3_modules_and_buffers_and_writing_productions_in_pyactr

April 2, 2021

1 Modules and buffers, and writing productions in `pyactr`

We recap the code from the previous notebook:

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

import pyactr as actr
```

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

# carLexeme = actr.makechunk(nameofchunk="car",
#                            typename="word",
#                            form="car",
#                            meaning="[[car]]",
#                            category="noun",
#                            number="sg",
#                            synfunction="subject")

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

1.1 Modules and buffers

Chunks do not live in a vacuum, they are always part of an ACT-R mind (a specific instantiation of the ACT-R mental architecture). The ACT-R building blocks for the human mind are modules and buffers. Each module in ACT-R serves a different mental function. But these modules cannot be accessed or updated directly. Input/output operations associated with a module are always

mediated by a buffer, and each module comes equipped with one such buffer. Think of it as the input/output interface for that mental module.

A buffer has a limited throughput capacity: at any given time, it can carry only one chunk. For example, the declarative memory module can only be accessed via the retrieval buffer. Internally, the declarative memory module supports massively parallel processes: basically all chunks can be simultaneously checked against a cue. But externally, the module can only be accessed serially by placing one cue at a time in its associated retrieval buffer. This is a typical example of how the ACT-R architecture captures actual cognitive behavior by combining serial and parallel components in specific ways (cf. **Anderson, John R., and Christian Lebiere. 1998.** ***The atomic components of thought*****. Hillsdale, NJ: Lawrence Erlbaum Associates**).

ACT-R conceptualizes the human mind as a system of modules and associated buffers, within and across which chunks are stored and transacted. This flow of information is driven by productions: ACT-R is a production-system based cognitive architecture. Recall that productions are stored in procedural memory, while chunks are stored in declarative memory. The architecture is more complex than that, but in this chapter we will be concerned with only these two major components of the ACT-R architecture for the human mind: procedural memory and declarative memory.

As we already mentioned, procedural memory stores productions. Procedural memory is technically speaking a module, but it is the core module for human cognition, so it does not have to be explicitly declared because it is always assumed to be part of any mind (any instantiation of the mental architecture). The buffer associated with the procedural module is the goal buffer. This reflects the ACT-R view of *human higher cognition as fundamentally goal-driven*. Similarly, declarative memory is a module, and it stores chunks. The buffer associated with the declarative memory module is called the retrieval buffer.

So let us now move beyond just storing arbitrary chunks, and start building a mind. The first thing we need to do is to create a container for the mind, which in `pyactr` terminology is a model:

[3]:
```
agreement = actr.ACTRModel()
```

The mind we intend to build is very simple. It is merely supposed to check for number agreement between the main verb and the subject of a sentence, hence the name of our ACT-R model above. We can now start fleshing out the anatomy and physiology of this very simple agreeing mind. That is, we will add information about modules, buffers, chunks and productions.

As mentioned above, any ACT-R model has a procedural memory module, but for convenience it also comes equipped by default with a declarative memory module and the goal and retrieval buffers. When initialized, these buffers/modules are empty. We can check that the declarative memory module is empty, for example:

[4]:
```
agreement.decmem
```

[4]:
```
{}
```

Note that `decmem` is an attribute of our `agreement` ACT-R model, and it stores the declarative memory module. The `retrieval` and `goal` attributes store the retrieval and the goal buffer, respectively, and they are also empty, as shown below.

[5]:
```
agreement.goal
```

[5]:
```
set()
```

[6]:
```
agreement.retrieval
```

[6]:
```
set()
```

It is convenient to have a shorter alias for the declarative memory module, so we introduce a new variable `dm` and assign the `decmem` module as its value:

[7]:
```
dm = agreement.decmem
```

We might want to add a chunk to our declarative memory, e.g., our `carLexeme` chunk. We add chunks by invoking the `add` method associated with the declarative memory module. The argument of this function call is the chunk that should be added:

[8]:
```
dm.add(carLexeme)
print(dm)
```

```
{word(category= noun, form= car, meaning= [[car]], number= sg, synfunction=
subject): array([0.])}
```

Note that when we inspect `dm`, we can see the chunk we just added. The chunk-encoding time is also recorded. This is the simulation time at which the chunk was added to declarative memory. We have not yet run the model, i.e., we have not yet started the model simulation, so that time is 0.

1.2 Writing productions in `pyactr`

Recall that productions are essentially conditionals (*if*-statements), with the preconditions that need to be satisfied listed in the antecedent of the conditional and the actions that are triggered if the preconditions are satisfied listed in the consequent. Thus, productions have two parts: the preconditions that precede the double arrow (==>) and the actions that follow it.

Let's add some productions to our model to simulate a basic form of verb agreement. Our model of subject-verb agreement will be very elementary, but the point is to learn how to assemble a basic ACT-R model / mind rather than to build a realistic processing model of this linguistic phenomenon. We restrict ourselves to agreement in number for 3rd person present tense verbs. We make no attempt to model syntactic parsing, we will just assume that our declarative memory already stores the subject of the clause, and that the current verb is already present in the goal buffer, where it is being actively assembled.

What should our agreement model do? One production should state that if the goal buffer has a chunk of category `verb` in it and the current task is to agree, then the subject should be retrieved. The second production should state that if the number specification on the subject in the retrieval buffer is `=x`, then the number of the verb in the goal buffer should also be `=x` (recall that the = sign before a string indicates that the string is the name of a variable). The third rule should say that if the verb is assigned a number, the task is done.

Let's start with the first production: noun retrieval. As shown below, line 1 below, we give the production a descriptive name `"retrieve"` that will make the simulation output more readable. In general, productions are created by the method `productionstring` associated with our ACT-R model, and they have two arguments (there is actually a third argument; more on that later): `name`, the name of the production, and `string`, which provides the actual content of the production.

[9]:
```
actr.chunktype("goal_lexeme", "category, number, task")

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
""")
```

[9]:
```
{'=g': goal_lexeme(category= verb, number= , task= agree), '?retrieval':
{'buffer': 'empty'}}
==>
{'=g': goal_lexeme(category= verb, number= , task= trigger_agreement),
'+retrieval': word(category= noun, form= , meaning= , number= , synfunction=
subject)}
```

The preconditions (the left hand side of the rule) and the actions (the right hand side of the rule) are separated by ==>. This separator can be seen on line 8 above above. Everything that precedes the separator belongs to the preconditions, and everything that follows it belongs to the actions. The rule has preconditions for two buffers. The first one starts on line 2. `=g>` indicates two things: the target buffer and the type of precondition this buffer has to satisfy. The precondition checks that the chunk currently stored in the *goal* buffer `g` is subsumed by the chunk that is specified on the following lines (lines 3-5). The `=` symbol encodes that we are interested in the subsume relation. That is, the chunk in the goal buffer has to be of category `verb` (line 4), and the current task for this lexeme should be `agree` (line 5). The chunk in the goal buffer could have other slot-value pairs, but we are not interested in them for the purposes of this rule.

The second precondition starts on line 6 above. `?retrieval>` indicates that this precondition will check whether the `retrieval` buffer is in a certain state. `?` in front of the buffer name indicates that we are interested in the state of the buffer, not in the chunk that is in it. The state that we want the retrieval buffer to be in is specified on line 7: the retrieval buffer needs to be `empty` (no chunk

should be stored there).

In general, we can check for a variety of states that buffers could be in. For example:

- `'?g> buffer full'` checks if the goal buffer is full (whether it carries a chunk);
- `'?retrieval> state busy'` checks if the retrieval buffer is working on retrieving a chunk;
- `'?retrieval> state error'` checks if the last retrieval request has failed (no chunk has been found).

If the preconditions on the two buffers are met, the rule triggers two actions. The first action is stated starting on line 9 below: we modify the `goal_lexeme` chunk by changing the current task from `agree` to `trigger_agreement`. When such a feature-value update takes place, the other features of the updated chunk remain the same.

The `trigger_agreement` task specified in the `goal_lexeme` chunk is to identify a subject noun so that the `goal_lexeme` can agree with that noun in number, which leads us to the second action. This action is stated starting on line 13 below: `+retrieval>` indicates that we access the `retrieval` buffer (recall that we just verified that this buffer is empty) and we add a new chunk to it (that is what + means). This chunk is our memory cue / query: we want to retrieve from declarative memory a chunk of type `word` that is a `noun` and a `subject`.

- strictly speaking, it is not necessary to ensure that the retrieval buffer is empty before placing a retrieval request
  - the model would have worked just as well if the retrieval buffer had been non-empty
  - the buffer would have been flushed / emptied first, and then the memory cue would have been placed in it

Memory cues always consist of chunks, i.e., feature structures, and the retrieval process asks the declarative memory module to provide a (possibly) larger chunk that the cue chunk is a part of (technically, a chunk in declarative memory that is subsumed by our cue chunk). In our specific case, the cue requests the retrieval of a chunk that has at least the following ⟨slot, value⟩ pairs: the chunk should be a `noun` that is a `subject`.

After this production rule is fired, a subject noun is retrieved from declarative memory and placed in the retrieval buffer (assuming the retrieval is successful), and the goal lexeme has `trigger_agreement` as its task. The second production rule, provided below, can now fire and actually perform the agreement:

[10]:
```
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
""")
```

[10]:
```
{'=g': goal_lexeme(category= verb, number= , task= trigger_agreement),
'=retrieval': word(category= noun, form= , meaning= , number= =x, synfunction=
subject)}
==>
{'=g': goal_lexeme(category= verb, number= =x, task= done)}
```

The two preconditions of the rule above ensure that we are in the correct state:

- lines 2-5: the chunk in the goal buffer is subsumed (=) by the chunk on lines 3–5, i.e., it has `verb` as the value of the slot `category`, and `trigger_agreement` as the value of the slot `task`
- lines 6-10: the chunk in the retrieval buffer is subsumed (=) by the chunk on lines 7–10, i.e., it must be of category `noun`, have the syntactic function of `subject` and have a number specification `=x`;
- since `=x` does not appear anywhere else in the preconditions, this last check is vacuous, as a variable can have any value; however, keep in mind that variables take scope within a rule and, therefore, any other part of this rule that will make use of `=x` will have to match in value with the `number` slot in the retrieval buffer.

After checking that we are in the correct state, we trigger the agreeing action. Lines 12-16 tell us that the chunk that is currently in the goal buffer should be kept there (that's what = on line 12 encodes) and its feature structure should be updated as follows. The type and category should stay the same (`goal_lexeme` and `verb`, respectively), but a new number specification should be added, namely `=x`, which is the same number specification as the one for the subject noun we have retrieved from declarative memory. This completes the agreement operation, so the `task` slot of the goal lexeme is updated and marked as `done` (line 16).

The third and final production rule just mops things up: we are done, so the goal buffer is flushed and our simulation ends. The action on line 6 below, namely `~g>`, simply discards the chunk in the goal buffer.

[11]:
```
agreement.productionstring(name="done", string="""
    =g>
    isa goal_lexeme
    task done
    ==>
    ~g>
""")
```

[11]:
```
{'=g': goal_lexeme(category= , number= , task= done)}
==>
{'~g': None}
```

In the next section, we run the model that we have just created. The notation introduced throughout this section is summarized in the two tables below, one for rule preconditions, and one for rule

actions.

Notation & terminology used in the preconditions of production rules

Symbol	=	?
Interpretation	check that subsumption holds	check the status of the buffer
Possible values	any chunk that subsumes the chunk in the buffer	buffer full buffer empty state busy state free state error

Notation & terminology used in the actions of production rules

Symbol	=	+	~
Interpretation	modify the current chunk	add a new chunk to buffer (triggers memory recall if added to retrieval buffer)	clear buffer
Possible values	the chunk in the buffer updated with the new slots & values	a chunk with specified slots & values (for retrieval buffer, old chunk from dec. mem. if recall succeeds)	N/A

[ ]: