Chapter 4
Syntax as a Cognitive Process: Left-Corner Parsing with Visual and Motor Interfaces

In the previous chapters, we introduced and used several ACT-R modules and buffers: the declarative memory module and the associated retrieval buffer, the procedural memory module and the associated goal buffer, and the imaginal buffer. These are core ACT-R modules, but focusing exclusively on them leads to solipsistic models that do not interact in any way with the environment.

In this chapter, we are going to change that and introduce the vision and motor modules, which give us basic ways to be affected by and, in turn, affect the environment outside the mind. We will then leverage these input/output interfaces when we build a psycholinguistically realistic left-corner parser for the syntactic component of the linguistic representations we will model in this book.

4.1 The Environment in ACT-R: Modeling Lexical Decision Tasks

We will introduce ACT-R environments by modeling a simple lexical decision task. Modeling lexical decision tasks is a good stepping stone towards our goal of providing an end-to-end model of self-paced reading. By end-to-end, we mean a model of self-paced reading that includes both syntactic and semantic parsing (and therefore lexical retrieval of both syntactic and semantic information), and that also has

- a suitable vision interface to model the way a human perceives the linguistic input that is incrementally presented on the screen, and
- a suitable motor interface to model the way a human self-paced reader interacts with the keyboard.

In lexical decision tasks, participants perceive a string and decide whether that string is a word in their language. We will build an ACT-R model to simulate human behavior in this type of tasks. The model will search a (virtual) screen, find a letter

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_4

string/word on the screen, and if the word matches its (impoverished) lexicon, it will press the J key on its (virtual) keyboard. Otherwise, it will press the F key.

We start by importing `pyactr` and creating an environment. Currently, the environment is just a (simulated) computer screen, and a pretty basic one at that: only plain text is supported. But that is enough for our purposes throughout this book.

[py1]
```
>>> import pyactr as actr
>>> environment = actr.Environment(focus_position=(0,0))
```

When the class `Environment` is instantiated, we can specify various parameters. Here, we only specify `focus_position`, which indicates the position the eyes focus on when the simulation starts. Two other parameters are: (i) `simulated_screen_size`, which specifies the physical size of the screen we are simulating in cm (default: 50 × 28 cm), and (ii) `viewing_distance`, which specifies the distance between the simulated participants eyes and the screen (default: 50 cm).

Now that the environment is initialized, we can initialize our ACT-R model:

[py2]
```
>>> lex_decision = actr.ACTRModel(
...     environment=environment,
...     automatic_visual_search=False,
...     motor_prepared=True
... )
```

This initialization is similar to what we used before except that this time, we specify environment-related arguments. We state what environment the model/mind is interacting with, and we set `automatic_visual_search` to `False` so that the model does not start searching the environment for input unless we specifically ask it to. Finally, we state that the motor module is prepared.

Setting `motor_prepared` to `False` would signal that we believed the model to be in a situation in which it did not recently use the motor module. This would make sense, for example, if we tried to model the first item in the experiment. But lexical decision tasks are long and repetitive, so it is more realistic to assume that participants have their motor module 'prepared', which means that there is no preparation phase for key presses. Otherwise, the module would need 250 ms before executing any manual action.

The `MODEL_PARAMETERS` attribute lists all the parameters that can be explicitly set when initializing a model, together with their default values. The majority of these parameters will be discussed in this and future chapters.

[py3]
```
>>> lex_decision.MODEL_PARAMETERS
{'subsymbolic': False, 'rule_firing': 0.05, 'latency_factor': 0.1,
 'latency_exponent': 1.0, 'decay': 0.5, 'baselevel_learning': True,
 'optimized_learning': False, 'instantaneous_noise': 0,
 'retrieval_threshold': 0, 'buffer_spreading_activation': {},
 'spreading_activation_restricted': False, 'strength_of_association': 0,
 'association_only_from_chunks': True, 'partial_matching': False,
 'mismatch_penalty': 1, 'activation_trace': False, 'utility_noise': 0,
 'utility_learning': False, 'utility_alpha': 0.2, 'motor_prepared': False,
 'strict_harvesting': False, 'production_compilation': False,
 'automatic_visual_search': True, 'emma': True, 'emma_noise': True,
 'emma_landing_site_noise': False, 'eye_mvt_angle_parameter': 1,
 'eye_mvt_scaling_parameter': 0.01}
```

We can now add the modules we used before. Given that we are simulating a lexical decision task, we also add some words to declarative memory that the model can access and check against the stimuli in the simulated experiment:

[py4]
```
>>> actr.chunktype("goal", "state")                                    1
>>> actr.chunktype("word", "form")                                     2
>>> dm = lex_decision.decmem                                           3
>>> for string in {"elephant", "dog", "crocodile"}:                    4
...     dm.add(actr.makechunk(typename="word", form=string))           5
...                                                                    6
>>> g = lex_decision.goal                                              7
>>> g.add(actr.makechunk(nameofchunk="beginning",                      8
...                      typename="goal",                             9
...                      state="start"))                              10
```

We add three words to our declarative memory using a Python `for` loop (lines 4–6 in [py4]) rather than adding them one at a time (needlessly verbose and tedious). This way of adding chunks to memory can save a lot of time if we want to add a lot of elements, e.g., a reasonably sized lexicon. We also add a chunk into the goal buffer that will get our lexical decision simulation started (lines 8–10).

4.1.1 The Visual Module

The visual module allows the ACT-R model to 'see' the environment. This interaction happens via two buffers: `visual_location` searches the environment for elements matching its search criteria, and `visual` stores the element found using `visual_location`. The two buffers are sometimes called the visual *Where* and *What* buffers.

The visual *Where* buffer searches the environment (the screen) and outputs the location of an element on the screen that matches some search criteria. Visual search cues have three possible slots: `color`, `screen_x` (the horizontal position on the screen) and `screen_y` (the vertical position on the screen). The x and y positions can be specified in precise terms, e.g., find an element at location `screen_x 100 screen_y 100`, where the numbers represent pixels. Or we could specify the location of an element only approximately: a `screen_x <100` cue would search for elements at screen locations at most 100 pixels from the left edge of the screen, and a `screen_x >100` cue would search for elements on the complementary side of the screen.

Three other values are possible for the `screen_x` and `screen_y` slots:

- `screen_x lowest` searches for the element with the lowest position on the horizontal axis (the element closest to the left edge);
- `screen_x highest` searches for the element with the highest position on the same axis (the element closest to the right edge);
- finally, `screen_x closest` searches for the closest element to the current focus position (the axis is actually ignored in this case).

The same applies to the `screen_y` slot.

The visual *What* buffer stores the element whose location was identified by the *Where* buffer. The *What* buffer is therefore accessed after the *Where* buffer, as we will see when we state the production rules for our lexical decision model.

The vision module as a whole is an implementation of EMMA (Eye Movements and Movement of Attention, Salvucci 2001), which in turn is a generalization and simplification of the E-Z Reader model (Reichle et al. 1998). While the latter model is used for reading, the EMMA model attempts to simulate any visual task, not just reading. For detailed discussions of these models and their empirical coverage, see Reichle et al. (1998), Salvucci (2001), Staub (2011).

4.1.2 The Motor Module

The motor module is limited to the simulation of a key press on the keyboard—or typing, if multiple key strokes are chained. The ACT-R typing model is based on EPIC's Manual Motor Processor (Meyer and Kieras 1997). It has one buffer that accepts requests to execute motor commands.

The ACT-R motor module currently implemented in `pyactr` is more limited, it currently supports only one command: `press_key`. But this should suffice for simulations of many experimental tasks used in (psycho)linguistics, including lexical decision tasks, self-paced reading, forced-choice tasks etc. All of these tasks commonly require only basic keyboard interaction on the participants' part (or mouse button presses, which we subsume under keyboard interaction).

The hands of the ACT-R model are assumed to be positioned in the home row on a standard (US) English keyboard, with index fingers at F and J. The model assumes a competent, albeit not expert, typist.

4.2 The Lexical Decision Model: Productions

We only need five productions to model our lexical decision task. The first rule requires the visual *Where* buffer to search the (virtual) screen and find the closest word relative to the starting (0, 0) position.

[py5]
```
>>> lex_decision.productionstring(name="find word", string="""
...     =g>
...     isa     goal
...     state   start
...     ?visual_location>
...     buffer  empty
...     ==>
...     =g>
...     isa     goal
...     state   attend
...     +visual_location>
...     isa _visuallocation
...     screen_x closest
... """)
{'=g': goal(state= start), '?visual_location': {'buffer': 'empty'}}
```

```
==>                                                                    16
{'=g': goal(state= attend), '+visual_location': _visuallocation(color= ,  17
 screen_x= closest, screen_y= , value= )}                              18
```

The rule requires the `start` chunk to be in the goal buffer (lines 2–4 in **[py5]**) and the visual location buffer to be empty (lines 5–6). If these preconditions are met, we enter a new goal state of 'attending' to the visual input (lines 8–10) and the visual location buffer will search for and be updated with the position of the closest element (lines 11–13).

Note that the search is done by specifying `+visual_location`, that is, the name of the buffer and the + operation. We used + before in connection to the goal and retrieval buffers, where + signals that a new chunk is added to these buffers. Furthermore, in the case of the retrieval buffer, the addition of a chunk automatically triggers a memory recall. For the visual *Where* buffer, the + operation triggers a similar action: a chunk is added that automatically starts a search, the only difference being that the visual *Where* buffer automatically starts searching the environment, while the retrieval buffer starts searching the declarative memory.

Once this rule fires, our ACT-R model will know the position of the closest element on the screen, but it won't know which element is actually present at that location. To access the element, we make use of the visual *What* buffer, as shown in the `"attend word"` rule below.

[py6]

```
>>> lex_decision.productionstring(name="attend word", string="""          1
...     =g>                                                            2
...     isa     goal                                                   3
...     state   attend                                                 4
...     =visual_location>                                              5
...     isa     _visuallocation                                        6
...     ?visual>                                                       7
...     state   free                                                   8
...     ==>                                                            9
...     =g>                                                            10
...     isa     goal                                                   11
...     state   retrieving                                             12
...     +visual>                                                       13
...     isa     _visual                                                14
...     cmd     move_attention                                         15
...     screen_pos =visual_location                                    16
...     ~visual_location>                                              17
... """)                                                               18
{'=g': goal(state= attend), '=visual_location': _visuallocation(color= ,  19
 screen_x= , screen_y= , value= ), '?visual': {'state': 'free'}}       20
==>                                                                    21
{'=g': goal(state= retrieving), '+visual': _visual(cmd= move_attention,  22
 color= , screen_pos= =visual_location, value= ),                      23
 '~visual_location': None}                                             24
```

This rule checks that the visual *Where* buffer has stored a location (lines 5–6) and that the visual *What* buffer is free, i.e., it is not carrying out any visual action. If these preconditions are satisfied, a new chunk is added to the visual *What* buffer that moves the focus of attention to the current visual location (lines 13–16). The attention focus is moved by setting the value of the `cmd` (command) slot to `move_attention`. In addition, the goal enters a `retrieving` state (lines 10–12) and the visual *Where* buffer (a.k.a. `visual_location`) is cleared (line 17).

The interaction between the two vision buffers simulates a two-step process: (i) noticing an object through the visual location (*Where*) buffer, and (ii) finding what that object is, i.e., attending to the object through the visual (*What*) buffer.

The next rule starts the memory retrieval process: we take the `value =val` of the chunk stored in the visual (*What*) buffer, which is a string, and check to see if there is a word in our lexicon that has that form. This retrieval request is actually the core part of our lexical decision model. The crucial parts of the rule are on lines 7 and 14 in [py7] below: the character string `=val` of the perceived chunk (line 7) becomes the declarative memory cue placed in the retrieval buffer (line 14).

[py7]
```
>>> lex_decision.productionstring(name="retrieving", string="""
...     =g>
...     isa     goal
...     state   retrieving
...     =visual>
...     isa     _visual
...     value   =val
...     ==>
...     =g>
...     isa     goal
...     state   retrieval_done
...     +retrieval>
...     isa     word
...     form    =val
... """)
{'=g': goal(state= retrieving), '=visual': _visual(cmd= , color= ,
 screen_pos= , value= =val)}
==>
{'=g': goal(state= retrieval_done), '+retrieval': word(form= =val)}
```

The final two rules we need are provided below. They consider the two possible outcomes of the retrieval process: (i) a lexeme was retrieved—the rule in [py8], or (ii) no lexeme was found with that form—the rule in [py9].

[py8]
```
>>> lex_decision.productionstring(name="lexeme retrieved", string="""
...     =g>
...     isa     goal
...     state   retrieval_done
...     ?retrieval>
...     buffer  full
...     state   free
...     ==>
...     =g>
...     isa     goal
...     state   done
...     +manual>
...     isa     _manual
...     cmd     press_key
...     key     J
... """)
{'=g': goal(state= retrieval_done),
 '?retrieval': {'buffer': 'full', 'state': 'free'}}
==>
{'=g': goal(state= done), '+manual': _manual(cmd= press_key, key= J)}
```

[py9]
```
>>> lex_decision.productionstring(name="no lexeme found", string="""
...     =g>
...     isa     goal
...     state   retrieval_done
...     ?retrieval>
...     buffer  empty
...     state   error
...     ==>
...     =g>
```

```
...     isa     goal                                                    10
...     state   done                                                    11
...     +manual>                                                        12
...     isa     _manual                                                 13
...     cmd     press_key                                               14
...     key     F                                                       15
... """)                                                                16
{'=g': goal(state= retrieval_done),                                     17
 '?retrieval': {'buffer': 'empty', 'state': 'error'}}                   18
==>                                                                     19
{'=g': goal(state= done), '+manual': _manual(cmd= press_key, key= F)}   20
```

The format of the rules should look familiar by now. The only new parts are on lines 12–15 (in both [py8] and [py9]). These lines set the motor module in action, which can accept only one command, namely pressing a key. This is implemented by placing a chunk of a special predefined type `_manual` in the `manual` buffer.

The chunk has two slots: `cmd` (what command should be carried out) and `key` (what key should be pressed). The command is the same for both rules (`press_key` on line 14 in both [py8] and [py9]), but the key to be pressed is different. If a lexeme is found, the ACT-R model simulates a human participant and presses `'J'`. Otherwise, the model presses `'F'` (line 15 in both [py8] and [py9]).

4.3 Running the Lexical Decision Model and Understanding the Output

Before we run the simulation of the model, we have to specify the set of stimuli (character strings) that should appear on the screen. We use a dictionary data structure for that (a dictionary is basically a partial variable assignment). The dictionary data structure is represented in Python using curly brackets `{}` and it consists of ⟨key, value⟩ pairs, i.e., ⟨variable, value⟩ pairs. Values can themselves be dictionaries.

In [py10], we specify that our first—and only—stimulus is the word *elephant*, which should be displayed on the screen starting at pixel ⟨320, 180⟩.

[py10]
```
>>> word = {1: {'text': 'elephant', 'position': (320, 180)}}   1
```

We are now ready to initialize the simulation:

[py11]
```
>>> lex_dec_sim = lex_decision.simulation(                                  1
...     realtime=True,                                                      2
...     gui=False,                                                          3
...     environment_process=environment.environment_process,                4
...     stimuli=word,                                                       5
...     triggers='',                                                        6
...     times=1)                                                            7
```

The first parameter, namely `realtime` (line 2 in [py11]), states that the simulation should appear in real time. Setting this parameter to `True` ensures that the simulation will take the same amount of real time as the model predicts (otherwise, the simulation is executed as fast as the processing power of the computer allows it). Note that this does not affect the actual model and its predictions in any way, it only affects the way the simulation is displayed.

The second parameter `gui` (line 3) specifies whether a graphical user interface should be started in a separate window to represent the environment, i.e., the virtual screen on which the stimuli are displayed. This option is switched off here, but feel free to switch it on by setting `gui` to `True`.

The third argument (line 4) states what environment process should appear in our environment. You can create your own, but there is one predefined in the `Environment` class that displays stimuli from the list in **[py10]** one at a time on the virtual screen.

The stimulus list to be displayed in the environment is specified by the fourth parameter (line 5 of **[py11]**).

The final two parameters are `triggers` (line 6), which specifies the triggers that the process should respond to (we do not have any triggers here, so we leave that list empty), and `times` (line 7), which specifies that each stimulus should be displayed for 1 s.

The simulation can now be run:

[py12]
```
>>> lex_dec_sim.run()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: find word')
****Environment: {1: {'text': 'elephant', 'position': (320, 180)}}
(0.05, 'PROCEDURAL', 'RULE FIRED: find word')
(0.05, 'g', 'MODIFIED')
(0.05, 'visual_location', 'CLEARED')
(0.05, 'visual_location', "ENCODED LOCATION:'_visuallocation(color= None,
      screen_x= 320, screen_y= 180, value= None)'")
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: attend word')
(0.1, 'PROCEDURAL', 'RULE FIRED: attend word')
(0.1, 'g', 'MODIFIED')
(0.1, 'visual_location', 'CLEARED')
(0.1, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION STARTED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'NO RULE FOUND')
(0.1127, 'visual', 'CLEARED')
(0.1127, 'visual', "ENCODED VIS OBJECT:'_visual(cmd= move_attention,
        color= , screen_pos= _visuallocation(color= None, screen_x= 320,
        screen_y= 180, value= None), value= elephant)'")
(0.1127, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1127, 'PROCEDURAL', 'RULE SELECTED: retrieving')
(0.1627, 'PROCEDURAL', 'RULE FIRED: retrieving')
(0.1627, 'g', 'MODIFIED')
(0.1627, 'retrieval', 'START RETRIEVAL')
(0.1627, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1627, 'PROCEDURAL', 'NO RULE FOUND')
(0.2127, 'retrieval', 'CLEARED')
(0.2127, 'retrieval', 'RETRIEVED: word(form= elephant)')
(0.2127, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2127, 'PROCEDURAL', 'RULE SELECTED: lexeme retrieved')
(0.253, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION COMPLETED')
(0.2627, 'PROCEDURAL', 'RULE FIRED: lexeme retrieved')
(0.2627, 'g', 'MODIFIED')
(0.2627, 'manual', 'COMMAND: press_key')
(0.2627, 'manual', 'PREPARATION COMPLETE')
(0.2627, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2627, 'PROCEDURAL', 'NO RULE FOUND')
(0.3127, 'manual', 'INITIATION COMPLETE')
(0.3127, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3127, 'PROCEDURAL', 'NO RULE FOUND')
(0.3815, 'visual', 'SHIFT COMPLETE TO POSITION: [320, 180]')
(0.3815, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3815, 'PROCEDURAL', 'NO RULE FOUND')
(0.4127, 'manual', 'KEY PRESSED: J')
(0.4127, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4127, 'PROCEDURAL', 'NO RULE FOUND')
```

```
(0.5627, 'manual', 'MOVEMENT FINISHED')                                    49
(0.5627, 'PROCEDURAL', 'CONFLICT RESOLUTION')                              50
(0.5627, 'PROCEDURAL', 'NO RULE FOUND')                                    51
```

Let us first consider the general picture that the temporal trace of our simulation paints. In this model, we see that it should take a bit more than 400 ms to find a stimulus, decide whether it is a word and press the right key—see the event `'KEY PRESSED: J'` on line 46 in **[py12]** above.

This is slightly faster than the 500–600 ms usually found in lexical decision tasks (Forster 1990a; Murray and Forster 2004), but it is a consequence of our inadequate modeling of memory retrieval. That is, while visual and motor processes are fairly realistically modeled, we assume retrieval always takes 50 ms regardless of the specific features of the word we're trying to retrieve and the cognitive state in which retrieval happens. We will address this in Chap. 6 when we start introducing the subsymbolic components of ACT-R, and in Chap. 7 where we use them to build much more realistic models of lexical decision tasks.

The remainder of this section is dedicated to discussing the visual and manual processes that are chronicled in the output of the simulation in **[py12]**.

4.3.1 Visual Processes in Our Lexical Decision Model

Traditionally, visual attention is equated to (keeping track of) the focus position of the eyes (e.g., Just and Carpenter 1980; Just et al. 1982): understanding which word one attends to is tantamount to identifying which word the eyes are focused on. But this identification of the unobservable cognitive state (attention) and overt behavior (eye focus position) is an overly simplified model. For example, it is known that when people read, some words —especially high-frequency ones—are processed without ever receiving eye focus (Schilling et al. 1998; Rayner 1998 among others).

The EMMA model (Salvucci 2001) incorporated in ACT-R and implemented in `pyactr` captures this by disassociating eye focus and attention: the two processes are related but not identical. In particular, a shift of attention to a visual object, for example, the command `move_attention` on line 15 in **[py6]** above, triggers:

i. an immediate attempt to encode the object as an internal representation and, at the same time,
ii. eye movement.

But the two processes proceed independently of each other.

We first discuss the process of encoding a visual object. The time t_{enc} needed to encode an object is modeled using a gamma distribution (a generalization of the exponential distribution) with mean T_{enc} and standard deviation one third of the mean. Note that the mean T_{enc} in (1–2) below is crucially parametrized by the distance d between the current eye focus position and the position of the target object:

(1) $t_{enc} \sim Gamma(\mu = T_{enc}, \sigma = T_{enc}/3)$[1]

(2) $T_{enc} = K \cdot (-\log f) \cdot e^{kd}$, where:

- f is the (normalized) frequency of the object (word) being encoded;
- d is the distance between the current focal point of the eyes and the object to be encoded measured in degrees of visual angle (d is the eccentricity of the object relative to the current eye position);
- k is a free parameter, scaling the effect of distance (set to 1 by default);
- K is a free parameter, scaling the encoding time itself (set to 0.01 by default).

In the trace of the simulation in **[py12]**, the time point of encoding a visual object is signaled by the event `ENCODED VIS OBJECT` (lines 19–21).

Let us turn now to discussing the eye movement process. The time needed for eye movement to the new object is split into two sub-processes: preparation and execution. The preparation is modeled once again as a Gamma distribution with mean 135 ms and a standard deviation of 45 ms (yet again, the standard deviation is one third of the mean).

The execution, which follows the preparation, is also modeled as a Gamma distribution with:

- a mean of 70 ms + 2 ms for every degree of visual angle between the current eye position and the targeted visual object, and
- a standard deviation that is one third of the mean.

It is only at the end of the execution sub-process that the eyes focus on the new position. Thus, the whole process of eye movement takes around 200 ms (≈135 + 70), which corresponds to average saccade latencies reported in previous studies (see, e.g., Fuchs 1971).

In our simulation **[py12]**, the event `PREPARATION TO SHIFT VISUAL ATTENTION COMPLETED` (line 33) signals the end of the preparation phase, and the end of the execution phase is signaled by `SHIFT COMPLETE TO POSITION [320, 180]` (line 43). It is only at this point that the eyes focus on the new location, but the internal representation of the object has already been encoded: the word has already been retrieved from memory by this point, as indicated by the earlier event `RETRIEVED: word(form= elephant)` (line 30).

How do the two processes of visual encoding and eye movement interact? One possibility is that encoding is done before the end of the preparation phase—this is actually the case in **[py12]**. When this happens, the planned eye movement can be canceled, but only if the cognitive processes following visual encoding are fast enough to cancel the eye shift or request a new eye-focus position before the end of the preparation phase.

---

[1] Gamma distributions are usually parametrized in terms of a shape α and a rate β or scale $\frac{1}{\beta}$. We can convert our non-standard parametrization into the standard one(s) as follows: shape $\alpha = (\frac{\mu}{\sigma})^2$ and rate $\beta = \frac{\mu}{\sigma^2}$ (equivalently: scale $\frac{1}{\beta} = \frac{\sigma^2}{\mu}$).

The second possibility is that visual encoding is finished only during the execution phase of the eye movement process. In that case, the eye movement cannot be stopped anymore and the eye shift is actually carried out.

The third and final possibility is that visual encoding is still not done after the eyes shift to a new position. In that case, visual encoding is restarted and since the eyes have moved closer to the position of the object we're trying to encode, the time necessary for visual encoding is now decreased.

To understand how the restarted visual encoding time is decreased, consider what the new encoding time would have been if this had been an initial visual encoding. We would have a random draw t'_{enc} from a Gamma distribution centered at a *new* mean T'_{enc}, because the distance between the object and the new position of the eyes has now changed to d':

(3) $t'_{enc} \sim Gamma(\mu = T'_{enc}, \sigma = T'_{enc}/3)$

(4) $T'_{enc} = K \cdot (-\log f) \cdot e^{kd'}$

But instead of taking the full t'_{enc} time to do the visual encoding, we will scale that down by the amount of time we already spent during our initial encoding attempt. Specifically, we will look at the initial expected encoding time t_{enc} and at the time $t_{completed}$ that we actually spent encoding. Note that necessarily, $t_{completed} < t_{enc}$. We can therefore say that we have already completed a percentage of the visual encoding process, and that percentage is $\frac{t_{completed}}{t_{enc}}$.

The new processing time should be the remaining percentage that we have not completed yet, i.e., $\frac{t_{enc}-t_{completed}}{t_{enc}}$ or equivalently $1 - \frac{t_{completed}}{t_{enc}}$. Thus, instead of saying that the new encoding time is the full t'_{enc}, we will only need the percentage of it that is the same as the percentage of incomplete processing we had left after our first encoding attempt:

(5) Visual reencoding time: $\left(1 - \frac{t_{completed}}{t_{enc}}\right) \cdot t'_{enc}$

This completes our brief introduction to visual processes in ACT-R/`pyactr`. For more details, see Salvucci (2001).

4.3.2 Manual Processes in Our Lexical Decision Model

Similarly to the vision process, the motor process is split into several sub-phases when carrying out a command: the preparation phase, the initiation phase, the actual key press and finishing the movement (returning to the original position). As in the case of the visual module, cognitive processes can interrupt a movement, but only during the preparation phase.

The time needed to carry out every phase is dependent on several variables:

- Is this the first movement or not? If a key was pressed before, was it pressed with the same hand or not? Answers to these questions influence the amount of time the preparation phase takes.

- Is the key to be pressed on the home row or not? The answer to this question influences the amount of time the actual movement requires, as well as the preparation phase.

We will not discuss here the details of the ACT-R/`pyactr` model of motor process—see Meyer and Kieras (1997) for a detailed presentation.

4.4 A Left-Corner Parser with Visual and Motor Interfaces

In this section, we introduce a left-corner parser that incorporates visual and motor interfaces. The left-corner parser builds on the basic top-down parser introduced in the previous chapter and on the lexical decision model with visual and motor interfaces introduced in this chapter.

As discussed at the end of the previous chapter, 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 NP in the sentence *Mary sleeps*, which is described as having evidence for the *left corner* of the `S` → `NP VP` rule, is sufficient to trigger it.

Following Hale (2014, Chap. 3), we summarize the left-corner parsing strategy in the 'project' and 'project and complete' rules below (see (6a) and (6b)). 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. Finally, the 'shift' rule in (6c) takes words one at a time from the incoming string of words and adds them to the top of the stack to be parsed.

(6) Left-corner parsing rule schemata (Hale 2014, Chap. 3):

   a. **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.

   b. **Project and 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.

c. **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 (1992), who argues that projection and 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 and 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 will have introduced 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.

[py13]
```
>>> import pyactr as actr                                                    1
>>> environment = actr.Environment(focus_position=(320, 180))               2
```

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, befitting the kind of information the imaginal buffer stores;
- finally, `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.

[py14]
```
>>> actr.chunktype("parsing_goal", "task stack_top stack_bottom\             1
...                                 parsed_word right_frontier")             2
>>> actr.chunktype("parse_state", "node_cat mother daughter1\                3
...                               daughter2 lex_head")                       4
>>> actr.chunktype("word", "form cat")                                       5
```

The `parsing_goal` chunk type in **[py14]** has the same slots as the type we used for the top-down parser discussed in Chap. 3, 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 and the `VP` node for object NPs. Specifically, whenever we will 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.

Let's now turn to the `parse_state` chunk type in **[py14]**. These intermediate parsing states are stored in the imaginal buffer, and record the progress of the parsing cognitive process. In particular:

- 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;
- 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 or uniquely identifiable time stamps for specific nodes in the tree. This is particularly constraining for phrases like NPs, which are repeatedly instantiated in a given structure. Their position in the hierarchical structure can be identified only if we encode additional information in their corresponding chunks.

Specifically, 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`). In **[py15]**, 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, and

the 200 ms encoding delay provides a better fit to the reaction time data associated with those processes.

In contrast, we believe that a low-delay or even 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.

[py15]
```
>>> 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, as shown in **[py16]** below. We also specify the goal chunk needed to get the parsing process started: the initial goal is to read the first word (line 18: the `task` is `read_word`), and to try to parse the whole sentence (line 19: `stack_top` is `S`).

[py16]
```
>>> 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
... """))
>>> g.add(actr.chunkstring(string="""
...     isa             parsing_goal
...     task            read_word
...     stack_top       S
...     right_frontier  S
... """))
```

With the lexicon in place, we can start specifying the production rules. Our first rule is the `"press spacebar"` rule below. This rule initializes the actions needed to read a word: if the task is `read_word` (line 4 in **[py17]**), the top of the stack is not empty (line 5), that is, we have some parsing goals left to accomplish, and the motor module is `free` (not currently busy), then we should press the space bar to display the next word.

[py17]
```
>>> parser.productionstring(name="press spacebar", string="""
...     =g>
...     isa             parsing_goal
...     task            read_word
...     stack_top       ~None
...     ?manual>
...     state           free
...     ==>
...     =g>
...     isa             parsing_goal
```

```
...      task            encode_word                                          11
...      +manual>                                                             12
...      isa             _manual                                              13
...      cmd             'press_key'                                          14
...      key             'space'                                              15
... """)                                                                      16
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,          17
 stack_top= ~None, task= read_word), '?manual': {'state': 'free'}}            18
==>                                                                           19
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,          20
                    stack_top= , task= encode_word),                          21
 '+manual': _manual(cmd= press_key, key= space)}                              22
```

Assuming the next word was displayed and the visual module retrieved its form, we trigger the `"encode word"` rule below, which gets the current value stored in the `visual` buffer, stores it in the goal buffer as the current `parsed_word`, and initializes a new `get_word_cat` task.

[py18]

```
>>> parser.productionstring(name="encode word", string="""                    1
...      =g>                                                                  2
...      isa             parsing_goal                                         3
...      task            encode_word                                          4
...      =visual>                                                             5
...      isa             _visual                                              6
...      value           =val                                                 7
...      ==>                                                                  8
...      =g>                                                                  9
...      isa             parsing_goal                                         10
...      task            get_word_cat                                         11
...      parsed_word    =val                                                  12
...      ~visual>                                                             13
... """)                                                                      14
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,          15
                    stack_top= , task= encode_word),                          16
 '=visual': _visual(cmd= , color= , screen_pos= , value= =val)}               17
==>                                                                           18
{'=g': parsing_goal(parsed_word= =val, right_frontier= , stack_bottom= ,      19
 stack_top= , task= get_word_cat), '~visual': None}                           20
```

The `get_word_cat` task consists of placing a retrieval request for a lexical item stored in declarative memory. As the rule `"retrieve category"` in **[py19]** below shows, the retrieval cue consists of the form/value we got from the `visual` buffer. While we wait for the result of this retrieval request, we enter a new `retrieving_word` task.

[py19]

```
>>> parser.productionstring(name="retrieve category", string="""              1
...      =g>                                                                  2
...      isa             parsing_goal                                         3
...      task            get_word_cat                                         4
...      parsed_word     =w                                                   5
...      ==>                                                                  6
...      +retrieval>                                                          7
...      isa             word                                                 8
...      form            =w                                                   9
...      =g>                                                                  10
...      isa             parsing_goal                                         11
...      task            retrieving_word                                      12
... """)                                                                      13
{'=g': parsing_goal(parsed_word= =w, right_frontier= , stack_bottom= ,        14
                    stack_top= , task= get_word_cat)}                         15
==>                                                                           16
{'+retrieval': word(cat= , form= =w),                                         17
 '=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,          18
                    stack_top= , task= retrieving_word)}                      19
```

If we are in a `retrieving_word` task and the declarative memory retrieval was successfully completed, which we know because the retrieved word is in the `retrieval` buffer, we can start building some syntactic structure, i.e., we can *sensu stricto* parse. The first parsing action is `"shift and project word"`, in **[py20]** below. This means that the syntactic category of the retrieved word is pushed onto the top of the stack (pushing to the bottom of the stack whatever was previously on top), and storing a new `parse_state` in the `imaginal` buffer. The parse state is a unary branching tree with the syntactic category of the retrieved word as the mother/root node and the phonological form of the word as the only daughter. We also enter a new `parsing` task, in which we see if we can trigger any other parsing, i.e., syntactic structure building, rules.

[py20]
```
>>> parser.productionstring(name="shift and project word", string="""
...     =g>
...     isa             parsing_goal
...     task            retrieving_word
...     stack_top       =t
...     stack_bottom    None
...     =retrieval>
...     isa             word
...     form            =w
...     cat             =c
...     ==>
...     =g>
...     isa             parsing_goal
...     task            parsing
...     stack_top       =c
...     stack_bottom    =t
...     +imaginal>
...     isa             parse_state
...     node_cat        =c
...     daughter1       =w
...     ~retrieval>
... """)
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None,
                    stack_top= =t, task= retrieving_word),
 '=retrieval': word(cat= =c, form= =w)}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= =t,
 stack_top= =c, task= parsing), '+imaginal': parse_state(daughter1= =w,
 daughter2= , lex_head= , mother= , node_cat= =c), '~retrieval': None}
```

We now reached that point in our parser specification when we simply encode all the grammar rules into parsing rules. The first two rules, listed in **[py21]** and **[py22]** below, project an NP node on top of a ProperN node. NP projection comes in two flavors depending on whether we are expecting an NP at the time we try to project one, or not.

Consider first the case in which we do not expect an NP, that is, rule `"project: NP ==> ProperN"` in **[py21]** below. This rule is triggered if the top of our stack is a ProperN and the bottom of our stack is not an NP. That is, we do not expect an NP at this time (`~NP` on line 5 in **[py21]** below). If this is our current parsing goal, then we will pop the ProperN category off the stack, replace it with an NP category and add the newly built structure to the `imaginal` buffer. This newly built structure is a unary branching NP node with ProperN as its only daughter. The NP node is in its turn attached to whatever the current right frontier `=rf` is, and it is indexed with the lexical head that projected the ProperN node in a previous parsing step.

[py21]
```
>>> parser.productionstring(name="project: NP ==> ProperN", string="""
...     =g>
...     isa             parsing_goal
...     stack_top       ProperN
...     stack_bottom    ~NP
...     right_frontier  =rf
...     parsed_word     =w
...     ==>
...     =g>
...     isa             parsing_goal
...     stack_top       NP
...     +imaginal>
...     isa             parse_state
...     node_cat        NP
...     daughter1       ProperN
...     mother          =rf
...     lex_head        =w
... """)
{'=g': parsing_goal(parsed_word= =w, right_frontier= =rf, stack_bottom= ~NP,
                    stack_top= ProperN, task= )}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,
 stack_top= NP, task= ), '+imaginal': parse_state(daughter1= ProperN,
 daughter2= , lex_head= =w, mother= =rf, node_cat= NP)}
```

The second case we consider is an NP projection on top of a ProperN when an NP node is actually expected, as shown in rule `"project and complete: NP ==> ProperN"` below. That is, the current parsing goal has a ProperN at the top of the stack and an NP right below it (at the bottom of the stack). If that is the case, we pop both the ProperN and the NP category off the stack (lines 14–15 in **[py22]**), add the relevant unary-branching NP structure to the `imaginal` buffer, and reenter a `read_word` task.

[py22]
```
>>> parser.productionstring(
...     name="project and complete: NP ==> ProperN",
...     string="""
...         =g>
...         isa             parsing_goal
...         stack_top       ProperN
...         stack_bottom    NP
...         right_frontier  =rf
...         parsed_word     =w
...         ==>
...         =g>
...         isa             parsing_goal
...         task            read_word
...         stack_top       None
...         stack_bottom    None
...         +imaginal>
...         isa             parse_state
...         node_cat        NP
...         daughter1       ProperN
...         mother          =rf
...         lex_head        =w
... """)
{'=g': parsing_goal(parsed_word= =w, right_frontier= =rf, stack_bottom=
                    NP, stack_top= ProperN, task= )}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None,
 stack_top= None, task= read_word), '+imaginal': parse_state(daughter1=
ProperN, daughter2= , lex_head= =w, mother= =rf, node_cat= NP)}
```

Now that we implemented the NP projection rules, we can turn to the S and VP grammar rules, implemented in **[py23]** and **[py24]** below. Both of these rules are project-and-complete rules because, in both cases, we have an expectation for the

mother node. We expect an S because that is the default starting goal of all parsing-model runs. And we expect a VP because the `"project and complete: S ==> NP VP"` rule in [py23] always adds a VP expectation to the stack.

The project-and-complete S rule in [py23] is triggered after we have already parsed the subject NP, which is sitting at the top of the stack (line 6), and we have an S expectation right below the NP. If that is the case, we pop both categories off the stack and add an expectation for a VP at the top of the stack (lines 12–13). We also reenter the `read_word` task (line 11), and introduce the expected VP node as the current right frontier that the object NP will attach to (line 14). Finally, as expected, we add the newly built syntactic structure to the `imaginal` buffer: this is a binary-branching structure with S as the mother/root node and NP and VP as the daughters (in that order; lines 17–19).

[py23]
```
>>> parser.productionstring(                                                    1
...     name="project and complete: S ==> NP VP",                               2
...     string="""                                                              3
...         =g>                                                                 4
...         isa             parsing_goal                                        5
...         stack_top       NP                                                  6
...         stack_bottom    S                                                   7
...         ==>                                                                 8
...         =g>                                                                 9
...         isa             parsing_goal                                        10
...         task            read_word                                           11
...         stack_top       VP                                                  12
...         stack_bottom    None                                                13
...         right_frontier  VP                                                  14
...         +imaginal>                                                          15
...         isa             parse_state                                         16
...         node_cat        S                                                   17
...         daughter1       NP                                                  18
...         daughter2       VP                                                  19
... """)                                                                        20
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= S,           21
                    stack_top= NP, task= )}                                     22
==>                                                                             23
{'=g': parsing_goal(parsed_word= , right_frontier= VP, stack_bottom= None,      24
 stack_top= VP, task= read_word), '+imaginal': parse_state(daughter1= NP,       25
 daughter2= VP, lex_head= , mother= , node_cat= S)}                             26
```

The `"project and complete: VP ==> V NP"` rule in [py24] below is very similar to the project-and-complete S rule. This rule is triggered if we have just parsed a verb V, which is sitting at the top of the stack (line 7), and we have an expectation for a VP right below it (line 8). If that is the case, we pop both categories off the stack and introduce a new expectation for the object NP at the top of the stack (lines 13–14), reenter the `read_word` task (line 12) and store the newly built binary-branching VP structure in the `imaginal` buffer (lines 17–19).

[py24]
```
>>> parser.productionstring(                                                    1
...     name="project and complete: VP ==> V NP",                               2
...     string="""                                                              3
...         =g>                                                                 4
...         isa             parsing_goal                                        5
...         task            parsing                                             6
...         stack_top       V                                                   7
...         stack_bottom    VP                                                  8
...         ==>                                                                 9
...         =g>                                                                 10
...         isa             parsing_goal                                        11
...         task            read_word                                           12
...         stack_top       NP                                                  13
```

```
...          stack_bottom    None                                                    14
...          +imaginal>                                                              15
...          isa             parse_state                                             16
...          node_cat        VP                                                      17
...          daughter1       V                                                       18
...          daughter2       NP                                                      19
... """)                                                                             20
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= VP,               21
                    stack_top= V, task= parsing)}                                    22
==>                                                                                  23
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None,             24
 stack_top= NP, task= read_word), '+imaginal': parse_state(daughter1= V,             25
 daughter2= NP, lex_head= , mother= , node_cat= VP)}                                 26
```

We have now implemented all the parsing rules corresponding to the grammar rules listed in Chap. 3, example (1). The final rule we need is a wrap-up rule that ends the parsing process if our to-parse stack is empty, i.e., we have no categories to parse at the top of the stack (line 5 in [py25] below). If that is the case, we simply flush the `g` (goal) and `imaginal` buffers, which empties their contents into declarative memory.

[py25]
```
>>> parser.productionstring(name="finished", string="""                            1
...     =g>                                                                         2
...     isa             parsing_goal                                                3
...     task            read_word                                                   4
...     stack_top       None                                                        5
...     ==>                                                                         6
...     ~g>                                                                         7
...     ~imaginal>                                                                  8
... """)                                                                            9
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,                10
                    stack_top= None, task= read_word)}                              11
==>                                                                                 12
{'~g': None, '~imaginal': None}                                                     13
```

Let us now run the left-corner parser on the sentence *Mary likes Bill* and examine its output. As shown by the `stimuli` variable in [py26] below, the sentence is presented self-paced reading style, with the words displayed one at a time in the center of the (virtual) screen (lines 1–3). We also specify that the simulation should be run for 1.5 s (the default time is 1 s, which would not be enough in this case).

[py26]
```
>>> stimuli = [{1: {'text': 'Mary', 'position': (320, 180)}},                       1
...            {1: {'text': 'likes', 'position': (320, 180)}},                     2
...            {1: {'text': 'Bill', 'position': (320, 180)}}]                      3
>>> parser_sim = parser.simulation(                                                4
...     realtime=True,                                                             5
...     gui=False,                                                                 6
...     environment_process=environment.environment_process,                       7
...     stimuli=stimuli,                                                           8
...     triggers='space')                                                          9
>>> parser_sim.run(max_time=1.5)                                                   10
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                         11
(0, 'PROCEDURAL', 'RULE SELECTED: press spacebar')                               12
****Environment: {1: {'text': 'Mary', 'position': (320, 180)}}                   13
(0, 'visual_location', 'ENCODED LOCATION: _visuallocation(color= ,               14
    screen_x= 320, screen_y= 180, value= )')                                     15
(0.007, 'visual', 'AUTOMATIC BUFFERING: _visual(cmd= , color= ,                  16
        screen_pos= _visuallocation(color= , screen_x= 320, screen_y= 180,       17
        value= ), value= Mary)')                                                 18
(0.05, 'PROCEDURAL', 'RULE FIRED: press spacebar')                               19
(0.05, 'g', 'MODIFIED')                                                          20
(0.05, 'manual', 'COMMAND: press_key')                                           21
(0.05, 'manual', 'PREPARATION COMPLETE')                                         22
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                      23
(0.05, 'PROCEDURAL', 'RULE SELECTED: encode word')                               24
```

```
(0.1, 'manual', 'INITIATION COMPLETE')
(0.1, 'PROCEDURAL', 'RULE FIRED: encode word')
(0.1, 'g', 'MODIFIED')
(0.1, 'visual', 'CLEARED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve category')
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve category')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'manual', 'KEY PRESSED: SPACE')
(0.2, 'retrieval', 'CLEARED')
****Environment: {1: {'text': 'likes', 'position': (320, 180)}}
(0.2, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Mary)')
(0.2, 'visual_location', 'ENCODED LOCATION: _visuallocation(color= ,
      screen_x= 320, screen_y= 180, value= )')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: shift and project word')
(0.2137, 'visual', 'AUTOMATIC BUFFERING: _visual(cmd= , color= ,
         screen_pos= _visuallocation(color= , screen_x= 320,
         screen_y= 180, value= ), value= likes)')
(0.25, 'PROCEDURAL', 'RULE FIRED: shift and project word')
(0.25, 'g', 'MODIFIED')
(0.25, 'retrieval', 'CLEARED')
(0.25, 'imaginal', 'CLEARED')
(0.25, 'imaginal', 'CREATED A CHUNK: parse_state(daughter1= Mary,
       daughter2= , lex_head= , mother= , node_cat= ProperN)')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: project: NP ==> ProperN')
(0.3, 'PROCEDURAL', 'RULE FIRED: project: NP ==> ProperN')
(0.3, 'g', 'MODIFIED')
(0.3, 'imaginal', 'CLEARED')
(0.3, 'imaginal', 'CREATED A CHUNK: parse_state(daughter1= ProperN,
      daughter2= , lex_head= Mary, mother= S, node_cat= NP)')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: project and complete: S ==> NP VP')
(0.35, 'manual', 'MOVEMENT FINISHED')
(0.35, 'PROCEDURAL', 'RULE FIRED: project and complete: S ==> NP VP')
(0.35, 'g', 'MODIFIED')
(0.35, 'imaginal', 'CLEARED')
(0.35, 'imaginal', 'CREATED A CHUNK: parse_state(daughter1= NP,
       daughter2= VP, lex_head= , mother= , node_cat= S)')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: press spacebar')
(0.4, 'PROCEDURAL', 'RULE FIRED: press spacebar')
(0.4, 'g', 'MODIFIED')
(0.4, 'manual', 'COMMAND: press_key')
(0.4, 'manual', 'PREPARATION COMPLETE')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'RULE SELECTED: encode word')
(0.45, 'manual', 'INITIATION COMPLETE')
(0.45, 'PROCEDURAL', 'RULE FIRED: encode word')
(0.45, 'g', 'MODIFIED')
(0.45, 'visual', 'CLEARED')
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.45, 'PROCEDURAL', 'RULE SELECTED: retrieve category')
(0.5, 'PROCEDURAL', 'RULE FIRED: retrieve category')
(0.5, 'g', 'MODIFIED')
(0.5, 'retrieval', 'START RETRIEVAL')
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5, 'PROCEDURAL', 'NO RULE FOUND')
(0.55, 'manual', 'KEY PRESSED: SPACE')
(0.55, 'retrieval', 'CLEARED')
****Environment: {1: {'text': 'Bill', 'position': (320, 180)}}
(0.55, 'retrieval', 'RETRIEVED: word(cat= V, form= likes)')
(0.55, 'visual_location', 'ENCODED LOCATION: _visuallocation(color= ,
       screen_x= 320, screen_y= 180, value= )')
(0.55, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.55, 'PROCEDURAL', 'RULE SELECTED: shift and project word')
(0.5659, 'visual', 'AUTOMATIC BUFFERING: _visual(cmd= , color= ,
         screen_pos= _visuallocation(color= , screen_x= 320,
         screen_y= 180, value= ), value= Bill)')
```

```
(0.6, 'PROCEDURAL', 'RULE FIRED: shift and project word')                    98
(0.6, 'g', 'MODIFIED')                                                       99
(0.6, 'retrieval', 'CLEARED')                                               100
(0.6, 'imaginal', 'CLEARED')                                                101
(0.6, 'imaginal', 'CREATED A CHUNK: parse_state(daughter1= likes,           102
     daughter2= , lex_head= , mother= , node_cat= V)')                      103
(0.6, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  104
(0.6, 'PROCEDURAL', 'RULE SELECTED: project and complete: VP ==> V NP')     105
(0.65, 'PROCEDURAL', 'RULE FIRED: project and complete: VP ==> V NP')       106
(0.65, 'g', 'MODIFIED')                                                     107
(0.65, 'imaginal', 'CLEARED')                                               108
(0.65, 'imaginal', 'CREATED A CHUNK: parse_state(daughter1= V,              109
      daughter2= NP, lex_head= , mother= , node_cat= VP)')                  110
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                 111
(0.65, 'PROCEDURAL', 'NO RULE FOUND')                                       112
(0.7, 'manual', 'MOVEMENT FINISHED')                                        113
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  114
(0.7, 'PROCEDURAL', 'RULE SELECTED: press spacebar')                        115
(0.75, 'PROCEDURAL', 'RULE FIRED: press spacebar')                          116
(0.75, 'g', 'MODIFIED')                                                     117
(0.75, 'manual', 'COMMAND: press_key')                                      118
(0.75, 'manual', 'PREPARATION COMPLETE')                                    119
(0.75, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                 120
(0.75, 'PROCEDURAL', 'RULE SELECTED: encode word')                          121
(0.8, 'manual', 'INITIATION COMPLETE')                                      122
(0.8, 'PROCEDURAL', 'RULE FIRED: encode word')                              123
(0.8, 'g', 'MODIFIED')                                                      124
(0.8, 'visual', 'CLEARED')                                                  125
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  126
(0.8, 'PROCEDURAL', 'RULE SELECTED: retrieve category')                     127
(0.85, 'PROCEDURAL', 'RULE FIRED: retrieve category')                       128
(0.85, 'g', 'MODIFIED')                                                     129
(0.85, 'retrieval', 'START RETRIEVAL')                                      130
(0.85, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                 131
(0.85, 'PROCEDURAL', 'NO RULE FOUND')                                       132
(0.9, 'manual', 'KEY PRESSED: SPACE')                                       133
(0.9, 'retrieval', 'CLEARED')                                               134
(0.9, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Bill)')             135
(0.9, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  136
(0.9, 'PROCEDURAL', 'RULE SELECTED: shift and project word')                137
(0.95, 'PROCEDURAL', 'RULE FIRED: shift and project word')                  138
(0.95, 'g', 'MODIFIED')                                                     139
(0.95, 'retrieval', 'CLEARED')                                              140
(0.95, 'imaginal', 'CLEARED')                                               141
(0.95, 'imaginal', 'CREATED A CHUNK: parse_state(daughter1= Bill,           142
      daughter2= , lex_head= , mother= , node_cat= ProperN)')               143
(0.95, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                 144
(0.95, 'PROCEDURAL', 'RULE SELECTED: project and complete: NP ==> ProperN') 145
(1.0, 'PROCEDURAL', 'RULE FIRED: project and complete: NP ==> ProperN')     146
(1.0, 'g', 'MODIFIED')                                                      147
(1.0, 'imaginal', 'CLEARED')                                                148
(1.0, 'imaginal', 'CREATED A CHUNK: parse_state(daughter1= ProperN,         149
     daughter2= , lex_head= Bill, mother= VP, node_cat= NP)')               150
(1.0, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                  151
(1.0, 'PROCEDURAL', 'RULE SELECTED: finished')                              152
(1.05, 'manual', 'MOVEMENT FINISHED')                                       153
(1.05, 'PROCEDURAL', 'RULE FIRED: finished')                                154
(1.05, 'g', 'CLEARED')                                                      155
(1.05, 'imaginal', 'CLEARED')                                               156
(1.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                 157
(1.05, 'PROCEDURAL', 'NO RULE FOUND')                                       158
```

The model output is prodigious since the parser runs for about 1 s, a fairly realistic time for a three-word sentence. We will now briefly examine how the cognitive process of parsing unfolds in time, and then examine the final result of the process more closely.

At the very beginning of the simulation (0 ms), the first word of the sentence (*Mary*) is already displayed on the screen (line 13), so the visual module immediately encodes its location (lines 14–15) and retrieves the visual value (word form) at that

location soon after that (lines 16–18). At the same time, the `"press spacebar"` rule is selected, which will ultimately trigger a key press that will reveal the second word of the sentence. This rule fires 50 ms later (line 19), initiating a manual process in the motor module.

Since the visual module already retrieved the form of the first word, we can select the `"encode word"` rule at the 50 ms point (line 24). At 100 ms, this rule fires (line 26), taking the retrieved visual value and encoding it in the goal chunk as the word to be parsed—line 27: the chunk in the `g` (goal) buffer is modified. The visual buffer is cleared and made available for the next word (line 28).

We can now attempt the retrieval of the encoded word from declarative memory: rule `"retrieve category"` is selected at 100 ms (line 30) and takes 50 ms to fire (line 31). At this point, i.e., at 150 ms, a retrieval process from declarative memory is started (line 33) that takes 50 ms to complete. So at 200 ms, the word *Mary* is retrieved with its syntactic category ProperN (line 39).

Meanwhile, the motor module executes the movement preparation and initiation triggered by the `"press spacebar"` rule fired at 50 ms (lines 22 and 25). This happens in parallel to the parsing steps triggered by the word *Mary*.

With the syntactic category of the first word in hand, we start a cascade of parsing rules triggered by the availability of this 'left corner'.

First, at 200 ms, we select the `"shift and project word"` rule, which fires 50 ms later and creates a chunk in the imaginal buffer storing a unary branching tree with the syntactic category ProperN as the mother node and the word *Mary* as the daughter (lines 51–52).

At this point (250 ms), we select the project-NP rule (line 54), which fires 50 ms later and creates a chunk in the imaginal buffer storing the next part of our tree, namely the NP node built on top of the ProperN node we previously built (lines 58–59).

We are now at 300 ms. We select the project-and-complete-S rule (line 61), which fires at 350 ms and creates a binary branching chunk in the imaginal buffer with S as the mother node, the previously built NP as its first daughter, and a VP as its second daughter that we expect to identify later on in the parsing process (lines 66–67).

In parallel to these parsing actions, the motor and visual modules execute actions that lead to the second word of the sentence being displayed and read. At 200 ms, the motor module is finally ready to press the space key, which displays the word *likes* on the screen (line 38). The visual location of this word is immediately encoded and the visual value is retrieved soon after that.

We are therefore ready at 350 ms to select the `"press spacebar"` rule once again (line 69), which fires 50 ms later and will eventually lead to displaying and reading the final word of the sentence. After that, at 400 ms, we are ready to encode the word *likes* that we displayed and perceived around 185 ms earlier. The `"encode word"` rule is selected at 400 ms (line 75), the rule fires at 450 ms (line 77), and triggers the selection of the `"retrieve category"` rule (line 81).

At the same time, the motor module prepares and initiates the second spacebar press much more quickly: the preparation is instantaneous (line 73) and the initiation takes 50 ms (line 76), so the space key is pressed again at 550 ms. The final word of

the sentence (*Bill*) is displayed on the screen (line 89), triggering the same visual-location encoding and visual-value retrieval steps as before.

Once again, these motor and visual processes happen in parallel, so at 500 ms we are able to fire the `"retrieve category"` rule (line 82) that we selected 50 ms earlier. The process of retrieval from declarative memory takes 50 ms, so at 550 ms we have the V category of our verb *likes* (line 90). We shift and project the verb, which leads to the creation of a chunk in the imaginal buffer projecting a V node on top of the word *likes* (lines 102–103).

We can now select the project-and-complete-VP rule, which fires at 650 ms, creating the VP node we were expecting (based on the previously triggered S rule) on top of the V node we just built, and adding a new expectation for an object NP (lines 109–110).

At this point (700 ms), we are in a state in which `"press spacebar"` can be selected, but since there are no more words to be read, the application of this rule will not have any effect on further parsing.

After that, the `"encode word"` rule can be selected. The rule fires 50 ms later. We then go through the retrieval process for the word *Bill*, after which we project a ProperN node on top of it (lines 142–143).

Finally, we trigger the project-and-complete-NP rule, which completes the object NP we were expecting by recognizing that the ProperN *Bill* is that NP (lines 149–150).

We are done parsing the sentence, so the `"finished"` rule fires at 1050 ms and the parsing simulation ends with the clearing of the `g` (goal) and `imaginal` buffers into declarative memory.

It is instructive to inspect the parse states stored in declarative memory at the end of the simulation, shown in **[py27]** below (sorted by time of (re)activation). We first sort all the contents of declarative memory (lines 1–3), after which we display only the `parse_state` chunks (lines 4–6).

[py27]
```
>>> sortedDM = sorted(([item[0], time]\                                    1
...                     for item in dm.items() for time in item[1]),       2
...                    key=lambda item: item[1])                          3
>>> for chunk in sortedDM:                                                 4
...     if chunk[0].typename == "parse_state":                             5
...         print(chunk[1], "\t", chunk[0])                                6
...                                                                        7
0.3     parse_state(daughter1= Mary, daughter2= , lex_head= ,              8
                    mother= , node_cat= ProperN)                           9
0.35    parse_state(daughter1= ProperN, daughter2= , lex_head= Mary,       10
                    mother= S, node_cat= NP)                               11
0.6     parse_state(daughter1= NP, daughter2= VP, lex_head= ,              12
                    mother= , node_cat= S)                                 13
0.65    parse_state(daughter1= likes, daughter2= , lex_head= ,             14
                    mother= , node_cat= V)                                 15
0.95    parse_state(daughter1= V, daughter2= NP, lex_head= ,               16
                    mother= , node_cat= VP)                                17
1.0     parse_state(daughter1= Bill, daughter2= , lex_head= ,              18
                    mother= , node_cat= ProperN)                           19
1.05    parse_state(daughter1= ProperN, daughter2= , lex_head= Bill,       20
                    mother= VP, node_cat= NP)                              21
```

We see here that the ACT-R architecture places significant constraints on the construction of deep hierarchical structures like syntactic trees: there is no global

view of the constructed tree in declarative memory, only local snapshots recording immediate domination relations, or at the most, two stacked immediate domination relations (when the `mother` slot is specified). As external observers, we can assemble a global view of the syntactic tree based on the `parse_state` chunks stored in declarative memory and their time stamps, but this syntactic tree is never present as such in declarative memory.

The time stamps of the parse states displayed in **[py27]** (lines 8–21) show how the parsing process unfolded over time. We first parsed the subject NP *Mary*, which was completed after about 300 ms (the usual time in word-by-word self-paced reading). Based on this left-corner evidence, we were able to project the S node and add a VP expectation. This expectation was confirmed when the verb *likes* was parsed, after about 300 ms more. But confirming the VP expectation added an object NP expectation; this expectation was confirmed when the final word *Bill* was parsed after yet another 300 ms.

It is similarly instructive to see the words in declarative memory at the end of the simulation (again sorted by time of (re)activation):

[py28]
```
>>> for chunk in sortedDM:
...     if chunk[0].typename == "word":
...         print(chunk[1], "\t", chunk[0])
...
0.0     word(cat= ProperN, form= Mary)
0.0     word(cat= ProperN, form= Bill)
0.0     word(cat= V, form= likes)
0.25    word(cat= ProperN, form= Mary)
0.6     word(cat= V, form= likes)
0.95    word(cat= ProperN, form= Bill)
```

All the words were added to declarative memory at the very beginning of the simulation, and then were reactivated as the parsing process unfolded. The reactivations are roughly spaced at 300 ms intervals, as expected for self-paced reading tasks.

This concludes our introduction to the symbolic part of the ACT-R framework, as well as the introduction of (basic) processing models for linguistic phenomena that can be developed in this cognitive framework.

Chapters 6 and 7 introduce the basic subsymbolic components of ACT-R, which enable us to provide realistic models of performance and on-line/real-time behavioral measures. These models and their numerical parameters can be fit to experimental data in the usual way, using frequentist or Bayesian methods for data analysis.

Chapter 5 provides a brief introduction to Bayesian methods of data analysis, which we will then be able to deploy in the remainder of the book to estimate the subsymbolic parameters of our ACT-R processing models.

4.5 Appendix: The Lexical Decision Model

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 **ch4_lexical_decision.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch4_lexical_decision.py.

File **ch4_leftcorner_parser.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch4_leftcorner_parser.py.