11_top_down_parsing_failures_to_parse_and_taking_snapshots_of_the

April 8, 2021

Open in Colab

Model up to this point:

[1]:
```
import pyactr as actr
```

[2]:
```
actr.chunktype("parsing_goal", "stack_top stack_bottom parsed_word task")
actr.chunktype("sentence", "word1 word2 word3")
actr.chunktype("word", "form, cat")
```

[3]:
```
parser = actr.ACTRModel()
dm = parser.decmem
g = parser.goal
imaginal = parser.set_goal(name="imaginal", delay=0.2)
```

[4]:
```
g.add(actr.chunkstring(string="""
    isa parsing_goal
    task parsing
    stack_top S
"""))

imaginal.add(actr.chunkstring(string="""
    isa sentence
    word1 Mary
    word2 likes
    word3 Bill
"""))
```

[5]:
```
dm.add(actr.chunkstring(string="""
    isa word
    form Mary
    cat ProperN
"""))
dm.add(actr.chunkstring(string="""
    isa word
    form Bill
    cat ProperN
"""))
dm.add(actr.chunkstring(string="""
    isa word
    form likes
    cat V
"""))
```

[6]:
```
parser.productionstring(name="expand: S ==> NP VP", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top S
    ==>
    =g>
    isa parsing_goal
    stack_top NP
    stack_bottom VP
""")

parser.productionstring(name="expand: NP ==> ProperN", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top NP
    ==>
    =g>
    isa parsing_goal
    stack_top ProperN
""")

parser.productionstring(name="expand: VP ==> V NP", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top VP
    ==>
    =g>
    isa parsing_goal
    stack_top V
    stack_bottom NP
""")

parser.productionstring(name="retrieve: ProperN", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top ProperN
    =imaginal>
    isa sentence
    word1 =w1
    ==>
    =g>
    isa parsing_goal
    task retrieving
    +retrieval>
    isa word
    form =w1
""")

parser.productionstring(name="retrieve: V", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top V
    =imaginal>
    isa sentence
    word1 =w1
    ==>
    =g>
    isa parsing_goal
    task retrieving
    +retrieval>
    isa word
    form =w1
""")

parser.productionstring(name="scan: word", string="""
    =g>
    isa parsing_goal
    task retrieving
    stack_top =y
    stack_bottom =x
    =retrieval>
    isa word
    form =w1
    cat =y
    =imaginal>
    isa sentence
    word1 =w1
    word2 =w2
    word3 =w3
    ==>
    =g>
    isa parsing_goal
    task printing
    stack_top =x
    stack_bottom None
    parsed_word =w1
    =imaginal>
    isa sentence
    word1 =w2
    word2 =w3
    word3 None
    ~retrieval>
""")

parser.productionstring(name="print parsed word", string="""
    =g>
    isa parsing_goal
    task printing
    =imaginal>
    isa sentence
    word1 ~None
    ==>
    !g>
    show parsed_word
    =g>
    isa parsing_goal
    task parsing
    parsed_word None
""")

parser.productionstring(name="done", string="""
    =g>
    isa parsing_goal
    task printing
    =imaginal>
    isa sentence
    word1 None
    ==>
    !g>
    show parsed_word
    ~imaginal>
    ~g>
""");
```

[7]:
```
# parser_sim = parser.simulation()
# parser_sim.run()
```

0.1 Failures to parse and taking snapshots of the mind when it fails

We can run the parser on ungrammatical sentences to see if and how exactly it fails.

Let's try to parse the word sequence *Bill Mary likes*. - the parser should fail while parsing the second word *Mary* because the noun does not match its expectation to see a verb

We add the relevant chunks to the goal and imaginal buffers and start a new simulation.

- in general, you should reset the declarative memory module (and various buffers) before rerunning a model simulation
- a simple way to reset the model is to reinitialize it from scratch, that is, restart with `parser = actr.ACTRModel()` etc.
  - you can take a look at the code for the more advanced models of lexical decision tasks to see how to reset the state of a model (without restarting it from scratch) so that multiple simulations with the same initial position can be run

[8]:
```
g.add(actr.chunkstring(string="""
    isa parsing_goal
    task parsing
    stack_top S
"""))
imaginal.add(actr.chunkstring(string="""
    isa sentence
    word1 Bill
    word2 Mary
    word3 likes
"""))
```

[9]:
```
parser_sim2 = parser.simulation()
parser_sim2.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: expand: S ==> NP VP')
(0.05, 'PROCEDURAL', 'RULE FIRED: expand: S ==> NP VP')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP ==> ProperN')
(0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP ==> ProperN')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Bill)')
```

```
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: scan: word')
(0.25, 'PROCEDURAL', 'RULE FIRED: scan: word')
(0.25, 'g', 'MODIFIED')
(0.25, 'imaginal', 'MODIFIED')
(0.25, 'retrieval', 'CLEARED')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: print parsed word')
(0.3, 'PROCEDURAL', 'RULE FIRED: print parsed word')
parsed_word Bill
(0.3, 'g', 'EXECUTED')
(0.3, 'g', 'MODIFIED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: expand: VP ==> V NP')
(0.35, 'PROCEDURAL', 'RULE FIRED: expand: VP ==> V NP')
(0.35, 'g', 'MODIFIED')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: retrieve: V')
(0.4, 'PROCEDURAL', 'RULE FIRED: retrieve: V')
(0.4, 'g', 'MODIFIED')
(0.4, 'retrieval', 'START RETRIEVAL')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.45, 'retrieval', 'CLEARED')
(0.45, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Mary)')
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.45, 'PROCEDURAL', 'NO RULE FOUND')
```

- just as before, our goal is to parse a sentence `S` (line 4), namely *Bill Mary likes* (lines 8-10)
- the parser correctly parses the first word *Bill* and prints it (line 40)

The parsing process stops after 450 ms because: - the word *Mary* retrieved from declarative memory is of category ProperN (line 55) - but the top of the parsing goal stack stores the category V, which is what the parser was expecting to retrieve (lines 48-49)

To facilitate the inspection of simulations and models, `pyactr` provides a way to advance simulations one step at a time rather than letting them run from beginning to end.

This makes it easy to check the internal state of the buffers, as well as to diagnose / debug our models. - e.g., if the model gets stuck in an infinite loop.

Let's run the simulation again and go through it step by step.

[10]:
```
g.add(actr.chunkstring(string="""
    isa parsing_goal
    task parsing
    stack_top S
"""))
imaginal.add(actr.chunkstring(string="""
    isa sentence
    word1 Bill
    word2 Mary
    word3 likes
"""))
```

[11]:
```
parser_sim3 = parser.simulation()
parser_sim3.step()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
```

Very little happens in the first step: - the parser simply enters a 'conflict resolution' state in which it identifies the rules that can be fired given the initial cognitive state (that is, the initial state of the buffers)

Let's go through some more steps. - to do that, we use the method `steps` with a parameter that provides the exact number of steps the simulation should advance through

Let's advance 10 steps, which are reflected in the 10 lines of simulation output:

[12]:
```
parser_sim3.steps(10)
```

```
(0, 'PROCEDURAL', 'RULE SELECTED: expand: S ==> NP VP')
(0.05, 'PROCEDURAL', 'RULE FIRED: expand: S ==> NP VP')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP ==> ProperN')
(0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP ==> ProperN')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
```

Now, let's advance our simulation to the point where the rule `"scan: word"` has just fired. To do that, we have to be able to:

- check the current event, i.e., the most recent step taken in the simulation, and
- stop when this event is a `"scan: word"`-rule firing

The current event is an attribute of the simulation. - for example, the current event in our simulation is a ProperN retrieval

[13]:
```
parser_sim3.current_event
```

[13]:
```
Event(time=0.15, proc='PROCEDURAL', action='RULE FIRED: retrieve: ProperN')
```

The event has three attributes:

- `time`: the simulation time at which the event occurred (150 ms in our case)
- `proc`: the module that is affected (procedural memory in our case)
- `action`: the cognitive action that has taken place

Let us now advance to the first firing of the `"scan: word"` rule.

- we do this by running a `while` loop in the Python interpreter
  - the command on line 2 below, i.e., advance one step through the simulation, should be taken while the condition on line 1 is satisfied
  - that condition says that the `action` attribute of the current event should *not* be a `"scan: word"` firing
- `!=` is non-identity in Python, as we already discussed in the notebook introducing Python
- `!` is customarily used for negation in programming languages, and it is distinct from ACT-R negation `~` that we sometimes use inside production rules

[14]:
```
while parser_sim3.current_event.action != 'RULE FIRED: scan: word':
    parser_sim3.step()
```

```
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Bill)')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: scan: word')
(0.25, 'PROCEDURAL', 'RULE FIRED: scan: word')
```

Let's take 3 more steps so that the changes triggered by the `"scan: word"` rule are recorded in the ACT-R mind / model:

[15]:
```
parser_sim3.steps(3)
```

```
(0.25, 'g', 'MODIFIED')
(0.25, 'imaginal', 'MODIFIED')
(0.25, 'retrieval', 'CLEARED')
```

Let's inspect our buffers at this point in the simulation:

[16]:
```
g
```

[16]:
```
{parsing_goal(parsed_word= Bill, stack_bottom= None, stack_top= VP, task=
printing)}
```

- as expected, the top of our parsing goal stack is a VP nonterminal
  - the ProperN nonterminal has just been popped off the stack

[17]:
```
imaginal
```

[17]:
```
{sentence(word1= Mary, word2= likes, word3= None)}
```

- the first word *Bill* has been removed from the sentence stored in the imaginal buffer

[18]:
```
parser.retrieval
```

[18]:
```
set()
```

- the lexical representation for *Bill* has been flushed from the retrieval buffer

Let us now advance to the point where the parsing process failed.

- we will step through the simulation until the `action` attribute of the current event starts with the string `'RETRIEVED'`
- that will be the point where the second word in our string, namely *Mary*, has been retrieved

```
[19]: while not parser_sim3.current_event.action.startswith('RETRIEVED'):
          parser_sim3.step()
```

```
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: print parsed word')
(0.3, 'PROCEDURAL', 'RULE FIRED: print parsed word')
parsed_word Bill
(0.3, 'g', 'EXECUTED')
(0.3, 'g', 'MODIFIED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: expand: VP ==> V NP')
(0.35, 'PROCEDURAL', 'RULE FIRED: expand: VP ==> V NP')
(0.35, 'g', 'MODIFIED')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: retrieve: V')
(0.4, 'PROCEDURAL', 'RULE FIRED: retrieve: V')
(0.4, 'g', 'MODIFIED')
(0.4, 'retrieval', 'START RETRIEVAL')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.45, 'retrieval', 'CLEARED')
(0.45, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Mary)')
```

We can once again inspect the current cognitive state of the model / mind, i.e., the buffer contents:

```
[20]: g
```

```
[20]: {parsing_goal(parsed_word= None, stack_bottom= NP, stack_top= V, task=
      retrieving)}
```

```
[21]: imaginal
```

```
[21]: {sentence(word1= Mary, word2= likes, word3= None)}
```

```
[22]: parser.retrieval
```

```
[22]: {word(cat= ProperN, form= Mary)}
```

And the cause of the parsing failure is apparent:

- the retrieval buffer stores a ProperN
- but the top of the parsing goal stack, i.e., our current parsing expectation / prediction, is a V

The parser therefore halts before the second word in our sentence can be scanned, as shown by the unchanged chunk in the imaginal buffer.

0.2 Top-down parsing as an imperfect psycholinguistic model

It is not enough for our parser to correctly parse grammatical sentences, and fail to parse ungrammatical ones.

- our top-down ACT-R parser is not simply an implementation of an arbitrary parsing algorithm that is satisfactory as long as it works correctly
- this parser is meant to be a limited but realistic model of a certain kind of human cognitive behavior:
  – syntactic parsing in sentence comprehension tasks (e.g., self-paced reading).

Is our parser even remotely adequate as a psycholinguistic model?

One of the empirical adequacy desiderata for our parser is that the temporal trace of parsing a sentence should correspond to the temporal trace of an average human participant completing the same task.

- for example, we see that our parser takes 800 ms to parse the sentence *Mary likes Bill*
- this is roughly correct

But other properties of our parser are more worrying:

- for one, the parser requires this much time while abstracting away from what human participants have to do during an actual self-paced reading task:
  – internalizing visual information
  – projecting sentence meaning
  – executing motor actions (pressing keys) etc.
- so ultimately 800 ms might be too much given the very narrow amount of work our parser actually does
- another issue is that retrieving lexical information always takes 50 ms in our current models and simulations
  – this is hardly realistic
  – we know that lexical retrieval is dependent on various factors, word frequency, priming etc.
  – these factors are completely ignored here
- finally, top-down parsers work well for right-branching structures like the sentence *Mary likes Bill*, but they have significant difficulties with left branching structures.
  – for such structures, the parser would have to store as many symbols on the stack as there are levels of embedding
  – since every expansion of a rule takes 50 ms, we expect left branching structures with n levels of embedding to take $50 * n$ ms
  – this is at odds with actual human performance, see:
    ∗ Johnson-Laird, Philip N. 1983. *Mental models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press
    ∗ Abney, Steven, and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research* 20:233–50

    * Resnik, Philip. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the Fourteenth International Conference on Computational Linguistics*. Nantes, France.
- the main reason for this issue is that our parser generates predictions about syntactic structure
  - **exclusively based on the grammar**
  - **completely ignoring the actual evidence** (the sentence to be parsed) until it reaches a terminal on the leftmost branch

In fact, purely top-down parsers consult the evidence (the word string) only after they predict all the way to lexical items:

- such pure top-down parsers would place memory retrieval requests based on the terminal at the top of the parsing goal stack
- for example, if a ProperN is at the top of the stack, they would retrieve an arbitrary ProperN from declarative memory and only after that, check whether the form of the retrieved ProperN matches the leftmost word to be parsed
- if not, a new retrieval request would be placed for a new ProperN in hopes that the form of that new chunk would match the word to be parsed
- in the worst case, such a purely top-down parser would retrieve all chunks of category ProperN one at at a time from declarative memory, and finally identify the one whose form matches the current word to be parsed
- the temporal trace of such a parser would be very far from the temporal trace of an average human participant completing the same task:
  - if the lexicon contains 20 chunks of ProperN category, and a retrieval takes around 50 ms, it would take a full second to parse the first word in the sentence *Mary likes Bill* in the worst-case scenario
  - and this ignores the time needed to verify that 19 of the retrieved chunks are mismatches, and then the time needed to backtrack and restart the retrieval process

A more plausible human parser would **consult the evidence, i.e., the word string to be parsed, earlier and more often** in the parsing process.

- our top-down parsing strategy needs to be complemented by a bottom-up parsing strategy
- in principle, we could switch from a purely top-down parser to a purely bottom-up parser that is completely driven by the evidence
- such a parser would be incremental, but it would not be predictive in the same way that the human parser seems to be
- we will therefore not explore purely bottom-up (shift-reduce) parsers and instead move directly to **left-corner parsers**, which combine top-down and bottom-up features
  - they can be thought of as predictive top-down parsers with incremental bottom-up filtering

[ ]:
```

```