Chapter 2
The ACT-R Cognitive Architecture and Its `pyactr` Implementation

In this chapter, we introduce the ACT-R cognitive architecture and the Python3 implementation `pyactr` we use throughout the book. We end with a basic ACT-R model for subject-verb agreement.

2.1 Cognitive Architectures and ACT-R

Adaptive Control of Thought—Rational (ACT-R[1]) is a cognitive architecture. Cognitive architectures are commonly used in cognitive science to integrate empirical results into a unified cognitive framework, which establishes their consistency and provides a comprehensive formal foundation for future research. They are also used to make/compute fully explicit predictions of abstract and complex theoretical claims.

Using a cognitive architecture can be very useful for the working linguist and psycholinguist, for the very same reasons. This book shows how the ACT-R cognitive architecture can be used to shed light on the cognitive mechanisms underlying a variety of linguistic phenomena, and to quantitatively and qualitatively capture the behavioral patterns observed in a variety of psycholinguistic tasks.

The term 'cognitive architecture' was first introduced by Bell and Newell (1971). A cognitive architecture specifies the general structure of the human mind at a level of abstraction that is sufficient to capture how the mind achieves its goals. Various cognitive architectures exist. They differ in many respects, but their defining

[1]'Control of thought' is used here in a descriptive way, similar to the sense of 'control' in the notion of 'control flow' in imperative programming languages: it determines the order in which programming statements—or cognitive actions—are executed/evaluated, and thus captures essential properties of an algorithm and its specific implementation in a program—or a cognitive system. 'Control of thought' is definitely not used in a prescriptive way roughly equivalent to 'mind control'/indoctrination.

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_2

characteristic is the level of abstractness that the architecture presupposes. As John R. Anderson, the founder of ACT-R, puts it:

> In science, choosing the best level of abstraction for developing a theory is a strategic decision. In the case of connectionist elements or symbolic structures in ACT-R, the question is which level will provide the best bridge between brain and mind [...]. In both cases, the units are a significant abstraction from neurons and real brain processes, but the gap is probably smaller from the connectionist units to the brain. Similarly, in both cases the units are a significant distance from functions of the mind, but probably the gap is smaller in the case of ACT-R units. In both cases, the units are being proposed to provide a useful island to support a bridge from brain to mind. The same level of description might not be best for all applications. Connectionist models have enjoyed their greatest success in describing perceptual processing, while ACT-R models have enjoyed their greatest success in describing higher level processes such as equation solving. [...] I believe ACT-R has found the best level of abstraction for understanding those aspects of the human mind that separate it from the minds of other species. (Anderson 2007, 38–39)

If nothing else, the preceding quote should sound intriguing to linguists or psycholinguists, who often work on higher-level processes involved in language production or comprehension and the competence-level representations that these processes operate on. Thus, linguists and psycholinguists are likely to see ACT-R as providing the right level of abstraction for their scientific enterprise. We hope that this book provides enough detail to show that this is not just an empty promise: ACT-R can be enlightening in formalizing theoretical linguistic claims, and making precise the ways in which these claims connect to processing mechanisms underlying linguistic behavior.

But being intrigued by the idea of cognitive architectures is not enough to justify why cognitive scientists in general, and linguists in particular, should care about cognitive architectures in their daily research. A better justification is that linguistics is part of the larger field of cognitive science, where *process* models of the kind cognitive architectures enable us to formulate are the proper scientific target to aim for. The term 'process models' is taken from Chap. 1 of Lewandowsky and Farrell (2010), who discuss why this type of models—roughly, models of human language performance—provide a higher scientific standard in cognitive science than *characterization* models—roughly, models of human language competence. Both process and characterization models are better than simply *descriptive* models,

> whose sole purpose is to replace the intricacies of a full data set with a simpler representation in terms of the model's parameters. Although those models themselves have no psychological content, they may well have compelling psychological implications. [In contrast, both characterization and process models] seek to illuminate the workings of the mind, rather than data, but do so to a greatly varying extent. Models that characterize processes identify and measure cognitive stages, but they are neutral with respect to the exact mechanics of those stages. [Process] models, by contrast, describe all cognitive processes in great detail and leave nothing within their scope unspecified.
>
> Other distinctions between models are possible and have been proposed [...], and we make no claim that our classification is better than other accounts. Unlike other accounts, however, our three classes of models [descriptive, characterization and process models] map into three distinct tasks that confront cognitive scientists. Do we want to describe data? Do we want to

> identify and characterize broad stages of processing? Do we want to explain how exactly a set of postulated cognitive processes interact to produce the behavior of interest? (Lewandowsky and Farrell 2010, 25)

The advantages and disadvantages of process (performance) models relative to characterization (competence) models can be summarized as follows:

> Like characterization models, [the power of process models] rests on hypothetical cognitive constructs, but [they provide] a detailed explanation of those constructs [...] One might wonder why not every model belongs to this class. After all, if one can specify a process, why not do that rather than just identify and characterize it? The answer is twofold.
>
> First, it is not always possible to specify a presumed process at the level of detail required for [a process] model [...] Second, there are cases in which a coarse characterization may be preferable to a detailed specification. For example, it is vastly more important for a weatherman to know whether it is raining or snowing, rather than being confronted with the exact details of the water molecules' Brownian motion.
>
> Likewise, in psychology [and linguistics!], modeling at this level has allowed theorists to identify common principles across seemingly disparate areas. That said, we believe that in most instances, cognitive scientists would ultimately prefer an explanatory process model over mere characterization. (Lewandowsky and Farrell 2010, 19)

However, there is a more basic reason why linguists should consider process/performance models—and the cognitive architectures that enable us to formulate them—in addition to and at the same time as characterization/competence models. The reason is that a priori, we cannot know whether the best analysis of a linguistic phenomenon is exclusively a matter of competence or performance or both, in much the same way that we do not know in advance whether certain phenomena are best analyzed in syntactic terms or semantic terms or both.[2] Such determinations can only be done *a posteriori*: a variety of accounts need to be devised first, instantiating various points on the competence-performance theoretical spectrum. Once specified in sufficient detail, the accounts can be empirically and methodologically evaluated in systematic ways. Our goal in this book is to provide a framework for building process models, i.e., integrated competence-performance theories, for formal linguistics in general and semantics in particular.

Characterization/competence models have been the focus of theorizing in formal linguistics, and will rightly continue to be one of its main foci for the foreseeable future. However, we believe that the field of linguistics in general—and formal semantics in particular—is now mature enough to start considering process/performance models in a more systematic fashion.

Our main goal for this book is to enable linguists to substantially and productively engage with performance questions related to the linguistic phenomena they investigate. We do this by making it possible and relatively easy for researchers to build integrated competence-performance linguistic models that formalize explicit (quantitative) connections between theoretical constructs and experimental data. Our

[2]We selected syntax and semantics only as a convenient example, since issues at the syntax/semantics interface are by now a staple of (generative) linguistics. Any other linguistic subdisciplines and their interfaces, e.g., phonology or pragmatics, would serve equally well to make the same point.

book should also be of interest to cognitive scientists other than linguists interested to see more ways in which contemporary linguistic theorizing can contribute back to the broader field of cognitive science.

2.2 ACT-R in Cognitive Science and Linguistics

This book and the cognitive models we build and discuss are not intended as a comprehensive introduction and/or reference manual for ACT-R. To become acquainted with ACT-R's theoretical foundations in their full glory, as well as its plethora of applications in cognitive psychology, consider Anderson (1990), Anderson and Lebiere (1998), Anderson et al. (2004), Anderson (2007) among others, and the ACT-R website http://act-r.psy.cmu.edu/.

A quick introduction to the motivation and ideas behind cognitive architectures can be obtained by (i) skimming through Newell (1973b), (ii) watching Allen Newell's 1991 address *Desires and Diversions*, which is an approximately one-hour long movie available on youtube (search for it or go directly to this link https://www.youtube.com/watch?v=_sD42h9d1pk), and (iii) reading the first two chapters of Anderson (2007), Chap. 1 (*Cognitive Architecture*) and Chap. 2 (*The Modular Organization of the Mind*), which are beginner-friendly.

ACT-R is probably the most popular cognitive architecture in linguistics. Its predecessor (ACT) has been used in Anderson (1976) to derive facts about language and grammar. This attempt was criticized in linguistics (Wexler 1978) and this particular research line of using ACT to model language phenomena was abandoned.

Renewed interest in integrating ACT-R and linguistics was sparked by the publication of Lewis and Vasishth (2005), while the contemporary and excellent Budiu and Anderson (2004, 2005) remained largely unknown in the (psycho)linguistic community. Lewis and Vasishth (2005) show that left-corner parsers, originally developed in computational linguistics (Johnson-Laird 1983; Resnik 1992) but with the aim of having cognitively plausible properties, can be implemented in ACT-R. Lewis and Vasishth's models were created by hand-crafting parsing rules and interweaving these rules and memory retrievals. Memory retrievals are needed in parsing to connect various language elements that depend on each other for their interpretation, e.g., verbs and their arguments, or reflexives and their antecedents. The models made precise quantitative predictions for reaction times in eye-tracking while reading and self-paced reading experiments. In particular, the models were successful in simulating effects of interference and distance on memory retrieval (as observable in reaction times).

ACT-R models of real-time language comprehension have since been used to predict the effects of frequency and priming in language production (Reitter et al. 2011), the interaction of parsing and oculomotor control (Engelmann et al. 2013; Dotlačil 2018), the interaction of predictability/surprisal and memory retrieval (Boston et al. 2011), and interference effects in the recall of structural information (Wagers and Phillips 2009; Dillon et al. 2013; Kush et al. 2015; Jäger et al. 2015, 2017; Nicenboim and Vasishth 2018). ACT-R language modeling has also been successful in explaining the acquisition of past-tense verb morphology (Taatgen and Anderson 2002), the semantic processing of metaphors (Budiu and Anderson 2004) and negation (Budiu and Anderson 2005), and impaired processing in individuals with aphasia (Mätzig et al. 2018).

ACT-R's success in modeling linguistic phenomena is to a large extent attributable to the fact that ACT-R is a so-called hybrid cognitive architecture. The "hybrid" qualification refers to the fact that ACT-R combines symbolic and subsymbolic components. The symbolic components enable us to incorporate formal linguistics theories, i.e., theories describing human language competence, in a fairly transparent way. The subsymbolic components enable the resulting ACT-R models to make quantitative predictions for human language *performance* that can be checked against experimental data. Thus, the hybrid architecture is useful in bridging the gap between competence and performance while retaining the essential features of current theorizing in linguistics. This is one of the main reasons it resonated with researchers in (computational) psycholinguistics.

In this book, we do not focus on one particular phenomenon or model, but instead show how ACT-R can be used to model a variety of lexical, syntactic and semantic phenomena. We hope that the variety of applications and the precise (and largely correct) predictions of the models will help researchers assess the usefulness of computational cognitive modeling in general, and ACT-R modeling in particular, for linguistic and psycholinguistic theorizing.

2.3 ACT-R Implementation

One of the main ways in which this book is different from many other texts in linguistics is its hands-on approach to modeling: we will not only discuss and characterize theoretical claims and language models; we will also implement these models in Python3, making extensive use of the ACT-R package `pyactr`, and we will see what the implemented models predict, down to very specific and fine-grained quantitative details.

The ACT-R theory has been implemented in several programming languages, including Lisp (the 'official' implementation), Java (jACT-R, Java ACT-R), Swift (PRIM) and Python2 (ccm). In this book, we will use a novel Python3 implementation: `pyactr`. This implementation is very close to the official implementation in Lisp, so once you learn it you should be able to fairly easily transfer your newly acquired skills to Lisp ACT-R, if you are so inclined.

However, Python seems to be the *de facto lingua franca* of the scientific computing world: it is widely used in the statistics, data science and machine learning communities and it has a very diverse and robust ecosystem of well-maintained and tested libraries, including an easy-to-use, fast, comprehensive, well-tested and up-to-date scientific computing stack. Because of this, implementing any components that do not directly pertain to ACT-R modeling and the specific linguistic phenomenon under investigation is much easier in Python than in Lisp. For example, Python makes it much easier to do data manipulation (wrangling/munging) or statistical analysis, to interact with the operating system, to plot results, to incorporate them in an article or book etc.[3]

Thus, we think `pyactr` is a better tool to learn ACT-R and cognitive modeling: the programming language is more familiar and commonly used, and data collection-manipulation-analysis-and-presentation—as well as general software maintenance—tasks, are much more likely to have good off-the-shelf solutions that require minimal customization. The tool will therefore stand less in the way of the task, so we can focus on actually designing cognitive models, evaluating them and communicating the results.

In addition to the convenience and ease of use that comes with Python, reimplementing ACT-R in `pyactr` also serves to show that ACT-R is a mathematical theory of human cognition that stands on its own, independently of its specific software implementations. While this is well-understood in the cognitive psychology community, it might not be self-evident to working (psycho)linguists or machine-learning researchers.

We will interleave theoretical notes and `pyactr` code throughout the book. We will therefore often display Python code and its associated output in numbered examples and/or numbered blocks so that we can refer to specific parts of the code and/or output and discuss them in more detail. For example, when we want to discuss code, we will display it like so:

(1)
```
2 + 2 == 4        1
3 + 2 == 6        2
```

Note the numbers on the far right—we can use them to refer to specific lines of code, e.g.: the equation on line 1 in (1) is true, while the equation on line 2 is false. We will sometimes also include in-line Python code, displayed like this: `2 + 2 == 4`.

Most of the time however, we will want to discuss both the code and its output, and we will display them in the same way they would appear in the interactive Python interpreter. For example:

[py1]
```
>>> 2 + 2 == 4        1
True                  2
>>> 3 + 2 == 6        3
False                 4
```

Once again, all the lines are numbered (both the Python code and its output) so that we can refer back to specific parts of a code block and output.

Examples—whether formulas, linguistic examples, examples of code etc.—will be numbered as shown in (1) above. Blocks of python code meant to be run interactively, together with their associated output, will be numbered separately, as shown in **[py1]** above.

[3]See https://xkcd.com/353/.

The code for all the models introduced and discussed in the book is available online on GitHub as part of the repository pyactr-book. You can access it by following the link below:

https://github.com/abrsvn/pyactr-book.

2.4 Knowledge in ACT-R

There are two types of knowledge in ACT-R: declarative knowledge and procedural knowledge (see also Newell 1990). Declarative knowledge is our knowledge of facts. For example, if one knows what the capital of the Netherlands is, this is encoded and stored in one's declarative knowledge. Procedural knowledge is knowledge that we display in our behavior (cf. Newell 1973a). This distinction is closely related to the distinction between explicit knowledge ('knowing that') and implicit knowledge ('knowing how') in analytical philosophy (Ryle 1949; Polanyi 1967; see also Davies 2001 and references therein for a more recent discussion).

It is often the case that our procedural knowledge is internalized: we are aware that we have it, but we would be hard pressed to explicitly and precisely describe it. Driving, swimming, riding a bicycle and, arguably, using language, are examples of procedural knowledge. Almost all people who can drive, swim, ride a bicycle, talk etc. do so in an 'automatic' manner. They are able to do it but if asked, they might completely fail to describe exactly how they do it.

ACT-R represents these two types of knowledge in two very different ways. Declarative knowledge is encoded in chunks. Procedural knowledge is encoded in production rules, or productions for short.

2.4.1 Declarative Memory: Chunks

Chunks are lists of attribute-value pairs, familiar to linguists acquainted with feature-based phrase structure grammars (e.g., GPSG, HPSG or LFG—cf. Kaplan et al. 1982; Pollard and Sag 1994; Shieber 2003). However, in ACT-R, we use the term *slot* instead of *attribute*. For example, we might think of one's lexical knowledge of the word *car* as a chunk of type WORD, with the value 'car' for the slot FORM, the value ⟦car⟧ for the slot MEANING, the value 'noun' for the slot CATEGORY and the value 'sg' (singular) for the slot NUMBER. This is represented in graph form in (2) below.

(2)
$$\begin{tikzcd} & \text{sg} & \\ \text{car} & \boxed{\mathit{car}_{\text{WORD}}} \arrow[l, "\text{FORM}"'] \arrow[r, "\text{MEANING}"] \arrow[u, "\text{NUMBER}"'] \arrow[d, "\text{CATEGORY}"] & [\![\text{car}]\!] \\ & \text{noun} & \end{tikzcd}$$

The slot values are the primitive elements ‘car’, ⟦car⟧, ‘noun’ and ‘sg’. Chunks (complex, non-primitive elements) are boxed and subscripted with their type, e.g., $\boxed{\mathit{car}_{\text{WORD}}}$, whereas primitive elements are simple text. A simple arrow ($\longrightarrow$) signifies that the chunk at the start of the arrow has the value at the end of the arrow in the slot with the name that labels the arrow.

The graph representation in (2) will be useful when we introduce activations and, more generally, ACT-R subsymbolic components (see Chap. 6). The same chunk can be represented as an attribute-value matrix (AVM). We will primarily use AVM representations like the one in (3) below from now on.

(3)
$$_{\text{WORD}}\begin{bmatrix} \text{FORM:} & \text{car} \\ \text{MEANING:} & [\![\text{car}]\!] \\ \text{CATEGORY:} & \text{noun} \\ \text{NUMBER:} & \text{sg} \end{bmatrix}$$

2.4.2 Procedural Memory: Productions

A production is an *if*-statement. It describes an action that takes place when the *if* ‘part’ (the antecedent clause) is satisfied. This is why we think of such productions as ⟨precondition, action⟩ pairs. For example, agreement on a verb can be (abstractly) expressed as follows:

(4) *If* the number slot of the subject NP in the sentence currently under construction has the value sg (precondition),
then check that the number slot of the main verb also has the value sg (action).

Of course, for number agreement in English, this is only half of the story. Another production rule would state a similar ⟨precondition, action⟩ pair for pl number. Thus, the basic idea behind production rules is that the *if* part specifies preconditions, and if these preconditions are true, the action specified in the *then* part of the rule is triggered.

Having two rules to specify subject-verb agreement—as we suggested in the previous paragraph—might seem like a cumbersome way of capturing agreement that misses an important generalization: the two rules are really just one agreement rule with two distinct values for the number slot. Could we then just state that the verb should have the same number specification as the subject? ACT-R allows us to state just that if we use variables.

A variable is assigned a value in the precondition part of a production, and it has the same value in the action part. In other words, the scope of any variable assignment is the production rule in which that assignment happens. Given this scope specification for variable assignments, and employing the ACT-R convention that variable names are preceded by '=', we can reformulate our agreement rule as follows:

(5) *If* the number slot of the subject NP in the sentence currently under construction has the value `=x`,
then check that the number slot of the main verb also has the value `=x`.

2.5 The Basics of `pyactr`: Declaring Chunks

We introduce the remainder of the ACT-R architecture by discussing its implementation in `pyactr`. In this section, we describe the inner workings of declarative memory in ACT-R and their implementation in `pyactr`. In the next section (Sect. 2.6), we turn to a discussion of ACT-R modules and buffers and their implementation in `pyactr`. We then turn to explaining how procedural knowledge, a.k.a. procedural memory, and productions are implemented in `pyactr` (Sect. 2.7).

To use `pyactr`, we have to import the relevant package:

[py2]
```
>>> import pyactr as actr
```

We use the `as` keyword so that every time we use methods (functions), classes etc. from the `pyactr` package, we can access them by simply invoking `actr` instead of the longer `pyactr`.

Chunks/feature structures are typed (see Carpenter 1992 for an in-depth discussion of typed feature structures): before introducing a specific chunk, we need to specify a chunk type and all the slots/attributes of that chunk type. This is just good housekeeping: by first declaring a type and the attributes associated with that type, we make clear from the start what kind of objects we take declarative memory to store.

Let's create a chunk type that will encode how our lexical knowledge is stored. We don't strive here for a linguistically realistic theory of lexical representations, we just want to get things off the ground and show the inner workings of ACT-R and `pyactr`:

[py3]
```
>>> actr.chunktype("word", "form, meaning, category, number")
```

The function `chunktype` creates a type `word` with four slots: `form`, `meaning`, `category`, `number`. The type name, provided as a character string `"word"`, is the first argument of the function. The list of slots, with the slots separated by commas, is the second argument. After declaring a type, we can create chunks of that type, e.g., a chunk that will encode our lexical entry for the noun *car*.

[py4]
```
>>> carLexeme = actr.makechunk(nameofchunk="car1",                    1
...                             typename="word",                      2
...                             form="car",                           3
...                             meaning="[[car]]",                    4
...                             category="noun",                      5
...                             number="sg")                          6
>>> print(carLexeme)                                                  7
word(category= noun, form= car, meaning= [[car]], number= sg)         8
```

The chunk is created using the function makechunk, which has two required arguments: nameofchunk, provided on line 1 in **[py4]**, and typename (line 2). Other than these two arguments (with their corresponding values), the chunk consists of whatever slot-value pairs we need it to contain—and they are specified as shown on lines 3–6 in **[py4]**. In general, we do not have to specify the values for all the slots that a chunk of a particular type has; the unspecified slots will be empty.

If you want to inspect a chunk, you can print it, as shown on line 7 in **[py4]**. Note that the order of the slot-value pairs is different from the one we used when we declared the chunk: for example, we defined form first (line 3), but that slot appears as the second slot in the output on line 8. This is because chunks are unordered lists of slot-value pairs, and Python assumes an arbitrary (alphabetic) ordering when printing chunks.

Specifying chunk types is optional. In fact, the information contained in the chunk type is relevant for pyactr, but it has no theoretical significance in ACT-R, it is just 'syntactic sugar'. A chunk type is not identified by the name we choose to give it, but by the slots it has. However, it is recommended to always declare a chunk type before instantiating a chunk of that type: declaring types clarifies what kind of AVMs are needed in our model, and establishes a correspondence between the phenomena and generalizations we are trying to model, on the one hand, and the computational model itself, on the other hand.

For this reason, pyactr will print a warning message if we don't specify a chunk type before declaring a chunk of that type. Among other things, this helps us debug our code. For example, if we accidentally mistype and declare a chunk of type "morphreme" instead of the "morpheme" type we previously declared, we would get a warning message that a new chunk type has been created. We will not display warnings in the code output for the remainder of the book.[4]

It is also recommended that you only use slots already defined in your chunk type declaration (or when you first used a chunk of a particular type). However, you can always add new slots along the way if you need to: pyactr will assume that all the previously declared chunks of the same type had no value for those slots. For example, imagine we realize half-way through our modeling session that it would be useful to specify the syntactic function that a word has. We didn't have that slot in our carLexeme chunk. So let's create a new chunk carLexeme2, which is like carLexeme except it adds this extra piece of information in the slot

[4]See the pyactr and Python3 documentation for more on warnings.

synfunction. We will assume that the synfunction value of carLexeme2 is subject, as shown on line 7 in [py5] below:

[py5]
```
>>> carLexeme2 = actr.makechunk(nameofchunk="car2",              1
...                             typename="word",                 2
...                             form="car",                      3
...                             meaning="[[car]]",               4
...                             category="noun",                 5
...                             number="sg",                     6
...                             synfunction="subject")           7
>>> print(carLexeme2)                                            8
word(category= noun, form= car, meaning= [[car]],                9
     number= sg, synfunction= subject)                           10
```

The command goes through successfully, as shown by the fact that we can print carLexeme2, but a warning message is issued (not displayed above):

```
UserWarning: Chunk type word is extended with new
slots.
```

Another, more intuitive way of specifying a chunk is to use the method chunkstring. When declaring chunks with chunkstring, the chunk type is provided as the value of the isa attribute. The rest of the ⟨slot, value⟩ pairs are listed immediately after that. A ⟨slot, value⟩ pair is specified by separating the slot and value with a blank space.

[py6]
```
>>> carLexeme3 = actr.chunkstring(string="""                     1
...     isa word                                                 2
...     form car                                                 3
...     meaning '[[car]]'                                        4
...     category noun                                            5
...     number sg                                                6
...     synfunction subject                                      7
... """)                                                         8
>>> print(carLexeme3)                                            9
word(category= noun, form= car, meaning= [[car]],                10
     number= sg, synfunction= subject)                           11
```

The method chunkstring provides the same functionality as makechunk. The argument string defines what the chunk consists of. The slot-value pairs are written as a plain string. Note that we use three quotation marks rather than one to provide the chunk string. Triple quotation signals that the string can appear on more than one line. The first slot-value pair, listed on line 2 in [py6], is special. It specifies the type of the chunk, and a special slot is used for this, isa. The resulting chunk is identical to the previous one: we print the chunk and the result is the same as before (see lines 10–11).[5]

Defining chunks as feature structures/AVMs induces a natural notion of identity and a natural notion of information-based ordering over the space of all chunks. A chunk is identical to another chunk if and only if (iff) they have the same slots and the same values for those slots. A chunk is a part of (less informative than) another chunk if the latter includes all the ⟨slot, value⟩ pairs of the former and possibly more.

[5]The value of a slot can also be enclosed in quotes, e.g., 'some-value-here', i.e., it can be provided as a string. The quotes themselves are not treated as part of the value. Using quotes is needed whenever we want to input non-alphanumeric characters, as we have done when we specified the value of the slot meaning.

The pyactr library overloads standard comparison operators for these tasks, as shown below:

[py7]
```
>>> carLexeme2 == carLexeme3
True
>>> carLexeme == carLexeme2
False
>>> carLexeme <= carLexeme2
True
>>> carLexeme < carLexeme2
True
>>> carLexeme2 < carLexeme
False
```

Note that chunk types are irrelevant for deciding identity or part-of relations. This might be counter-intuitive, but it's an essential feature of ACT-R: chunk types are 'syntactic sugar', useful only for the human modeler. This means that if we define a new chunk type that happens to have the same slots as another chunk type, chunks of one type might be identical to, or part of, chunks of the other type:

[py8]
```
>>> actr.chunktype("syncat", "category")
>>> anynoun = actr.makechunk(nameofchunk="anynoun1",
...                          typename="syncat",
...                          category="noun")
>>> anynoun < carLexeme
True
>>> anynoun < carLexeme2
True
```

This way of defining chunk identity is a direct expression of ACT-R's hypothesis that the human declarative memory is content-addressable memory. The only way we have to retrieve a chunk is by means of its slot-value content.[6] Chunks are not indexed in any way and cannot be accessed via their index or their memory address. The only way to access a chunk is by specifying a cue, which is a slot-value pair or a set of such pairs, and retrieving chunks that conform to that pattern, i.e., that are *subsumed* by it.[7]

2.6 Modules and Buffers

Chunks do not live in a vacuum, they are always part of an ACT-R mind (a specific instantiation of the ACT-R mental architecture). The ACT-R building blocks for the human mind are modules and buffers. Each module in ACT-R serves a different mental function. But these modules cannot be accessed or updated directly. Input/output

[6]See McElree (2006) and Jäger et al. (2017) for discussions and summaries of language-related evidence for content-addressable memory retrieval.

[7]A feature structure, a.k.a. chunk, C_1 subsumes another chunk C_2 iff all the information that is contained in C_1 is also contained in C_2. We write this as $C_1 \leq C_2$ or $C_1 \sqsubseteq C_2$. In pyactr, we write `C1 <= C2`. C_1 subsumes C_2 iff all the slots in the domain of C_1 are also in the domain of C_2, and for each of the slots in the domain of C_1, the value of that slot is *identical* to the value of the corresponding slot in C_2. Note that subsumption in ACT-R (also, in pyactr) is not recursively defined, which would require "is *identical* to" in the previous sentence to be replaced by "subsumes".

operations associated with a module are always mediated by a buffer, and each module comes equipped with one such buffer. Think of it as the input/output interface for that mental module.

A buffer has a limited throughput capacity: at any given time, it can carry only one chunk. For example, the declarative memory module can only be accessed via the retrieval buffer. Internally, the declarative memory module supports massively parallel processes: basically all chunks can be simultaneously checked against a cue. But externally, the module can only be accessed serially by placing one cue at a time in its associated retrieval buffer. This is a typical example of how the ACT-R architecture captures actual cognitive behavior by combining serial and parallel components in specific ways (cf. Anderson and Lebiere 1998).

ACT-R conceptualizes the human mind as a system of modules and associated buffers, within and across which chunks are stored and transacted. This flow of information is driven by productions: ACT-R is a production-system based cognitive architecture. Recall that productions are stored in procedural memory, while chunks are stored in declarative memory. The architecture is more complex than that, but in this chapter we will be concerned with only these two major components of the ACT-R architecture for the human mind: procedural memory and declarative memory.

As we already mentioned, procedural memory stores productions. Procedural memory is technically speaking a module, but it is the core module for human cognition, so it does not have to be explicitly declared because it is always assumed to be part of any mind (any instantiation of the mental architecture). The buffer associated with the procedural module is the goal buffer. This reflects the ACT-R view of *human higher cognition as fundamentally goal-driven*. Similarly, declarative memory is a module, and it stores chunks. The buffer associated with the declarative memory module is called the retrieval buffer.

So let us now move beyond just storing arbitrary chunks, and start building a mind. The first thing we need to do is to create a container for the mind, which in `pyactr` terminology is a model:

[py9]
```
>>> agreement = actr.ACTRModel()
```

The mind we intend to build is very simple. It is merely supposed to check for number agreement between the main verb and the subject of a sentence, hence the name of our ACT-R model in **[py9]** above. We can now start fleshing out the anatomy and physiology of this very simple agreeing mind. That is, we will add information about modules, buffers, chunks and productions.

As mentioned above, any ACT-R model has a procedural memory module, but for convenience, it also comes equipped by default with a declarative memory module and the goal and retrieval buffers. When initialized, these buffers/modules are empty. We can check that the declarative memory module is empty, for example:

[py10]
```
>>> agreement.decmem
{}
```

Note that `decmem` is an attribute of our `agreement` ACT-R model, and it stores the declarative memory module. The `retrieval` and `goal` attributes store the retrieval and the goal buffer, respectively, and they are also empty, as shown below.

[py11]
```
>>> agreement.goal
set()
>>> agreement.retrieval
set()
```

It is convenient to have a shorter alias for the declarative memory module, so we introduce a new variable `dm` and assign the `decmem` module as its value:

[py12]
```
>>> dm = agreement.decmem
```

We might want to add a chunk to our declarative memory, e.g., our `carLexeme2` chunk. We add chunks by invoking the `add` method associated with the declarative memory module. The argument of this function call is the chunk that should be added:

[py13]
```
>>> dm.add(carLexeme2)                                              1
>>> print(dm)                                                       2
{word(category= noun, form= car, meaning= [[car]], number= sg,     3
      synfunction= subject): array([0.])}                           4
```

Note that when we inspect `dm`, we can see the chunk we just added. The chunk-encoding time is also recorded. This is the simulation time at which the chunk was added to declarative memory. We have not yet run the model, i.e., we have not yet started the model simulation, so that time is 0 (line 4 in **[py13]**).

2.7 Writing Productions in pyactr

Recall that productions are essentially conditionals (*if*-statements), with the preconditions that need to be satisfied listed in the antecedent of the conditional and the actions that are triggered if the preconditions are satisfied listed in the consequent. Thus, productions have two parts: the preconditions that precede the double arrow (==>) and the actions that follow it.

Let's add some productions to our model to simulate a basic form of verb agreement.[8] Our model of subject-verb agreement will be very elementary, but the point is to learn how to assemble a basic ACT-R model/mind rather than to build a realistic processing model of this linguistic phenomenon. We restrict ourselves to agreement in number for 3rd person present tense verbs. We make no attempt to model syntactic parsing, we will just assume that our declarative memory already stores the subject of the clause, and that the current verb is already present in the goal buffer, where it is being actively assembled.

What should our agreement model do? One production should state that if the goal buffer has a chunk of category `verb` in it and the current task is to agree, then

[8]The full model is linked to in the appendix of this chapter.

the subject should be retrieved. The second production should state that if the number specification on the subject in the retrieval buffer is =x, then the number of the verb in the goal buffer should also be =x (recall that the = sign before a string indicates that the string is the name of a variable). The third rule should say that if the verb is assigned a number, the task is done.

Let's start with the first production: noun retrieval. As shown in [py14], line 1 below, we give the production a descriptive name "retrieve" that will make the simulation output more readable. In general, productions are created by the method productionstring associated with our ACT-R model, and they have two arguments (there is actually a third argument; more on that later): name, the name of the production, and string, which provides the actual content of the production.

[py14]
```
>>> agreement.productionstring(name="retrieve", string="""                1
...     =g>                                                               2
...     isa goal_lexeme                                                   3
...     category verb                                                     4
...     task agree                                                        5
...     ?retrieval>                                                       6
...     buffer empty                                                      7
...     ==>                                                               8
...     =g>                                                               9
...     isa goal_lexeme                                                  10
...     category verb                                                    11
...     task trigger_agreement                                           12
...     +retrieval>                                                      13
...     isa word                                                         14
...     category noun                                                    15
...     synfunction subject                                              16
... """)                                                                 17
{'=g': goal_lexeme(category= verb, task= agree),                         18
 '?retrieval': {'buffer': 'empty'}}                                      19
==>                                                                      20
{'=g': goal_lexeme(category= verb, task= trigger_agreement), '+retrieval':21
 word(category= noun, form= , meaning= , number= , synfunction= subject)} 22
```

The preconditions (the left hand side of the rule) and the actions (the right hand side of the rule) are separated by ==>. This separator can be seen on line 8 in [py14] above. Everything that precedes the separator belongs to the preconditions, and everything that follows it belongs to the actions. The rule has preconditions for two buffers. The first one starts on line 2. =g> indicates two things: the target buffer and the type of precondition this buffer has to satisfy. The precondition checks that the chunk currently stored in the *goal* buffer g is subsumed by the chunk that is specified on the following lines (lines 3–5). The = symbol encodes that we are interested in the subsume relation. That is, the chunk in the goal buffer has to be of category verb (line 4), and the current task for this lexeme should be agree (line 5). The chunk in the goal buffer could have other slot-value pairs, but we are not interested in them for the purposes of this rule.

The second precondition starts on line 6 in [py14] above. ?retrieval> indicates that this precondition will check whether the retrieval buffer is in a certain state. ? in front of the buffer name indicates that we are interested in the state of the buffer, not in the chunk that is in it. The state that we want the retrieval buffer to be in is specified on line 7: the retrieval buffer needs to be empty (no chunk should be stored there).

In general, we can check for a variety of states that buffers could be in. For example:

- `'?g> buffer full'` checks if the goal buffer is full (whether it carries a chunk);
- `'?retrieval> state busy'` checks if the retrieval buffer is working on retrieving a chunk;
- `'?retrieval> state error'` checks if the last retrieval request has failed (no chunk has been found).

If the preconditions on the two buffers are met, the rule triggers two actions. The first action is stated starting on line 9 in [py14]: we modify the `goal_lexeme` chunk by changing the current task from `agree` to `trigger_agreement`. When such a feature-value update takes place, the other features of the updated chunk remain the same.

The `trigger_agreement` task specified in the `goal_lexeme` chunk is to identify a subject noun so that the `goal_lexeme` can agree with that noun in number, which leads us to the second action. This action is stated starting on line 13 in [py14]: `+retrieval>` indicates that we access the `retrieval` buffer (recall that we just verified that this buffer is empty) and we add a new chunk to it (that is what + means). This chunk is our memory cue/query: we want to retrieve from declarative memory a chunk of type `word` that is a `noun` and a `subject`.[9]

Memory cues always consist of chunks, i.e., feature structures, and the retrieval process asks the declarative memory module to provide a (possibly) larger chunk that the cue chunk is a part of (technically, a chunk in declarative memory that is subsumed by our cue chunk). In our specific case, the cue requests the retrieval of a chunk that has at least the following ⟨slot, value⟩ pairs: the chunk should be a `noun` that is a `subject`.

After this production rule is fired, a subject noun is retrieved from declarative memory and placed in the retrieval buffer (assuming the retrieval is successful), and the goal lexeme has `trigger_agreement` as its task. The second production rule, provided in ([py15]) below, can now fire and actually perform the agreement:

[py15]
```
>>> agreement.productionstring(name="agree", string="""
...     =g>
...     isa goal_lexeme
...     category verb
...     task trigger_agreement
...     =retrieval>
...     isa word
...     category noun
...     number =x
...     synfunction subject
...     ==>
...     =g>
...     isa goal_lexeme
...     category verb
```

[9]Strictly speaking, it is not necessary to ensure that the retrieval buffer is empty before placing a retrieval request. The model would have worked just as well if the retrieval buffer had been non-empty. The buffer would have been flushed/emptied first, and then the memory cue would have been placed in it.

```
...     number =x                                                       15
...     task done                                                       16
... """)                                                                17
{'=g': goal_lexeme(category= verb, task= trigger_agreement),            18
 '=retrieval': word(category= noun, form= , meaning= ,                  19
                    number= =x, synfunction= subject)}                  20
==>                                                                     21
{'=g': goal_lexeme(category= verb, number= =x, task= done)}             22
```

The two preconditions of the rule in **[py15]** above ensure that we are in the correct state:

- lines 2–5: the chunk in the goal buffer is subsumed (=) by the chunk on lines 3–5, i.e., it has `verb` as the value of the slot `category`, and `trigger_agreement` as the value of the slot `task`
- lines 6–10: the chunk in the retrieval buffer is subsumed (=) by the chunk on lines 7–10, i.e., it must be of category `noun`, have the syntactic function of `subject` and have a number specification `=x`;
  - since `=x` does not appear anywhere else in the preconditions, this last check is vacuous, as a variable can have any value; however, keep in mind that variables take scope within a rule and, therefore, any other part of this rule that will make use of `=x` will have to match in value the `number` slot in the retrieval buffer.

After checking that we are in the correct state, we trigger the agreeing action. Lines 12–16 in **[py15]** tell us that the chunk that is currently in the goal buffer should be kept there (that's what = on line 12 encodes) and its feature structure should be updated as follows. The type and category should stay the same (`goal_lexeme` and `verb`, respectively), but a new number specification should be added, namely `=x`, which is the same number specification as the one for the subject noun we have retrieved from declarative memory. This completes the agreement operation, so the `task` slot of the goal lexeme is updated and marked as `done` (line 16).

The third and final production rule just mops things up: we are done, so the goal buffer is flushed and our simulation ends. The action on line 6 in **[py16]** below, namely `~g>`, simply discards the chunk in the goal buffer.

[py16]

```
>>> agreement.productionstring(name="done", string="""                  1
...     =g>                                                             2
...     isa goal_lexeme                                                 3
...     task done                                                       4
...     ==>                                                             5
...     ~g>                                                             6
... """)                                                                7
{'=g': goal_lexeme(category= , number= , task= done)}                   8
==>                                                                     9
{'~g': None}                                                            10
```

In the next section, we run the model that we have just created. The notation introduced throughout this section is summarized in Table 2.1 (for preconditions) and Table 2.2 (for actions).

Table 2.1 Notation and terminology used in the preconditions of production rules

	Symbol	
	=	?
Interpretation	Check that subsumption holds	Check the status of the buffer
Possible values	Any chunk that subsumes the chunk in the buffer	Buffer full Buffer empty State busy State free State error

Table 2.2 Notation and terminology used in the actions of production rules

	Symbol		
	=	+	~
Interpretation	Modify the current chunk	Add a new chunk to buffer (triggers memory recall if added to retrieval buffer)	Clear buffer
Possible values	The chunk in the buffer updated with the new slots and values	A chunk with specified slots and values (for retrieval buffer, old chunk from dec. mem. if recall succeeds)	N/A

2.8 Running Our First Model

To run the agreement model, we just have to add an appropriate chunk to the goal buffer. Recall that ACT-R conceptualizes higher cognition as fundamentally goal-driven: if there is no goal, no productions will fire and the mind will not change state.

We add a goal chunk in [py17] below. First, we declare our `goal_lexeme` type (line 1 in [py17]). Then, we add one such chunk to the goal buffer (lines 2–6). Chunks are always added to buffers/modules using the method `add`. We check that the chunk has been added to the goal buffer by printing its contents (line 7). Note that the number specification on line 8 is empty.

[py17]
```
>>> actr.chunktype("goal_lexeme", "task, category, number")      1
>>> agreement.goal.add(actr.chunkstring(string="""                 2
...     isa goal_lexeme                                           3
...     category verb                                             4
...     task agree                                                5
...     """))                                                     6
>>> agreement.goal                                                7
{goal_lexeme(category= verb, number= , task= agree)}              8
```

We can now run the model by invoking the `simulation` method (with no arguments), as shown in [py18], line 1 below. This takes the model specification and initializes various parameters as dictated by the model (e.g., simulation start time). We can then execute one run of the simulation, as shown on line 2 in [py18].

[py18]

```
>>> agreement_sim = agreement.simulation()                              1
>>> agreement_sim.run()                                                 2
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')                                3
(0, 'PROCEDURAL', 'RULE SELECTED: retrieve')                            4
(0.05, 'PROCEDURAL', 'RULE FIRED: retrieve')                            5
(0.05, 'g', 'MODIFIED')                                                 6
(0.05, 'retrieval', 'START RETRIEVAL')                                  7
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')                             8
(0.05, 'PROCEDURAL', 'NO RULE FOUND')                                   9
(0.1, 'retrieval', 'CLEARED')                                          10
(0.1, 'retrieval', 'RETRIEVED: word(category= noun, form= car,         11
      meaning= [[car]], number= sg, synfunction= subject)')            12
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')                             13
(0.1, 'PROCEDURAL', 'RULE SELECTED: agree')                            14
(0.15, 'PROCEDURAL', 'RULE FIRED: agree')                              15
(0.15, 'g', 'MODIFIED')                                                16
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')                            17
(0.15, 'PROCEDURAL', 'RULE SELECTED: done')                            18
(0.2, 'PROCEDURAL', 'RULE FIRED: done')                                19
(0.2, 'g', 'CLEARED')                                                  20
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')                             21
(0.2, 'PROCEDURAL', 'NO RULE FOUND')                                   22
```

The output of the `run()` command is the temporal trace of our model simulation. Each line specifies three elements: (i) the simulation time (in seconds); (ii) the module (name in upper-case letters) or buffer (name in lower-case letters) that is affected; and finally (iii) a description of what is happening to the module or buffer. By default, every cognitive step in the model takes 50 ms, i.e., 0.05 s. This is the ACT-R default time for an elementary cognitive operation.

The first line of our temporal trace (line 3 in [py18]) states that conflict resolution is taking place in the procedural memory module, i.e., the module where all the production rules reside. This happens at simulation time 0. The main function of 'conflict resolution' is to examine the current state of the mind (basically, the state of the buffers in our model) and to determine if any production rule can apply, i.e., to check if the current state of the mind satisfies the preconditions of any production rule.

Note how ACT-R once again combines serial and parallel components to capture actual cognitive behavior. Checking if the current state of the mind satisfies the preconditions of any rule is a massively parallel process: all rules are simultaneously and very quickly (instantaneously) checked. But rule firing is serial: at any given point in the cognitive process, only one rule can fire/apply. This is similar to the interaction between the parallel computations in the declarative memory module (all chunks are simultaneously checked against a pattern/cue) and the serial way in which retrieval cues can be placed in the retrieval buffer (one at a time).

'Conflict resolution' is particularly simple in the present case. Given the state of the goal and retrieval buffers, only one rule can apply: our first production rule, which we named `retrieve` in [py14] above. Line 4 in [py18] shows that the `retrieve` rule is selected at time 0. The rule fires, and this takes the ACT-R default time of 50 ms, as shown on line 5. The state of our mind has changed as a consequence of this rule firing, and the subsequent lines in the output report on that new state: the goal buffer has been modified (line 6; the task is now `trigger_agreement`) and the retrieval buffer has started a memory retrieval procedure (line 7), which will take time to complete.

Now that the retrieve rule has fired, the procedural module enters a ‘conflict resolution’ state again and looks for production rules to apply (line 8). The current state of the mind (i.e., the buffer state) does not satisfy the preconditions of any rule, so none is fired (line 9).

However, a memory retrieval process has been started and is completed 50 ms later, i.e., at the next simulation time of 100 ms. Retrieval time is set to a default value of 50 ms here, but ACT-R specifies in great detail how memory behaves, and makes clear predictions about retrieval accuracy and retrieval latency. This is discussed in detail in Chap. 6, but we want to keep our first model simple, so we use the default retrieval time of 50 ms here.

At the 100 ms point, the memory retrieval process has been completed. The retrieval buffer is cleared (line 10) so that the newly retrieved chunk can be placed there (lines 11–12).

The mind is now in a new state since the buffer contents have changed, so the procedural module reenters a ‘conflict resolution’ state of rule collection and rule selection (line 13). This time, the resolution process identifies one rule that can fire (line 14), namely the second production rule we discussed in **[py15]** above and which we named agree.

The agree rule takes 50 ms to fire (line 15 of **[py18]**), so we are now at 150 ms in simulation time. As a consequence of the agree rule, the chunk in the goal buffer has been modified (line 16): its number specification has been updated so that it is now the same number as the noun chunk in the retrieval buffer.

Agreement has been performed, so the third and final production rule is selected (lines 17–18). The rule takes 50 ms to fire (line 19), so at time 0.2 s, the goal buffer is cleared (line 20), and no further rule can apply (lines 21–22).

When the goal buffer is cleared, the information stored in it does not disappear. The ACT-R architecture specifies that the cleared information is automatically transferred (‘harvested’) to declarative memory. The intuition behind this is that our past accomplished goals, i.e., the results of our past successful cognitive processes, become our present (newly acquired) memory facts. This is also the case in pyactr. We can inspect the final state of the declarative memory module to see that it stores the cleared goal-buffer chunk:

[py19]
```
>>> dm
{word(category= noun, form= car, meaning= [[car]], number= sg,
      synfunction= subject): array([0.]),
 goal_lexeme(category= verb, number= sg, task= done): array([0.2])}
```

Note that this newly added chunk is time-stamped with the simulation time at which the goal buffer was cleared (0.2 s).

And that’s it. At its core, ACT-R provides a fairly simple framework for building process models that is accessible to generative linguists because it is production-rule based and manipulates feature structures of a familiar kind.

To be sure, our first model and the introduction to ACT-R and pyactr in this chapter are overly simplistic in many ways. But the main point is that we can now start building explicit and more realistic computational models for linguistic processes and behaviors.

Our development of integrated competence-performance theories for linguistic phenomena is now at a stage similar to the one in a formal semantics intro course where the semantics for classical first order logic (FOL) has just been introduced. FOL semantics is in many ways an overly simplistic model for natural language semantics, but it provides the basic structure that more realistic theories of natural language interpretation (in the Montagovian tradition) can build on.

2.9 Some More Models

In this section, we present three more (simple) ACT-R models. The models do not add any new concepts to what we have learned so far about ACT-R and `pyactr`. Before we delve into the models, we should point out that none of these models is necessarily cognitively realistic or plausible. We simply present them here to solidify the reader's knowledge of the concepts introduced in this chapter. They also serve as preparation for the more complex linguistic performance models we develop in the remainder of the book.

The first model shows how counting can be simulated in ACT-R. This is a classical, toy example that modelers are often first introduced to when learning about ACT-R.[10] It is a subcomponent of a larger model. The larger model does strive to simulate actual human cognition: it captures how young children learn addition (see Lebiere 1999). However, our simple model does not have this ambitious goal. The second and third models show how regular grammars and counter automata can be implemented in ACT-R.

2.9.1 The Counting Model

The model starts with an initial number and keeps incrementing it by one until it reaches a final number. We have two chunk types: (i) `countOrder`, used to store the list of natural numbers we are counting over in pairs of successive numbers, and (ii) `countFrom`, used to store the current state of the counting process.

[py20]
```
>>> counting = actr.ACTRModel()
>>> actr.chunktype("countOrder", ("first", "second"))
>>> actr.chunktype("countFrom", ("start", "end", "count"))
```

Let's say we want to simulate counting from 2 to 4. We do so by encoding these two parameters in the goal buffer:

[py21]
```
>>> counting.goal.add(actr.chunkstring(string="""
...     isa     countFrom
...     start   2
```

[10]It is the first model in the tutorial units available on the official ACT-R website http://act-r.psy.cmu.edu/.

```
...     end     4
... """))
```

Next, we will store counting knowledge in declarative memory. Since counting goes only up to 4 in our toy example, we will only store the first four numbers and their successors:

[py22]
```
>>> dm = counting.decmem
>>> dm.add(actr.chunkstring(string="""
...     isa     countOrder
...     first   1
...     second  2
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa     countOrder
...     first   2
...     second  3
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa     countOrder
...     first   3
...     second  4
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa     countOrder
...     first   4
...     second  5
... """))
```

Finally, our model will have three rules: `"start"`, `"increment"` and `"stop"`. The `"start"` rule is specified in [py23] below.

[py23]
```
>>> counting.productionstring(name="start", string="""                1
...     =g>                                                          2
...     isa     countFrom                                            3
...     start   =x                                                   4
...     count   None                                                 5
...     ==>                                                          6
...     =g>                                                          7
...     isa     countFrom                                            8
...     count   =x                                                   9
...     +retrieval>                                                  10
...     isa countOrder                                               11
...     first   =x                                                   12
... """)                                                             13
{'=g': countFrom(count= None, end= , start= =x)}                     14
==>                                                                  15
{'=g': countFrom(count= =x, end= , start= ),                         16
 '+retrieval': countOrder(first= =x, second= )}                      17
```

Recall that rules are conditionalized actions and ==> separates preconditions from actions. In this rule, the preconditions simply state that the goal buffer must have a chunk that has no value for the slot `count`. Furthermore, the slot `start` has the value `=x` (since `=x` does not appear anywhere else in preconditions, this is trivially satisfied). As for the actions, the rule specifies changes in two buffers: the goal buffer (lines 7–9) and the retrieval buffer (lines 10–12). The ACT-R model will change the value of the slot `count` to the value assigned to the variable `=x`. This means that the value of the `count` slot in the goal buffer will be matched to the value of the `start` slot. Second, we place a retrieval request for a declarative memory chunk that has the value `=x` in the slot `first`. That is, we want to recall the successor of `=x` from memory.

The `"increment"` rule in [py24] below has preconditions involving the goal and retrieval buffers. It requires the value of `count` in the goal buffer to not match the final, `end` number (lines 4–5). This is achieved by specifying that `count` has the value `=x` and `end` does not have the same value (˜ is negation). Second, the retrieval buffer carries a chunk whose `first` value matches the `count` value in the goal buffer. This condition will be satisfied if the retrieval request placed by the rule `"start"` succeeds. If these preconditions are satisfied, we trigger two actions (lines 11–16). First, the current `count` value will be updated to the value of its successor, which is the value stored in the `second` slot of the chunk in the retrieval buffer (lines 9 and 13). Second, we place a retrieval request for the next increment, i.e., the successor of the updated count (lines 14–16).

[py24]
```
>>> counting.productionstring(name="increment", string="""        1
...     =g>                                                       2
...     isa     countFrom                                         3
...     count   =x                                                4
...     end     ~=x                                               5
...     =retrieval>                                               6
...     isa     countOrder                                        7
...     first   =x                                                8
...     second  =y                                                9
...     ==>                                                       10
...     =g>                                                       11
...     isa     countFrom                                         12
...     count   =y                                                13
...     +retrieval>                                               14
...     isa     countOrder                                        15
...     first   =y                                                16
... """)                                                          17
{'=g': countFrom(count= =x, end= ~=x, start= ),                   18
 '=retrieval': countOrder(first= =x, second= =y)}                 19
==>                                                               20
{'=g': countFrom(count= =y, end= , start= ),                      21
 '+retrieval': countOrder(first= =y, second= )}                   22
```

Finally, if the current count matches the final number (specified in the slot `end`), the `"stop"` rule clears the goal buffer, indicating that the counting goal has been achieved.

[py25]
```
>>> counting.productionstring(name="stop", string="""
...     =g>
...     isa     countFrom
...     count   =x
...     end     =x
...     ==>
...     ~g>
... """)
{'=g': countFrom(count= =x, end= =x, start= )}
==>
{'~g': None}
```

We can now run the counting model:

[py26]
```
>>> counting_sim = counting.simulation()
>>> counting_sim.run()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: start')
(0.05, 'PROCEDURAL', 'RULE FIRED: start')
(0.05, 'g', 'MODIFIED')
(0.05, 'retrieval', 'START RETRIEVAL')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
```

```
(0.1, 'retrieval', 'CLEARED')                                          10
(0.1, 'retrieval', 'RETRIEVED: countOrder(first= 2, second= 3)')       11
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')                             12
(0.1, 'PROCEDURAL', 'RULE SELECTED: increment')                        13
(0.15, 'PROCEDURAL', 'RULE FIRED: increment')                          14
(0.15, 'g', 'MODIFIED')                                                15
(0.15, 'retrieval', 'START RETRIEVAL')                                 16
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')                            17
(0.15, 'PROCEDURAL', 'NO RULE FOUND')                                  18
(0.2, 'retrieval', 'CLEARED')                                          19
(0.2, 'retrieval', 'RETRIEVED: countOrder(first= 3, second= 4)')       20
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')                             21
(0.2, 'PROCEDURAL', 'RULE SELECTED: increment')                        22
(0.25, 'PROCEDURAL', 'RULE FIRED: increment')                          23
(0.25, 'g', 'MODIFIED')                                                24
(0.25, 'retrieval', 'START RETRIEVAL')                                 25
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')                            26
(0.25, 'PROCEDURAL', 'RULE SELECTED: stop')                            27
(0.3, 'retrieval', 'CLEARED')                                          28
(0.3, 'PROCEDURAL', 'RULE FIRED: stop')                                29
(0.3, 'retrieval', 'RETRIEVED: countOrder(first= 4, second= 5)')       30
(0.3, 'g', 'CLEARED')                                                  31
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')                             32
(0.3, 'PROCEDURAL', 'NO RULE FOUND')                                   33
```

The counting process unfolds in the expected way. The model starts at number 2: rule "start" is selected at 0 ms and fires 50 ms later (lines 4–5 in **[py26]**). The retrieval request for the successor of 2 is placed at the 50 ms point (line 7) and is completed successfully at the 100 ms point (line 11).

At this point, the preconditions of the "increment" rule are satisfied, so the rule is selected at 100 ms and fires at 150 ms. The current count is updated to 3 (the g buffer is modified on line 15) and a retrieval request for the successor of 3 is placed.

The retrieval is completed at 200 ms (line 20), at which point the "increment" rule is selected again and fires at 250 ms. Yet again, the current count is updated (line 24), reaching the end goal of 4, and a retrieval request is placed (line 25). The retrieval request is not needed but it is still placed as part of the actions triggered by the "increment" rule.

However, at the same time (that is, we're still at 250 ms) the preconditions of the "stop" rule are satisfied, since the current count matches the end number. The "stop" rule is therefore selected (line 27) and fires 50 ms later (line 29). We are now at 300 ms. The retrieval request for the successor of 4 is successful (line 30), but the counting process is over and the g (goal) buffer is cleared (line 31).

In sum, the model simulates basic counting by successor finding, i.e., incrementing by one. Obviously, this is too trivial compared to how adults actually count, but children arguably learn counting by incrementing by one and only later generalize this procedure. At the same time, children memorize particularly frequent (hence, useful) cases of counting. For more details about ACT-R modeling of arithmetic learning, see Lebiere (1999).

2.9.2 Regular Grammars in ACT-R

Regular grammars can be classified into right-regular and left-regular grammars. Right-regular grammars are grammars whose rules are of the following form:

- X → a Y (where a is a terminal and X, Y are non-terminals)
- X → a (where a is a terminal and X is a non-terminal)
- X → ϵ (where ϵ is the empty string and X is a non-terminal).

That is, the right-hand side of all production rules is constrained so that non-terminal symbols can only occur in the second position/on the right. Right-regular grammars are famously not expressive enough for natural languages (Chomsky 1956), but they make for a good introductory example of modeling basic linguistic patterns in ACT-R.

Let us implement a right-regular grammar in ACT-R, which will generate NP (noun phrase) constituents consisting of indefinitely long strings of nouns. We will represent nouns with the terminal symbol 'N'. We effectively restrict ourselves to one rule. This rule is of the form NP → N NP. That is, every run of the model will generate an NP consisting of a potentially infinite number of Ns.

We need only one chunk type—`goal_chunk` on line 2 of **[py27]** below—encoding the rule NP (`mother`) → N (`daughter1`) NP (`daughter2`). In addition to these three slots, this chunk type has a fourth slot `state`, which will enable us to toggle between printing the value of `daughter1` and applying the 'NP → N NP' rule recursively to the NP in the `daughter2` slot.

[py27]
```
>>> regular_grammar = actr.ACTRModel()
>>> actr.chunktype("goal_chunk", "mother daughter1 daughter2 state")
```

We initialize the goal buffer to an NP `mother` node. The value of `state` will be `rule` which will simply signal that the rewrite rule should be triggered.

[py28]
```
>>> regular_grammar.goal.add(actr.chunkstring(string="""
...     isa         goal_chunk
...     mother      NP
...     state       rule
... """))
```

We need only three rules:

i. one which implements our 'NP → N NP' rule: we rewrite the NP mother node as the daughters N and NP (in that order); see **[py29]** below;
ii. another rule that prints the first daughter, i.e., the terminal node N; see **[py30]**;
iii. a final rule that sets the second daughter, which is the non-terminal NP, as the current node so that the rewrite rule can apply again; see **[py31]**.

The `"NP ==> N NP"`rule in **[py29]** is triggered if our `goal_chunk` has NP as the mother node, no daughters, and is in a state expecting the rule to be applied. If these preconditions are satisfied, we generate the daughter nodes and we enter a `show` state in which the first daughter will be printed.

[py29]
```
>>> regular_grammar.productionstring(name="NP ==> N NP", string="""
...     =g>
...     isa         goal_chunk
...     mother      NP
...     daughter1   None
...     daughter2   None
...     state       rule
...     ==>
...     =g>
...     isa         goal_chunk
...     daughter1   N
...     daughter2   NP
...     state       show
... """)
{'=g': goal_chunk(daughter1= None, daughter2= None, mother= NP, state= rule)}
==>
{'=g': goal_chunk(daughter1= N, daughter2= NP, mother= , state= show)}
```

The "print N" rule in [py30] below is triggered only when the goal_chunk is in a show state. In that case, the value of the daughter1 slot is printed and the state is switched back to a rule application state. Printing is done by specifying that a buffer should execute an action (that is what ! encodes; see line 6 in [py30]), and then specifying the action. In this particular case, the command show on line 7 prints the value of the slot daughter1.

[py30]
```
>>> regular_grammar.productionstring(name="print N", string="""         1
...     =g>                                                               2
...     isa         goal_chunk                                            3
...     state       show                                                  4
...     ==>                                                               5
...     !g>                                                               6
...     show        daughter1                                             7
...     =g>                                                               8
...     isa         goal_chunk                                            9
...     state       rule                                                 10
... """)                                                                 11
{'=g': goal_chunk(daughter1= , daughter2= , mother= , state= show)}     12
==>                                                                      13
{'!g': ([(['show', 'daughter1'], {})], {}),                              14
 '=g': goal_chunk(daughter1= , daughter2= , mother= , state= rule)}     15
```

The final rule "get new mother" in [py31] sets the value of the daughter2 slot as the new mother node (assuming this value is not None), preparing the ground for a new application of the "NP ==> N NP" rule. It also erases the current values of the daughter1 and daughter2 slots, which ensures that the "get new mother" rule cannot apply to its own output. This way, only the "NP ==> N NP" rule can be selected after the "get new mother" rule fires.

[py31]
```
>>> regular_grammar.productionstring(name="get new mother", string="""
...     =g>
...     isa         goal_chunk
...     daughter2   =x
...     daughter2   ~None
...     state       rule
...     ==>
...     =g>
...     isa         goal_chunk
...     mother      =x
...     daughter1   None
...     daughter2   None
... """)
{'=g': goal_chunk(daughter1= , daughter2= =x~None, mother= , state= rule)}
==>
{'=g': goal_chunk(daughter1= None, daughter2= None, mother= =x, state= )}
```

We can now run the simulation for different amounts of time and, depending on that, we will get NPs rewritten as N sequences of varying lengths. To see only the sequence of Ns, we suppress all other output by turning off the temporal trace for the simulation— see `trace=False` in [py32] below.

[py32]
```
>>> regular_grammar_sim = regular_grammar.simulation(trace=False)
>>> regular_grammar_sim.run(0.5)
daughter1 N
daughter1 N
daughter1 N
>>> regular_grammar_sim = regular_grammar.simulation(trace=False)
>>> regular_grammar_sim.run(1)
daughter1 N
daughter1 N
daughter1 N
daughter1 N
daughter1 N
daughter1 N
```

If we want to examine the full trace of the model, we can run it with the `trace` turned on (which is the default setting, so we do not normally need to explicitly specify it). We see that the model runs in repeated cycles: first, the `"NP ==> N NP"` rule fires, then the `"print N"` rule fires, then the `"get new mother"` rule fires, after which this three-rule cycle begins again. The trace in [py33] does not begin with the `"NP ==> N NP"` rule because the model state (specifically, the chunk in the goal buffer) is the one that was the result of the last simulation run in [py32] above.

[py33]
```
>>> regular_grammar_sim = regular_grammar.simulation(trace=True)
>>> regular_grammar_sim.run(0.5)
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: print N')
(0.05, 'PROCEDURAL', 'RULE FIRED: print N')
daughter1 N
(0.05, 'g', 'EXECUTED')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: get new mother')
(0.1, 'PROCEDURAL', 'RULE FIRED: get new mother')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: NP ==> N NP')
(0.15, 'PROCEDURAL', 'RULE FIRED: NP ==> N NP')
(0.15, 'g', 'MODIFIED')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'RULE SELECTED: print N')
(0.2, 'PROCEDURAL', 'RULE FIRED: print N')
daughter1 N
(0.2, 'g', 'EXECUTED')
(0.2, 'g', 'MODIFIED')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: get new mother')
(0.25, 'PROCEDURAL', 'RULE FIRED: get new mother')
(0.25, 'g', 'MODIFIED')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: NP ==> N NP')
(0.3, 'PROCEDURAL', 'RULE FIRED: NP ==> N NP')
(0.3, 'g', 'MODIFIED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: print N')
(0.35, 'PROCEDURAL', 'RULE FIRED: print N')
daughter1 N
(0.35, 'g', 'EXECUTED')
(0.35, 'g', 'MODIFIED')
```

```
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: get new mother')
(0.4, 'PROCEDURAL', 'RULE FIRED: get new mother')
(0.4, 'g', 'MODIFIED')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'RULE SELECTED: NP ==> N NP')
(0.45, 'PROCEDURAL', 'RULE FIRED: NP ==> N NP')
(0.45, 'g', 'MODIFIED')
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.45, 'PROCEDURAL', 'RULE SELECTED: print N')
```

2.9.3 Counter Automata in ACT-R

The last example we discuss is the implementation of a counter automaton in ACT-R/pyactr. A counter automaton is a type of push-down automaton: it is a push-down automaton that allows only two symbols to appear on the stack. One well-known example of a language recognized by a counter automaton, but not generated by a regular grammar, is the language $\{a^n b^n : n \geq 1\} = \{ab, aabb, aaabbb, aaaabbbb, \dots\}$, for two arbitrary terminals a and b. One-counter automata recognize a subset of the context-free languages (see Hopcroft et al. 2001, 351 et seqq for more details).

Let's implement a grammar corresponding to this automaton in ACT-R, and use it to generate (a finite subset of) this language. Our implementation builds on the counting model we discussed above, since we need to count the number of a and b occurrences. Let's initialize the model and incorporate the counting model specification first. Everything in **[py34]** below is the same as in the counting model, with the exception of the fact that we add another slot terminal to our countFrom chunk type.

[py34]
```
>>> counter = actr.ACTRModel()

>>> actr.chunktype("countOrder", "first, second")
>>> actr.chunktype("countFrom", ("start", "end", "count", "terminal"))

>>> dm = counter.decmem
>>> dm.add(actr.chunkstring(string="""
...     isa         countOrder
...     first       1
...     second      2
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa         countOrder
...     first       2
...     second      3
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa         countOrder
...     first       3
...     second      4
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa         countOrder
...     first       4
...     second      5
... """))
```

We will let the model start with the goal of generating two adjacent sequences of two elements each, the first sequence consisting of ‘*a*’s:

[py35]
```
>>> counter.goal.add(actr.chunkstring(string="""
...     isa         countFrom
...     start       1
...     end         3
...     terminal    a
... """))
```

We can now specify our production rules. The `"start"` rule in [py36] below is the same as in the counting model. The other two rules—`"increment"` in [py37] and `"restart counting"` in [py38] below—are almost identical to the rules of the counting model except: (i) whenever we increment, we print the terminal (*a* or *b*), [py37]; (ii) when we are done counting the first sequence of terminals (the sequence of ‘*a*’s), we do not stop but switch to counting and printing ‘*b*’s, [py38].

[py36]
```
>>> counter.productionstring(name="start", string="""
...     =g>
...     isa         countFrom
...     start       =x
...     count       None
...     ==>
...     =g>
...     isa         countFrom
...     count       =x
...     +retrieval>
...     isa         countOrder
...     first       =x
... """)
{'=g': countFrom(count= None, end= , start= =x, terminal= )}
==>
{'=g': countFrom(count= =x, end= , start= , terminal= ),
 '+retrieval': countOrder(first= =x, second= )}
```

[py37]
```
>>> counter.productionstring(name="increment", string="""
...     =g>
...     isa         countFrom
...     count       =x
...     end         ~=x
...     =retrieval>
...     isa         countOrder
...     first       =x
...     second      =y
...     ==>
...     !g>
...     show        terminal
...     =g>
...     isa         countFrom
...     count       =y
...     +retrieval>
...     isa         countOrder
...     first       =y
... """)
{'=g': countFrom(count= =x, end= ~=x, start= , terminal= ),
 '=retrieval': countOrder(first= =x, second= =y)}
==>
{'!g': ([(['show', 'terminal'], {})], {}),
 '=g': countFrom(count= =y, end= , start= , terminal= ),
 '+retrieval': countOrder(first= =y, second= )}
```

[py38]
```
>>> counter.productionstring(name="restart counting", string="""
...     =g>
...     isa         countFrom
...     count       =x
...     end         =x
...     terminal    a
...     ==>
...     +g>
...     isa         countFrom
...     start       1
...     end         =x
...     terminal    b
... """)
{'=g': countFrom(count= =x, end= =x, start= , terminal= a)}
==>
{'+g': countFrom(count= , end= =x, start= 1, terminal= b)}
```

We can now run the model. Notice that it prints 2 *a*s (since we start at 1 and count up to 3), followed by the same number of *b*s.

[py39]
```
>>> counter_sim = counter.simulation(trace=False)
>>> counter_sim.run()
terminal a
terminal a
terminal b
terminal b
```

The model can in principle generate indefinitely long $a^n b^n$ expressions (if we add enough number knowledge to declarative memory), but in practice, it is limited by time and memory constraints. One might see this as a limitation of the implemented model, but this actually makes the model cognitively more realistic since humans are also limited by time and memory constraints.

2.10 Appendix: The Four Models for Agreement, Counting, Regular Grammars and Counter Automata

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 you would any other Python3 script. Links to the specific files that contain the models discussed in this chapter are provided below:

File **ch2_agreement.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch2_agreement.py.

File **ch2_count.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch2_count.py.

File **ch2_regular_grammar.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch2_regular_grammar.py.

File **ch2_counter_automaton.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch2_counter_automaton.py.