8_top_down_parsing_model_structure_and_dec_mem

April 8, 2021

Open in Colab

0.1 Top-down parsing

Now that the basic ACT-R cognitive architecture is in place and we're more familiar with its specific implementation in `pyactr`, let us build a basic model of syntactic parsing. Specifically, we will build a top-down parser, i.e., a parser that uses the grammar to make predictions about the sentential structure of the upcoming input.

There are three properties of the human parser that we want our model to capture:

- the parser is *incremental*: syntactic parsing and semantic interpretation do not lag significantly behind the perception of individual words;
- the parser is *predictive*: the processor forms explicit representations of words and phrases that have not yet been heard;
- finally, the parser *satisfies the competence hypothesis*: understanding a sentence / discourse involves the recovery of the structural description of that sentence / discourse on the syntax side, and of the meaning representation on the semantic side.

Some references: - Marslen-Wilson, William. 1973. Linguistic structure and speech shadowing at very short latencies. *Nature* 244:522–523 - Frazier, Lyn, and Janet Dean Fodor. 1978. The sausage machine: A new two-stage parsing model. *Cognition* 6:291–325 - Tanenhaus, M. K., M. J. Spivey-Knowlton, K. M. Eberhard, and J. C. Sedivy. 1995. Integration of visual and linguistic information in spoken language comprehension. *Science* 268:1632–1634 - Steedman, Mark. 2001. *The syntactic process*. Cambridge, MA: MIT Press - Hale, John. 2011. What a rational parser would do. *Cognitive Science* 35:399–443

A top-down parser satisfies these conditions, and it has the pedagogical advantage of being very simple (too simple, in fact, to be cognitively plausible). So let's start with one.

Suppose we have a simplecontext-free grammar with the following rules:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	ProperN
VP	$\rightarrow$	V NP
ProperN	$\rightarrow$	Mary
ProperN	$\rightarrow$	Bill
V	$\rightarrow$	likes

We assume that we have only - two proper names - one transitive verb

Our goal is to build a top-down parser that is able to analyze the sentence *Mary likes Bill*.

We assume the sentence is presented to the comprehender one word at a time in the manner of self-paced reading tasks (Just, Marcel A., Patricia A. Carpenter, and Jacqueline D. Woolley. 1982. Paradigms and pro- cesses in reading comprehension. *Journal of Experimental Psychology: General* 111:228–238).

In such tasks, the words are hidden and only one word is uncovered at a time with a spacebar press. The human reader decides when to press the spacebar to uncover the next word (which automatically hides the current word), hence the name of self-paced reading.

So reading our sentence *Mary likes Bill* will happen in four successive stages. In one such version of self-paced reading (the so-called non-cumulative moving-window paradigm), the whole process would look as shown below.

- initial display:

```
---- ----- ---
```

- after one spacebar press:

```
Mary ----- ---
```

- after another spacebar press:

```
---- likes ---
```

- after the third spacebar press:

```
---- ----- Bill
```

Self-paced reading tasks mimic an essential aspect of naturally-occurring language comprehension with auditory stimuli:

- the signal is strictly linearly and strictly incrementally presented one word at a time

Just as in naturally-occurring verbal interactions, and unlike in normal reading situations, the linguistic signal cannot be ‘rewound’ to previous words

- we cannot just look back and reread previous parts of the text

or ‘fast-forwarded’ to subsequent words

- we cannot jump ahead to parts of the text that do not immediately follow the word currently being read.

We can proceed to the characterization of our processing model. A top-down parser can be thought of as a push-down automaton, i.e., an automaton that has a basic form of memory represented as a **stack**.

- the stack stores parsing goals and subgoals in a strict, total order
- these goals are accomplished one at a time by accessing the top of the stack

In our case, the parsing goals are simply syntactic categories that have to be parsed, i.e., that have to be identified in the incoming string.

For example, when we start the parsing process, we push the initial goal of parsing an S node onto the stack. The stack has now only one goal in it, namely 'parse an S', and the goal sits at the top of the stack.

$$\boxed{S}$$

We pop goals off the stack one at a time: we can only look at the top of the stack and remove the current top goal when this goal is accomplished or broken down exhaustively into subgoals.

For example, we will pop the 'parse an S' goal off the stack when we apply the first grammar rule above and replace this goal with two subgoals:

- first parse an NP (i.e., identify an NP in the incoming word input)
- then parse a VP.

The resulting stack will now have two goals: the top one is 'parse an NP', and the one below it is 'parse a VP':

$$\boxed{S} \quad \Rightarrow \quad \begin{array}{|c|}\hline NP \\ \hline VP \\ \hline\end{array}$$

The parser works by modifying the contents of its stack based on two pieces of information: the top element on the stack and, possibly, the current word that has to be parsed (the leftmost word in the incoming string of words).

We can sum up top-down parsing as a parsing strategy that applies two algorithm schemata, *expand* and *scan*, in this order (see, for example, Hale, John T. 2014. *Automaton theories of human sentence comprehension*. Stanford: CSLI Publications, for a good, detailed introduction):

Top-down parsing rules:

- **expand**:
  - if the stack has a symbol X on top, and the grammar contains a rule $X \to A\ B$ or $X \to A$, then:
    - pop X and push down onto the stack the symbols B and A (in that order), or
    - pop X and push down onto the stack the symbol A.
- **scan**:
  - if the top of stack has a terminal symbol – a symbol like V or *ProperN* that rewrites to a lexical item; that is, a part of speech – and w, the leftmost word to be parsed, is of that part of speech, then:
    - pop the terminal symbol off the stack and remove w from the word string that is to be parsed.

Let us now code a top-down parser in `pyactr` that implements these two general parsing rules and uses the grammar above. Recall that the example sentence we will parse is *Mary likes Bill*.

0.2 Building a top-down parser in `pyactr` [Part 1]

Let us start with the first standard step, importing `pyactr`.

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

We should now specify the types of chunks we need. We will have one type for parsing goals. The parsing goal will keep track of:

- the stack content: we only need two positions in the stack for our current purposes – the top and the bottom of the stack; this is a consequence of the fact that our grammar generates at most binary branching trees with no left recursion (cf. Resnik, Philip. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the Fourteenth International Conference on Computational Linguistics*. Nantes, France);
- the current word being parsed (if any);
- the current task of the parser, that is, the current state our parsing model is in – basically, 'parsing' if the parse is still ongoing, and 'done' if the parsing is finished.

[2]:
```
actr.chunktype("parsing_goal", "stack_top stack_bottom parsed_word task")
```

The second chunk type we need to declare is one that will enable us to represent the incoming sentence, i.e., the incoming word string, to be parsed.

This might seem counter-intuitive: why should we represent the sentence to be parsed in a chunk? The sentence is external to the agent, it's what the agent reads or hears.

However, at this point, we have no way of representing the surrounding environment and the basic input/output interfaces between the mind and the environment. We therefore have to represent a sentence internally as a chunk.

When we introduce the vision and motor modules in future notebooks (associated with Chapter 4 of the *Computational Cognitive Modeling and Linguistics Theory* book), we will be able to develop a more intuitive and elegant solution.

The chunk type for sentences only needs to store three words since our target sentence is only that long.

[3]:
```
actr.chunktype("sentence", "word1 word2 word3")
```

0.2.1 Modules, buffers, and the lexicon

Let us now initialize the model and set up more convenient ways of accessing the declarative memory module and the goal buffer:

[4]:
```
parser = actr.ACTRModel()
dm = parser.decmem
g = parser.goal
```

The goal buffer will store a `parsing_goal` chunk, which:

- carries the information that drives the parsing process, and
- is updated throughout that process.

But we also need to store the word sequence that we need to parse, so we will create a second buffer that is similar to the goal buffer and that will store the sentence to be parsed.

Having two goal-like buffers is not uncommon in ACT-R.

- the first buffer is the actual goal buffer, which keeps track of the information driving the cognitive process
- the other one is the *imaginal* buffer; this buffer:
  - is associated with the imaginal module
  - maintains an internal image of the information associated with the current cognitive process that provides contextual information relevant for the current task.

Thus, storing the sentence to be parsed in the imaginal buffer is an acceptable approximation of the cognitive behavior we're trying to model.

[5]:
```
imaginal = parser.set_goal(name="imaginal", delay=0.2)
```

Above, we create a new goal buffer, which we call the `imaginal` buffer.

- the string `"imaginal"` sets the name under which the model will recognize and access the buffer (e.g., in production rules)
- the `delay` attribute of the imaginal buffer encodes the delay required to set a chunk in the buffer:
  - it will take 0.2 seconds (200 ms) to set a chunk in the `imaginal` buffer
  - this is the standard value for this buffer, in contrast to the `goal` buffer, which sets a chunk immediately

Finally, we assign this new buffer to a variable `imaginal` so that we can access it more easily.

The goal and imaginal buffers, and more generally the state of the buffers at any given point in a cognitive process provides the internal state, or the context, of the cognitive process at that point.

For example, items in memory that share values with items in the goal or imaginal buffers are contextually 'primed':

- they are more salient (technically, more activated) than other items
- they are therefore easier to retrieve, precisely because they are relevant in context.

Thus, the cognitive context in the sense of **the current state of the buffers** has a function similar to variable assignments in first-order logic.

- assignments in first-order logic provide the current context of interpretation relative to which incoming expressions are interpreted
- similarly, the state of the buffers in an ACT-R model of the mind provide the context for the next step in the cognitive process.

Incidentally, the counterpart of the model in first-order logic is the content of the modules, particularly the facts stored in declarative memory and the rules stored in procedural memory.

We can now add chunks to the `goal` and `imaginal` buffers:

[6]:
```
g.add(actr.chunkstring(string="""
    isa parsing_goal
    task parsing
    stack_top S
"""))

g
```

[6]:
```
{parsing_goal(parsed_word= , stack_bottom= , stack_top= S, task= parsing)}
```

- the `goal` buffer switches to an active `parsing` state / `task`, and
- the current parsing goal, i.e., the top of the stack, is set to parsing a sentence (`S`)

[7]:
```
imaginal.add(actr.chunkstring(string="""
    isa sentence
    word1 Mary
    word2 likes
    word3 Bill
"""))

imaginal
```

[7]:
```
{sentence(word1= Mary, word2= likes, word3= Bill)}
```

- in the `imaginal` buffer, we set the `sentence` to be parsed to *Mary likes Bill*

We are now ready to start answering our main question: **how do we code the top-down parser itself?**

We will assume that the grammar and associated parsing rules are part of the `procedural` module, i.e., they are encoded by production rules.

- this contrasts with lexical information, which is commonly encoded in declarative memory
- see **Lewis, Richard, and Shravan Vasishth. 2005. An activation-based model of sentence process- ing as skilled memory retrieval.** ***Cognitive Science*** **29:1–45** for more discussion and arguments for this division of labor between the lexicon and grammar.

We specify our lexicon first. For simplicity, our lexical representations will encode only:

- the form (represented as the standard English spelling of the lexeme)
- the part of speech (syntactic category) tags of our lexical items

[8]:
```
actr.chunktype("word", "form, cat")

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
"""))

dm
```

[8]:
```
{word(cat= ProperN, form= Mary): array([0.]), word(cat= ProperN, form= Bill):
array([0.]), word(cat= V, form= likes): array([0.])}
```

[ ]:
```
```