particularly simple in the present case. Given the state of the goal and retrieval buffers, only one rule can apply: our first production rule, which we named `retrieve` above. Line 4 in our temporal trace shows that the `retrieve` rule is selected at time 0. The rule fires, and this takes the ACT-R default time of 50 ms, as shown on line 5. The state of our mind has changed as a consequence of this rule firing, and the subsequent lines in the output report on that new state: the goal buffer has been modified (line 6; the task is now `trigger_agreement`) and the retrieval buffer has started a memory retrieval procedure (line 7), which will take time to complete.

Now that the `retrieve` rule has fired, the procedural module enters a 'conflict resolution' state again and looks for production rules to apply (line 8). The current state of the mind (i.e., the buffer state) does not satisfy the preconditions of any rule, so none is fired (line 9).

However, a memory retrieval process has been started and is completed 50 ms later, i.e., at the next simulation time of 100 ms. Retrieval time is set to a default value of 50 ms here, but ACT-R specifies in great detail how memory behaves, and makes clear predictions about retrieval accuracy and retrieval latency. This is discussed in detail in Chapter 6, but we want to keep our first model simple so we use the default retrieval time of 50 ms here.

At the 100 ms point, the memory retrieval process has been completed and the retrieval buffer is cleared (line 10) so that the newly retrieved chunk can be placed there (lines 11-12).

The mind is now in a new state since the buffer contents have changed, so the procedural module reenters a 'conflict resolution' state of rule collection & rule selection (line 13). This time, the resolution process identifies one rule that can fire (line 14), namely the second production rule we discussed above and which we named `agree`.

The `agree` rule takes 50 ms to fire (line 15), so we are now at 150 ms in simulation time. As a consequence of the `agree` rule, the chunk in the goal buffer has been modified (line 16): its number specification has been updated so that it is now the same number as the noun chunk in the retrieval buffer.

Agreement has been performed, so the third and final production rule is selected (lines 17-18). The rule takes 50 ms to fire (line 19), so at time 0.2 s, the goal buffer is cleared (line 20), and no further rule can apply (lines 21-22).

When the goal buffer is cleared, the information stored in it does not disappear. The ACT-R architecture specifies that the cleared information is automatically transferred ('harvested') to declarative memory. The intuition behind this is that our past accomplished goals, i.e., the results of our past successful cognitive processes, become our present (newly acquired) memory facts. This is also the case in pyactr. We can inspect the final state of the declarative memory module to see that it stores the cleared goal-buffer chunk:

```
[6]: dm
```

```
[6]: {word(category= noun, form= car, meaning= [[car]], number= sg, synfunction=
     subject): array([0.]), goal_lexeme(category= verb, number= sg, task= done):
     array([0.2])}
```

Note that this newly added chunk is time-stamped with the simulation time at which the goal buffer was cleared (0.2 s).

And that's it. At its core, ACT-R provides a fairly simple framework for building process models that is accessible to generative linguists because it is production-rule based and manipulates feature structures of a familiar kind.

To be sure, our first model and the introduction to ACT-R and pyactr in this chapter are overly simplistic in many ways. But the main point is that we can now start building explicit and more realistic computational models for linguistic processes and behaviors.

Our development of integrated competence-performance theories for linguistic phenomena is now at a stage similar to the one in a formal semantics intro course where the semantics for classical first order logic (FOL) has just been introduced. FOL semantics is in many ways an overly simplistic model for natural language semantics, but it provides the basic structure that more realistic theories of natural language interpretation (in the Montagovian tradition) can build on.

```
[ ]:
```