17_left_corner_parsing_intro_and_dec_mem

April 12, 2021

Open in Colab

0.1 A left-corner parser with visual and motor interfaces

In this notebook, we introduce a left-corner parser that incorporates visual and motor interfaces. The left-corner parser builds on: - the basic top-down parser we introduced before, and - the lexical decision model with visual and motor interfaces we just discussed.

As we mentioned before, left-corner parsers combine top-down and bottom-up features: they can be thought of as predictive top-down parsers with incremental bottom-up filtering.

Left-corner parsing differs from top-down parsing with respect to the amount of evidence necessary to trigger a production rule:

- a grammar rule cannot be triggered without any evidence from the incoming signal / string of words, as it would be in a top-down parser
- but we do not need to accumulate complete evidence, that is, all the necessary words, to trigger a rule, as we would in bottom-up parsing
  - for example, we do not need both words in *Mary sleeps* to trigger the `S -> NP VP` rule

Thus, in left-corner parsing, partial evidence is:

- necessary, in contrast to top-down parsing),
- and also sufficient, in contrast to bottom-up parsing

For example, having evidence for the very first category on the right-hand side of the rule, namely

- the NP in the sentence *Mary sleeps*
- which we describe as having evidence for the *left corner* of the `S -> NP VP` rule

is sufficient to trigger it.

Following Hale, John T. 2014. *Automaton theories of human sentence comprehension*. Stanford: CSLI Publications, we summarize the left-corner parsing strategy in the 'project' and 'project & complete' rules below.

- the only difference between them is the context in which the left-corner rule is triggered:
  - if the mother node, e.g., `S` in our example above, is not expected in context, it is added to the context as a 'found' symbol
    - this is the simple 'project' rule
  - but if the mother node is already expected in context, we check off that expectation as satisfied

        * this is the ‘project & complete’ rule

Finally, the ‘shift’ rule takes words one at a time from the incoming string of words and adds them to the top of the stack to be parsed.

Left-corner parsing rule schemata (Hale, John T. 2014. *Automaton theories of human sentence comprehension*. Stanford: CSLI Publications):

- **Project**:
    - if the symbol Y is at the top of the stack, and there is a grammar rule X → Y Z whose right-hand side starts with Y
    - then replace Y with two new symbols:
        * a record that X has been found, and
        * an expectation for the remaining right-hand side symbol(s) Z
- **Project & complete**:
    - if the symbol Y is at the top of the stack and right below it is an expectation of finding symbol X, and there is a grammar rule X → Y Z
    - then replace both Y and X with an expectation for the remaining right-hand side symbol(s) Z
- **Shift**: if the next word of the sentence is a terminal symbol of the grammar, push it on the top of the stack

The distinction between the two different kinds of left-corner projection

- projection *tout court*, and
- projection plus a completion step

was proposed in Resnik, Philip. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the Fourteenth International Conference on Computational Linguistics*. Nantes, France.

Resnik argues that projection & completion is necessary to keep the stack depth reasonably low when parsing both left-branching and right-branching structures.

- most of our rules will be project & complete rules, with the exception of NPs projected by ProperNs
- if the ProperN is in subject position, it will trigger a simple projection rule for the NP dominating it since we do not have an NP expectation at that point
- but if the ProperN is in object position, the previous application of the `VP -> V NP` rule added an NP expectation to the context, so we can both project and complete the NP at the same time

Let’s build a left-corner parser in ACT-R.

We start by importing `pyactr` and setting the position on the virtual screen where the words in our example – the simple sentence *Mary likes Bill* – will be displayed one at a time.

[1]:
```
import pyactr as actr

environment = actr.Environment(focus_position=(320, 180))
```

We then declare the chunk types we need:

- `parsing_goal` chunks will be stored in the goal buffer and they will drive the parsing cognitive process
- `parse_state` chunks will be stored in the imaginal buffer, and they will provide intermediate internal snapshots of the parsing process, as is befitting of information stored in the imaginal buffer
- `word` chunks will be stored in declarative memory and encode lexical information (in our case, just phonological form and syntactic category) for the words in our target example

[2]:
```
actr.chunktype("parsing_goal", "task stack_top stack_bottom\
                                parsed_word right_frontier")
actr.chunktype("parse_state", "node_cat mother daughter1\
                               daughter2 lex_head")
actr.chunktype("word", "form cat")
```

The `parsing_goal` chunk type has the same slots as in the top-down parser discussed before, with the addition of a `right_frontier` slot.

- the right-frontier slot will be used to record the attachment points for NPs:
  - the `S` node for subject NPs
  - the `VP` node for object NPs

Whenever we store a `parse_state` chunk in the imaginal buffer that will contain information about an NP that has just been parsed, we will take the value in the `right_frontier` slot and record it as the value of the `mother` node for the NP in the imaginal buffer.

Which brings us to the `parse_state` chunk type: these intermediate parsing states that are stored in the imaginal buffer record the progress of the parsing cognitive process.

- the `node_cat` slot records the syntactic category of the current node, i.e., the node that has just been parsed
- the `mother` slot records the mother node of the current node
- the `daughter1` and `daughter2` slots record the daughter nodes of the current node
- and finally, the `lex_head` slot records the lexical head of the current phrasal projection

The `parse_state` chunk type gives us a window into how much ACT-R constrains theories of 'high-level' cognitive processes.

- the goal of the parsing cognitive process can be characterized as incrementally building an unobservable hierarchical tree structure (a structural description) for the target sentence
- but there are strict limits on how the partially-built structure is maintained and accessed during the cognitive process: *we can only store one chunk at a time in any given buffer (goal, imaginal, retrieval)*

This means that **the mind never has a global view of the syntactic tree it is constructing**. - instead, the structure is viewed through a limited, moving window that can 'see' only a part of the under-construction structure

Furthermore, the values stored in the slots of a chunk can encode only 'descriptive' content, not specific memory addresses (e.g., uniquely identifiable time stamps) of specific nodes in the tree.

- this is particularly constraining for phrases like NPs, which are multiply instantiated in a given structure

- their position in the hierarchical structure can be identified only if we encode additional information in their corresponding chunks
- we need to record the lexical head associated with an NP to be able to identify which word it dominates, otherwise the NP might end up dominating any ProperN that has already been built / parsed
  - hence the need for the `lex_head` slot
- we also need to record the point where the full NP is attached in the larger tree, otherwise we might end up attaching the direct object NP to the S node as if it were a subject
  - hence the need for the `mother` slot

These two slots of the `parse_state` chunk type, namely `lex_head` and `mother`, will be exclusively needed for NPs in the left-corner parser introduced in this section.

- there is no deep reason for this
- for simplicity, we only focus on simple mono-clausal target sentences, so only NP and ProperN nodes will be multiply instantiated in any given tree
- when we scale up the parser to include multi-clausal sentences and/or multi-sentential discourses, we will end up using these slots for other node types, e.g., VP and S

We can now initialize the `parser` model and set up separate variables for the declarative memory module (`dm`), the goal buffer (`g`) and the imaginal buffer (`imaginal`).

- we set a delay of 0 ms for the imaginal buffer, going against its default setting of 200 ms
- this default setting is motivated by non-linguistic cognitive processes that are structurally much simpler than language comprehension
  - the 200 ms encoding delay provides a better fit to the reaction time data associated with those processes
- in contrast, a low-delay, or even a no-delay, setting is necessary when modeling language comprehension in ACT-R because this requires rapidly building complex hierarchical representations that are likely to extensively rely on imaginal chunks
- in general, it is reasonable to expect that the systematic modeling of language processing in ACT-R – still very much a nascent endeavor – will occasionally require such departures from received ACT-R wisdom

[3]:
```
parser = actr.ACTRModel(environment, motor_prepared=True)

dm = parser.decmem
g = parser.goal
imaginal = parser.set_goal(name="imaginal", delay=0)
```

We are ready to add lexical entries to declarative memory.

- just as in the case of our top-down parser, we keep the lexical information to a minimum and store only phonological forms and syntactic categories

[4]:
```
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
"""))
```

We also add the starting goal chunk:

- the goal is to read the first word and try to parse a sentence (`stack_top` is `S`)

[5]:
```
g.add(actr.chunkstring(string="""
    isa             parsing_goal
    task            read_word
    stack_top       S
    right_frontier  S
"""))
```

[ ]:
```
```