Chapter 3
The Basics of Syntactic Parsing in ACT-R

In this chapter, we introduce the basics of syntactic parsing in ACT-R. We build a top-down parser and learn how we can extract intermediate stages of `pyactr` simulations. This enables us to inspect detailed snapshots of the cognitive states that our processing models predict.

3.1 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 (Marslen-Wilson 1973, Frazier and Fodor 1978, Tanenhaus et al. 1995, Steedman 2001, Hale 2011 among others):

i. the parser is *incremental*: syntactic parsing and semantic interpretation do not lag significantly behind the perception of individual words;
ii. the parser is *predictive*: the processor forms explicit representations of words and phrases that have not yet been heard;
iii. 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.

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). It is, therefore, a good place to start.

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_3

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

(1)

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

For simplicity, we assume that we have only two proper names in our language and 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 et al. 1982). 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 in (2) below.

(2)

i.	initial display:	`---- ----- ----`
ii.	after one spacebar press:	`Mary ----- ----`
iii.	after another spacebar press:	`---- likes ----`
iv.	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.

With the empirical task fully characterized as a self-paced reading task, 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 and 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.

(3) $\boxed{\text{S}}$

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 (i) accomplished or (ii) 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 in (1) 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’.

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

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 Hale 2014 among others for an introduction):

(5) Top-down parsing rules:

a. *expand*: if the stack has a symbol X on top, and the grammar contains a rule $X \rightarrow A\ B$ or $X \rightarrow A$, pop X and push down onto the stack the symbols B and A (in that order), or the symbol A, respectively.

b. *scan*: if the top of the 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 implement a top-down parser in `pyactr` that consists of these two general parsing rules and uses the grammar in (1). Recall that the example sentence we will parse is *Mary likes Bill*.

3.2 Building a Top-Down Parser in `pyactr`

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

[py1]
```
>>> import pyactr as actr                                    1
```

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 (1) generates at most binary branching trees with no left recursion (cf. Resnik 1992);
- 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.

[py2]
```
>>> 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 word string to be parsed. This might seem counterintuitive: 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 Chap. 4, 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 that long:

[py3]
```
>>> actr.chunktype("sentence", "word1 word2 word3")
```

3.2.1 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:

[py4]
```
>>> 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 which 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 and maintains an internal image of the information associated with the current cognitive process, thereby providing 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.

[py5]
```
>>> imaginal = parser.set_goal(name="imaginal", delay=0.2)
```

In **[py5]**, we create a new goal buffer, 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 is the time needed to encode/set a chunk in the buffer, which is 0.2 s (200 ms). This is the default ACT-R value for this buffer, in contrast to the `goal` buffer which sets a chunk immediately. Finally, **[py5]** assigns this new buffer to a variable `imaginal` so that we can access it more easily in the Python interpreter.

The goal and imaginal buffers—more generally, all the buffers at any given point in a cognitive process—provide the internal state, or the context, of the cognitive process at that point. For example, chunks in memory that share values with chunks in the goal or imaginal buffers are contextually 'primed': they are more salient than other items and are easier to retrieve 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 upcoming 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.

We can even extend this analogy to models, i.e., to the other parameter that the interpretation function in first-order logic is relativized to. The ACT-R counterpart of a first-order logic model 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:

[py6]
```
>>> g.add(actr.chunkstring(string="""
...     isa       parsing_goal
...     task      parsing
...     stack_top S
... """))
>>> g
{parsing_goal(parsed_word= , stack_bottom= , stack_top= S, task= parsing)}
>>> imaginal.add(actr.chunkstring(string="""
...     isa   sentence
...     word1 Mary
...     word2 likes
...     word3 Bill
... """))
>>> imaginal
{sentence(word1= Mary, word2= likes, word3= Bill)}
```

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`). In the `imaginal` buffer, we set the `sentence` to be parsed to *Mary likes Bill*.

We are now ready to start answering the main question of the chapter: how do we implement 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 in production rules. This contrasts with lexical information, which is commonly encoded in declarative memory. See Lewis and Vasishth (2005) for more discussion and arguments for this division of labor between declarative and procedural memory when encoding the lexicon and the grammar and parser.

We specify our lexicon first. For simplicity, our lexical representations will encode only the form (we use the written form, for simplicity) and the part of speech (syntactic category) tags of our lexical items:

[py7]
```
>>> 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
{word(cat= ProperN, form= Mary): array([0.]),
 word(cat= ProperN, form= Bill): array([0.]),
 word(cat= V, form= likes): array([0.])}
```

3.2.2 *Production Rules*

We now turn to the production rules that encode both our context-free grammar rules in (1) and the top-down parsing strategy codified by the `expand` and `scan` rules in (5).

The first rule is an expanding rule, encoding the first phrase structure rule of our grammar: we expand S into NP and VP, in that order.

[py8]
```
>>> 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
... """)
{'=g': parsing_goal(parsed_word= , stack_bottom= ,
                    stack_top= S, task= parsing)}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= VP,
                    stack_top= NP, task= )}
```

Note how the rule pops the `S` goal off the stack and replaces it with two subgoals `NP` and `VP`, in that order. We do not modify the current `task`, which should remain specified as `parsing`, so we omit it from the specification of the action: the chunk in the consequent/right-hand side of the production rule only specifies the slots whose values should be updated, namely `stack_top` and `stack_bottom`.

The second rule is once again an expanding rule: NP is expanded into ProperN.

[py9]
```
>>> parser.productionstring(name="expand: NP ==> ProperN", string="""
...     =g>
...     isa       parsing_goal
...     task      parsing
...     stack_top NP
...     ==>
...     =g>
...     isa       parsing_goal
...     stack_top ProperN
... """)
{'=g': parsing_goal(parsed_word= , stack_bottom= ,
                    stack_top= NP, task= parsing)}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= ,
                    stack_top= ProperN, task= )}
```

Note that the rule only updates the top of the stack. The bottom of the stack is left unmodified, so it is omitted throughout the rule.

The third production rule expands VP into V and NP:

[py10]
```
>>> 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
... """)
{'=g': parsing_goal(parsed_word= , stack_bottom= ,
                    stack_top= VP, task= parsing)}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= NP,
                    stack_top= V, task= )}
```

This rule is almost identical to the first rule: we only change the syntactic category symbols. Crucially, note that the rule is triggered only when the ‘parse a VP’ goal is at the *top* of the stack. Thus, to trigger this third rule, something must happen after the successive application of the first and second rules `"expand: S ==> NP VP"` and `"expand: NP ==> ProperN"` that will promote the VP goal from the bottom of the stack to the top of the stack.

Goals at the bottom of the stack can be promoted to the top when the top goal is popped off the stack and is not replaced by another goal. This is what happens in a *scan* step: in our case, a `scan` rule needs to (i) pop the `ProperN` goal off the top of the stack and, at the same time, (ii) scan the first word `Mary` of our target sentence.

That is, once we have a terminal (ProperN, V) at the top of our stack, we have to check that the terminal matches the category of the word to be parsed. If so, the word is parsed. We achieve this by means of two rules. First, we place a retrieval request for a lexical item stored in declarative memory whose form is the current word to be parsed. Then, if a lexical item is successfully retrieved and the syntactic category of that lexical item is the same as the terminal at the top of our stack, the current word is scanned and the top symbol on our stack is popped.

The two retrieval rules for our two terminal symbols (ProperN, V) are provided below. In both cases, we place a retrieval request based on the form of the first word in the sentence to be parsed (`=w1`) and we change the state of the parsing goal to `retrieving` (rather than `parsing`):

[py11]
```
>>> 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
... """)
{'=g': parsing_goal(parsed_word= , stack_bottom= ,
                    stack_top= ProperN, task= parsing),
 '=imaginal': sentence(word1= =w1, word2= , word3= )}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= ,
                    stack_top= , task= retrieving),
 '+retrieval': word(cat= , form= =w1)}
```

[py12]
```
>>> 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
... """)
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= V,
 task= parsing), '=imaginal': sentence(word1= =w1, word2= , word3= )}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= ,
                    stack_top= , task= retrieving),
 '+retrieval': word(cat= , form= =w1)}
```

If the retrieved lexical item matches the top of our stack in syntactic category, we parse the word, pop the top symbol off the stack, and move to the next word in our sentence (that is, we promote `word2` in our sentence to `word1`, and `word3` to `word2`):

[py13]
```
>>> 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>                                                   17
...     isa          parsing_goal                             18
...     task         printing                                 19
...     stack_top    =x                                       20
...     stack_bottom None                                     21
...     parsed_word  =w1                                      22
...     =imaginal>                                            23
...     isa   sentence                                        24
...     word1 =w2                                             25
...     word2 =w3                                             26
...     word3 None                                            27
...     ~retrieval>                                           28
... """)                                                      29
{'=g': parsing_goal(parsed_word= , stack_bottom= =x,          30
                    stack_top= =y, task= retrieving),         31
 '=retrieval': word(cat= =y, form= =w1),                      32
 '=imaginal': sentence(word1= =w1, word2= =w2, word3= =w3)}   33
==>                                                           34
{'=g': parsing_goal(parsed_word= =w1, stack_bottom= None,     35
                    stack_top= =x, task= printing),           36
 '=imaginal': sentence(word1= =w2, word2= =w3, word3= None),  37
 '~retrieval': None}                                          38
```

Note how on lines 20–21 of **[py13]**, the top of the stack is popped, so the symbol on the bottom of the stack is promoted to the top of the stack. Similarly, the imaginal buffer is updated on lines 23–27. The word `=w1` that we just parsed is deleted from the sentence, so the word string that we still need to parse contains only words `=w2` and `=w3`. These remaining words are promoted to the `word1` and `word2` positions. We also clear the retrieval buffer (`~retrieval>` on line 28).

Finally, as a convenience, the parsed word `=w1` is stored in the `parsed_word` slot of the parsing goal chunk (line 22 in **[py13]**), and we enter a new `printing` state (line 19 in **[py13]**). This new state will trigger a print action reporting which word was just parsed. The print action, performed by the rule in **[py14]** below, is helpful to us as modelers, but it is not a necessary part of our processing model.

[py14]

```
>>> parser.productionstring(name="print parsed word", string="""   1
...     =g>                                                        2
...     isa  parsing_goal                                          3
...     task printing                                              4
...     =imaginal>                                                 5
...     isa   sentence                                             6
...     word1 ~None                                                7
...     ==>                                                        8
...     !g>                                                        9
...     show parsed_word                                           10
...     =g>                                                        11
...     isa         parsing_goal                                   12
...     task        parsing                                        13
...     parsed_word None                                           14
... """)                                                           15
{'=g': parsing_goal(parsed_word= , stack_bottom= ,                 16
                    stack_top= , task= printing),                  17
 '=imaginal': sentence(word1= ~None, word2= , word3= )}            18
==>                                                                19
{'!g': ([(['show', 'parsed_word'], {})], {}),                      20
 '=g': parsing_goal(parsed_word= None, stack_bottom= ,             21
                    stack_top= , task= parsing)}                   22
```

The production rule in **[py14]** says that, if the current parsing goal is in a `printing` state (line 4 in **[py14]**) and the slot `word1` in the imaginal buffer is not empty (the squiggle ~ on line 7 is negation), that is, we still have words to parse, then we should print the `parsed_word` in the `goal` buffer (lines 9–10). Line 9 `!g>

should execute an action that involves the goal buffer. The action is then specified on line 10: call the method `show`, which will print the value of the `parsed_word` slot. When we're done printing, we delete the contents of the `parsed_word` slot and re-enter an active state of `parsing` (lines 11–14).

The last production we have to consider is the 'wrap-up' production we trigger at the end of the parsing process, provided in [py15] below. The parsing process ends when the `word1` slot in the imaginal buffer chunk has the value `None` (line 7) and the task is `printing` (line 4). We therefore print the final word of the sentence that was just parsed (lines 9–10) and declare the parsing process done by clearing the `imaginal` and `goal` buffers (lines 11–12).

[py15]
```
>>> parser.productionstring(name="done", string="""             1
...     =g>                                                      2
...     isa  parsing_goal                                        3
...     task printing                                            4
...     =imaginal>                                               5
...     isa   sentence                                           6
...     word1 None                                               7
...     ==>                                                      8
...     !g>                                                      9
...     show parsed_word                                        10
...     ~imaginal>                                              11
...     ~g>                                                     12
... """)                                                        13
{'=g': parsing_goal(parsed_word= , stack_bottom= ,              14
                    stack_top= , task= printing),               15
 '=imaginal': sentence(word1= None, word2= , word3= )}          16
==>                                                             17
{'!g': ([(['show', 'parsed_word'], {})], {}),                   18
 '~imaginal': None, '~g': None}                                 19
```

3.3 Running the Model

We run the model as before: we first instantiate a simulation of the model and then run it.

[py16]
```
>>> parser_sim = parser.simulation()
>>> parser_sim.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= Mary)')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: scan: word')
(0.25, 'PROCEDURAL', 'RULE FIRED: scan: word')
```

```
(0.25, 'g', 'MODIFIED')                                             23
(0.25, 'imaginal', 'MODIFIED')                                      24
(0.25, 'retrieval', 'CLEARED')                                      25
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')                         26
(0.25, 'PROCEDURAL', 'RULE SELECTED: print parsed word')            27
(0.3, 'PROCEDURAL', 'RULE FIRED: print parsed word')                28
parsed_word Mary                                                    29
(0.3, 'g', 'EXECUTED')                                              30
(0.3, 'g', 'MODIFIED')                                              31
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')                          32
(0.3, 'PROCEDURAL', 'RULE SELECTED: expand: VP ==> V NP')           33
(0.35, 'PROCEDURAL', 'RULE FIRED: expand: VP ==> V NP')             34
(0.35, 'g', 'MODIFIED')                                             35
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')                         36
(0.35, 'PROCEDURAL', 'RULE SELECTED: retrieve: V')                  37
(0.4, 'PROCEDURAL', 'RULE FIRED: retrieve: V')                      38
(0.4, 'g', 'MODIFIED')                                              39
(0.4, 'retrieval', 'START RETRIEVAL')                               40
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')                          41
(0.4, 'PROCEDURAL', 'NO RULE FOUND')                                42
(0.45, 'retrieval', 'CLEARED')                                      43
(0.45, 'retrieval', 'RETRIEVED: word(cat= V, form= likes)')         44
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')                         45
(0.45, 'PROCEDURAL', 'RULE SELECTED: scan: word')                   46
(0.5, 'PROCEDURAL', 'RULE FIRED: scan: word')                       47
(0.5, 'g', 'MODIFIED')                                              48
(0.5, 'imaginal', 'MODIFIED')                                       49
(0.5, 'retrieval', 'CLEARED')                                       50
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')                          51
(0.5, 'PROCEDURAL', 'RULE SELECTED: print parsed word')             52
(0.55, 'PROCEDURAL', 'RULE FIRED: print parsed word')               53
parsed_word likes                                                   54
(0.55, 'g', 'EXECUTED')                                             55
(0.55, 'g', 'MODIFIED')                                             56
(0.55, 'PROCEDURAL', 'CONFLICT RESOLUTION')                         57
(0.55, 'PROCEDURAL', 'RULE SELECTED: expand: NP ==> ProperN')       58
(0.6, 'PROCEDURAL', 'RULE FIRED: expand: NP ==> ProperN')           59
(0.6, 'g', 'MODIFIED')                                              60
(0.6, 'PROCEDURAL', 'CONFLICT RESOLUTION')                          61
(0.6, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')             62
(0.65, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')               63
(0.65, 'g', 'MODIFIED')                                             64
(0.65, 'retrieval', 'START RETRIEVAL')                              65
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')                         66
(0.65, 'PROCEDURAL', 'NO RULE FOUND')                               67
(0.7, 'retrieval', 'CLEARED')                                       68
(0.7, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Bill)')     69
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')                          70
(0.7, 'PROCEDURAL', 'RULE SELECTED: scan: word')                    71
(0.75, 'PROCEDURAL', 'RULE FIRED: scan: word')                      72
(0.75, 'g', 'MODIFIED')                                             73
(0.75, 'imaginal', 'MODIFIED')                                      74
(0.75, 'retrieval', 'CLEARED')                                      75
(0.75, 'PROCEDURAL', 'CONFLICT RESOLUTION')                         76
(0.75, 'PROCEDURAL', 'RULE SELECTED: done')                         77
(0.8, 'PROCEDURAL', 'RULE FIRED: done')                             78
parsed_word Bill                                                    79
(0.8, 'g', 'EXECUTED')                                              80
(0.8, 'imaginal', 'CLEARED')                                        81
(0.8, 'g', 'CLEARED')                                               82
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')                          83
(0.8, 'PROCEDURAL', 'NO RULE FOUND')                                84
```

The parser runs as expected: we successfully parse our three-word sentence. The time course of the parsing is as follows.

The first word *Mary* is parsed at the 250 ms mark when the `scan: word` rule is fired for the first time (lines 22–25) and printed by the time 300 ms of simulation time have elapsed (line 29 in **[py16]**).

The second word *likes* is parsed at the 500 ms mark when the `scan: word` rule is fired for the second time (lines 47–50) and printed after 550 ms of total simulation time (line 54).

The final word *Bill* is parsed at the 750 ms mark when the `scan: word` rule is fired for the third and final time (lines 72–75) and printed after 800 ms of simulation time have passed (line 79).

Let us examine the content of the declarative memory module at the end of the simulation. It should contain the lexical items we added at the very beginning of the simulation, as well as the chunks stored in the goal and imaginal buffers right before we cleared them at the end of the parsing process (recall that clearing the buffers always moves their contents to declarative memory).

[py17]
```
>>> dm
{word(cat= ProperN, form= Mary): array([0.  , 0.25]),
 word(cat= ProperN, form= Bill): array([0.  , 0.75]),
 word(cat= V, form= likes): array([0. , 0.5]),
 sentence(word1= None, word2= None, word3= None): array([0.8]),
 parsing_goal(parsed_word= Bill, stack_bottom= None,
              stack_top= None, task= printing): array([0.8])}
```

As expected, we see in [py17] that the goal chunk stored in declarative memory has an empty stack, and the imaginal chunk has an empty sentence (no words). Furthermore, both these chunks have been stored/activated in memory at the 800 ms mark, i.e., at the end of the simulation.

We also see the three lexical items *Mary*, *likes* and *Bill*, each of which has two activation time stamps. The first activation time is at 0 ms, when they were all added to declarative memory before running the simulation. The second activation time is at 250, 500 and 750 ms respectively, when they were parsed during the simulation. Specifically, these are the times when the retrieval buffer was cleared by the three firings of the `scan: word` rule.

Chapter 6 discusses the inner workings of declarative memory in detail. We will see there that this schedule of activations for items in memory is a crucial component of determining the relative salience of items in memory. The salience, or activation, of an item modulates how easy it is to retrieve it—specifically, the probability of a successful retrieval and the time that the retrieval takes.

3.4 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 the parser's expectation to see a verb.

We add the relevant chunks to the goal and imaginal buffers and start a new simulation. Note that, in general, it is recommended to 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 in Chaps. 7, 8 and 9 to see how to reset the state of a model without restarting it from scratch, so that multiple simulations with the same initial model state can be run.

[py18]
```
>>> g.add(actr.chunkstring(string="""                                        1
...     isa       parsing_goal                                               2
...     task      parsing                                                    3
...     stack_top S                                                          4
... """))                                                                    5
>>> imaginal.add(actr.chunkstring(string="""                                 6
...     isa   sentence                                                       7
...     word1 Bill                                                           8
...     word2 Mary                                                           9
...     word3 likes                                                          10
... """))                                                                    11
>>> parser_sim2 = parser.simulation()                                        12
>>> parser_sim2.run()                                                        13
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                     14
(0, 'PROCEDURAL', 'RULE SELECTED: expand: S ==> NP VP')                      15
(0.05, 'PROCEDURAL', 'RULE FIRED: expand: S ==> NP VP')                      16
(0.05, 'g', 'MODIFIED')                                                      17
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  18
(0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP ==> ProperN')                19
(0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP ==> ProperN')                    20
(0.1, 'g', 'MODIFIED')                                                       21
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                   22
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')                      23
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')                        24
(0.15, 'g', 'MODIFIED')                                                      25
(0.15, 'retrieval', 'START RETRIEVAL')                                       26
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  27
(0.15, 'PROCEDURAL', 'NO RULE FOUND')                                        28
(0.2, 'retrieval', 'CLEARED')                                                29
(0.2, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Bill)')              30
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                   31
(0.2, 'PROCEDURAL', 'RULE SELECTED: scan: word')                             32
(0.25, 'PROCEDURAL', 'RULE FIRED: scan: word')                               33
(0.25, 'g', 'MODIFIED')                                                      34
(0.25, 'imaginal', 'MODIFIED')                                               35
(0.25, 'retrieval', 'CLEARED')                                               36
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  37
(0.25, 'PROCEDURAL', 'RULE SELECTED: print parsed word')                     38
(0.3, 'PROCEDURAL', 'RULE FIRED: print parsed word')                         39
parsed_word Bill                                                             40
(0.3, 'g', 'EXECUTED')                                                       41
(0.3, 'g', 'MODIFIED')                                                       42
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                   43
(0.3, 'PROCEDURAL', 'RULE SELECTED: expand: VP ==> V NP')                    44
(0.35, 'PROCEDURAL', 'RULE FIRED: expand: VP ==> V NP')                      45
(0.35, 'g', 'MODIFIED')                                                      46
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  47
(0.35, 'PROCEDURAL', 'RULE SELECTED: retrieve: V')                           48
(0.4, 'PROCEDURAL', 'RULE FIRED: retrieve: V')                               49
(0.4, 'g', 'MODIFIED')                                                       50
(0.4, 'retrieval', 'START RETRIEVAL')                                        51
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                   52
(0.4, 'PROCEDURAL', 'NO RULE FOUND')                                         53
(0.45, 'retrieval', 'CLEARED')                                               54
(0.45, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Mary)')             55
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  56
(0.45, 'PROCEDURAL', 'NO RULE FOUND')                                        57
```

Just as before, our goal is to parse a sentence `S` (**[py18]**, line 4), namely *Bill Mary likes* (lines 8–10). The parser correctly parses the first word *Bill* and prints it (line 40). But the parsing process stops after 450 ms because the word *Mary* retrieved from declarative memory is of category ProperN (line 55). The top of the goal stack, however, 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 in **[py18]** again and go through it step by step.

[py19]
```
>>> 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
... """))
>>> 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. In **[py20]**, we advance 10 steps, as reflected in the 10 lines of simulation output.

[py20]
```
>>> 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')
```

Let's now advance our simulation to the point where the rule `"scan: word"` has just fired. To be able 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:

[py21]
```
>>> parser_sim3.current_event
Event(time=0.15, proc='PROCEDURAL', action='RULE FIRED: retrieve: ProperN')
```

As shown in [py21], the event has three attributes:

- `time`—the simulation time at which the event occurred (150 ms in our case),
- `proc`—the module or buffer that is affected (procedural memory in our case), and
- `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 in [py22] below, i.e., advance one step through the simulation, should be executed 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. Note that `!=` is non-identity in Python; `!` is customarily used for negation in programming languages, and it is distinct from ACT-R negation `~`.

[py22]

```
>>> while parser_sim3.current_event.action != 'RULE FIRED: scan: word':   1
...     parser_sim3.step()                                                2
...                                                                       3
(0.15, 'g', 'MODIFIED')                                                   4
(0.15, 'retrieval', 'START RETRIEVAL')                                    5
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')                               6
(0.15, 'PROCEDURAL', 'NO RULE FOUND')                                     7
(0.2, 'retrieval', 'CLEARED')                                             8
(0.2, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Bill)')           9
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                10
(0.2, 'PROCEDURAL', 'RULE SELECTED: scan: word')                          11
(0.25, 'PROCEDURAL', 'RULE FIRED: scan: word')                            12
```

We can now inspect our buffers. As expected, the top of our parsing goal stack is a ProperN terminal, the first word *Bill* is about to be removed from the sentence stored in the imaginal buffer (but is still there at this simulation step), and the lexical representation for *Bill* is accessible in the retrieval buffer:

[py23]

```
>>> g
{parsing_goal(parsed_word= Bill, stack_bottom= None,
              stack_top= VP, task= printing)}
>>> imaginal
{sentence(word1= Bill, word2= Mary, word3= likes)}
>>> parser.retrieval
{word(cat= ProperN, form= Bill)}
```

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:

[py24]

```
>>> while not parser_sim3.current_event.action.startswith('RETRIEVED'):
...     parser_sim3.step()
...
(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)')
```

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

[py25]
```
>>> parser.retrieval
{word(cat= ProperN, form= Mary)}
>>> g
{parsing_goal(parsed_word= None, stack_bottom= NP,
              stack_top= V, task= retrieving)}
>>> imaginal
{sentence(word1= Mary, word2= likes, word3= None)}
```

And the cause of the parsing failure is apparent: the retrieval buffer stores a ProperN while 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.

3.5 Top-Down Parsing as an Imperfect Psycholinguistic Model

It is, however, not enough for our parser to correctly parse grammatical sentences and fail for 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, namely syntactic parsing in comprehension-like tasks (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 there are various other properties of our parser that 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, but 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 grammar 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 1983; Abney and Johnson 1991; Resnik 1992).[1] The main reason for this is that our parser generates predictions about syntactic structure exclusively based on the grammar and completely ignores 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. That is, 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, they 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 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.

Thus, 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. The next chapter introduces left-corner parsers and models them in ACT-R and `pyactr`.

[1]Possessives provide typical examples of left-branching structures in English. This is a naturally-occurring example: "You are, officially, my aunt's sixth great-uncle's wife's mother's husband's brother's wife's eighth great-granddaughter." (https://people.com/archive/scoop-vol-82-no-13/).

3.6 Appendix: The Top-Down Parser

All the code discussed in this chapter is available on GitHub as part of the repository https://github.com/abrsvn/pyactr-book. If you want to examine it and run it, install pyactr (see Chap. 1), download the files and run them the same way as any other Python script.

File **ch3_topdown_parser.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch3_topdown_parser.py.