Computing dynamic meanings: Building integrated competence-performance theories for formal semantics

Adrian Brasoveanu, Jakub Dotlačil[1]

May 8, 2017

[1]ACKNOWLEDGMENTS to be inserted here . . . This document has been created with LaTeX and PythonTex (Poore, 2013). The usual disclaimers apply.

Contents

Chapter 1

Introduction

– overview of the book, intended audience, getting started (installation instructions etc.)

1.1 Using `pyactr` – people familiar with Python

If you are familiar with Python, you can install `pyactr` (the Python package that enables ACT-R) and proceed to Chapter 2. `pyactr` is a Python 3 package and can be installed using `pip` (for Python 3): type the command below in your terminal.

```
$ pip3 install pyactr
```

Alternatively, you can download the package here: https://github.com/jakdot/pyactr and follow the instructions there to install the package.

If you are not familiar with Python, you should consider the steps below.

1.2 Using `pyactr` – beginners

`pyactr` is a package for Python 3. To get started, you should consider a web-based service for Python3 like `PythonAnywhere`. In this type of services, computation is hosted on separate servers and you don't have to install anything on your computer (of course, you'll need Internet access). If you find you like working with Python and `pyactr`, you can install them on your computer at a later point together with a good text editor for code – or install an integrated desktop environment (IDE) for Python – a common choice is `anaconda`, which comes with a variety of ways of working interactively with Python (IDE with Spyder as the editor, `ipython` notebooks etc.). But none of this is required to run `pyactr` and the code in this book.

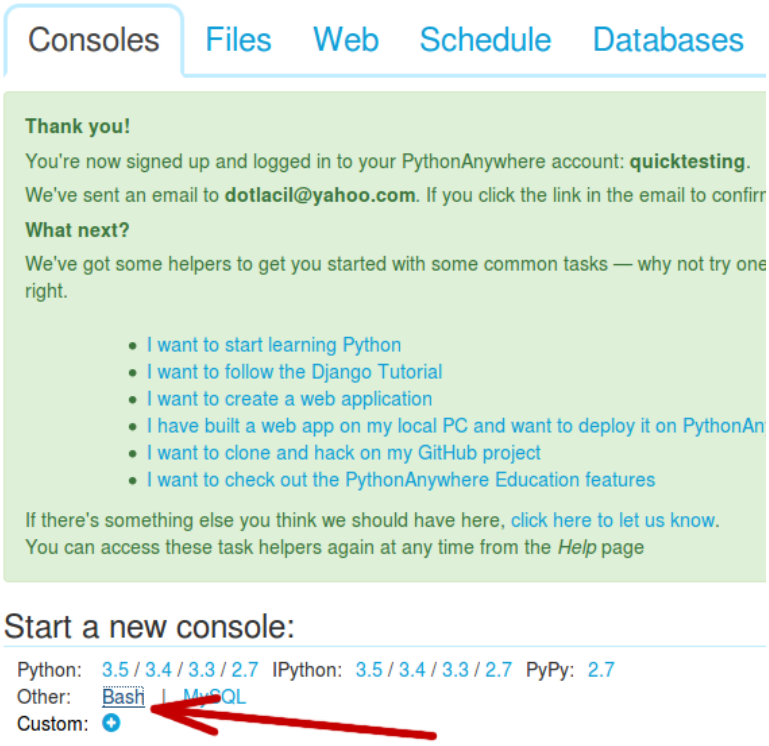
a. Go to www.pythonanywhere.com and sign up there.

b. You'll receive a confirmation e-mail. Confirm your account / e-mail address.

c. Log into your account on www.pythonanywhere.com.

d. You should see a window like the one below. Click on Bash (below "Start a new Console").

Figure 1: Opening Bash in PythonAnywhere.



e. In Bash, type:

```
$ pip3 install --user pyactr
```

This will install `pyactr` in your Python account (not on your computer). The output of this command should be similar to this:

```
Collecting pyactr
Downloading pyactr-0.1.9-py3-none-any.whl (50kB)
100%
Requirement already satisfied (use --upgrade to upgrade):
    pyparsing in /usr/local/lib/python3.5/dist-packages (from pyactr)
Requirement already satisfied (use --upgrade to upgrade):
    simpy in /usr/local/lib/python3.5/dist-packages (from pyactr)
Requirement already satisfied (use --upgrade to upgrade):
    numpy in /usr/local/lib/python3.5/dist-packages (from pyactr)
Installing collected packages: pyactr
Successfully installed pyactr
```

f. Go back to Consoles. Start Python by clicking on any version higher than 3.2.

g. A console should open. Type:

```
import pyactr
```

If no errors appear, you are set and can proceed to Chapter 2. You might get a warning about the lack of `tkinter` support and that the simulation GUI is set to false:

```
~/.local/lib/python3.5/site-packages/pyactr/simulation.py:10:
    UserWarning: Simulation cannot start a new window because tkinter
    is not installed. You will have no GUI for environment. If you want
    to change that, install tkinter. warnings.warn("Simulation cannot
    start a new window because tkinter is not installed. You will have
    no GUI for environment. If you want to change that, install tkinter.")
~/.local/lib/python3.5/site-packages/pyactr/simulation.py:11:
    UserWarning: Simulation GUI is set to False.
warnings.warn("Simulation GUI is set to False.")
```

Ignore it.

Throughout the book, we will introduce and discuss various ACT-R models coded in Python. You can either type them in line by line or even better, load them as files in your session on PythonAnywhere. Scripts are uploaded under the tab Files. You should be aware that the free account of PythonAnywhere allows you to run only two consoles, and there is a limit on the amount of CPU you might use per day. The limit should suffice for the tutorials but if you find this too constraining, you should consider installing Python (Python3) and `pyactr` on your computer and running scripts directly there.

1.3 The structure of the book

!!! Add here brief summary for the structure of the book.

Chapter 2 introduces the ACT-R cognitive framework, the Python 3 implementation we will be using (`pyactr`) and ends with a basic ACT-R model for subject-verb agreement. Chapter 3 introduces the basics of syntactic parsing in ACT-R. We build a top-down parser and learn more about how we can closely examine intermediate results of `pyactr` simulations to obtain detailed snapshots of the cognitive states our processing models predict. Chapter 4 introduces a psycholinguistically realistic model of syntactic parsing (left-corner parsing), as well as vision and motor modules that enable our models of the human mind to interact with the environment in the same way human participants do in actual psycholinguistic experiments.

Chapter 5 introduces the 'subsymbolic' components needed to have a realistic model of human declarative memory and provides end-to-end models for a variety of psycholinguistic tasks – lexical decision, self-paced reading and eye-tracking – focusing exclusively on the syntactic aspects of linguistic behavior in these tasks. The models are end-to-end in the sense that they include explicit linguistic analyses together with a realistic model of declarative memory and simple but reasonably realistic vision and motor modules. The chapter also discusses how closely these models fit actual data.

The next chapters finally talk about semantics ...

Chapter 2

The ACT-R cognitive architecture and its `pyactr` implementation

2.1 Introduction

Adaptive Control of Thought – Rational (ACT-R[1]) is a cognitive architecture: it is a theory of the structure of the human mind/brain that explains and predicts human cognition. The ACT-R theory has been implemented in several programming languages, including Java (jACT-R, Java ACT-R), Swift (PRIM), Python2 (ccm). The canonical implementation has been created and is maintained in Lisp.

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 transfer your skills very quickly to code models in Lisp ACT-R if you wish to do that. At the same time, Python is currently much more widespread than Lisp and has a much larger and more diverse ecosystem of libraries, so coding parts that do not directly pertain to the ACT-R model are much better supported and much easier than in Lisp – for example, data manipulation/munging, statistical analysis, interactions with the operating system, displaying simulation results, incorporating them into tex / pdf documents etc.

Thus, we think `pyactr` is a better tool to learn ACT-R and cognitive modeling for linguists: the programming language is likely more familiar and commonly used, and data collection, piping, analysis and presentation as well as general software maintenance/enhancement 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 doing cognitive modeling for linguistic phenomena, evaluating our models and communicating our results, rather than having to spend a significant amount of time on issues having to do with the computational tools we need to achieve our modeling goals.

In addition to the convenience and ease of use that comes with Python, reimplementing ACT-R in `pyactr` or, as we will sometimes do, implementing parts of ACT-R within Bayesian

[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 cognitive system). 'Control of thought' is definitely not used in a prescriptive way roughly equivalent to 'mind control' / indoctrination.

pymc3 models (pymc3 is a Python3 library for Bayesian modeling; see chapters 5-6) 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 linguists that ACT-R's empirical predictions are driven by general, formalized theoretical principles rather than tweaking arcane features of a software package.

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 as well as the ACT-R website http://act-r.psy.cmu.edu/.

The main goal of this book is to take a hands-on approach to introducing ACT-R by constructing models that aim to solve linguistic problems. 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 the 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 equality in (1), line 1 is true, while the equality in (1), line 2 is false. We will sometime 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 lines are numbered (both the Python code and its output) so that we can refer back to it.

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.

2.2 Why do we care about ACT-R, and cognitive architectures and modeling in general

Linguistics is part of the larger field of cognitive science. So the answer to the question "Why do we care about ACT-R and cognitive architectures / modeling in general?" is one that applies to cognitive sciences in general. Here is one recent formulation of what we

take to be the right answer, taken from chapter 1 of Lewandowsky and Farrell (2010). That chapter mounts an argument for *process* models as the proper scientific target to aim for in the cognitive sciences – roughly, models of human language performance – rather 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 generative linguists should consider process / performance models 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 generative linguistics in general and formal semantics in particular.

Characterization / competence models have been the focus of linguistic theorizing over the 60 years in which the field of generative linguistics matured, and will rightly continue to be one of its main foci for the foreseeable future. However, we believe that the field of generative 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 semanticists to productively engage with performance questions related to the linguistic phenomena they investigate. We do this by making it possible and relatively easy for semanticists, and generative linguists in general, to build integrated competence/performance linguistic models of semantic phenomena that formalize explicit (quantitative) connections between theoretical constructs and experimental data. Our book should also be of interest to cognitive scientists other than linguists interested to see more ways in which contemporary generative linguistic theorizing can contribute back to the broader field of cognitive science.

2.3 Knowledge in ACT-R

There are two types of knowledge in ACT-R: declarative knowledge and procedural knowledge (see also Newell 1990).

The declarative knowledge represents our knowledge of facts. For example, if one knows what the capital of the Netherlands is, this would be represented in one's declarative knowledge.

Procedural knowledge is knowledge that we display in our behavior (cf. Newell 1973). 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 are examples of procedural knowledge. Almost all people who can drive / swim / ride a bicycle do so in an automatic way. They are able to do it but they might completely fail to describe how exactly they do it when asked. This distinction is closely related to the distinction between explicit ('know that') and implicit ('know how') knowledge in analytical philosophy (Ryle 1949; Polanyi 1967; see also Davies 2001 and references therein for more recent discussions).

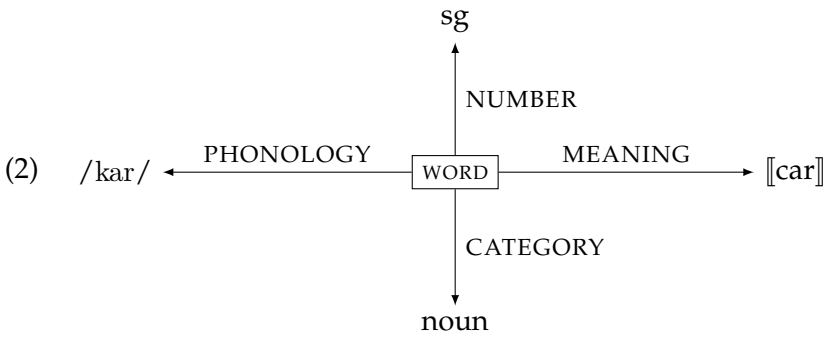
ACT-R represents these two types of knowledge in two very different ways. The declarative knowledge is instantiated in chunks. The procedural knowledge is instantiated in production rules, or productions for short.

[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.

2.3.1 Representing declarative knowledge: chunks

Chunks are lists of attribute-value pairs, familiar to linguists acquainted with feature-based phrase structure grammars (e.g., GPSG, HPSG or LFG – INSERT REFERENCES HERE). However, in ACT-R, we use the term *slot* instead of *attribute*. For example, we might think of one's knowledge of the word *carLexeme* as a chunk of type WORD with the value /kar/ for the slot *phonology*, 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 (2) below:

(2)



The slot values are the primitive elements /kar/, ⟦carLexeme⟧, *noun* and *sg*, respectively. Chunks (complex, non-primitive elements) are boxed, whereas primitive elements are simple text. A simple arrow (⟶) 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 Chapter 6). The same chunk can be represented as an attribute-value matrix (AVM), which is much more familiar to linguists. We will overwhelmingly use AVM representations like the one in (3) from now on.

(3)

$$\text{WORD}\begin{bmatrix}\text{PHONOLOGY:} & /\text{kar}/\\ \text{MEANING:} & [\![\text{car}]\!]\\ \text{CATEGORY:} & \text{noun}\\ \text{NUMBER:} & \text{sg}\end{bmatrix}$$

2.3.2 Representing procedural knowledge: productions

A production is an *if*-statement. It describes an action that takes place if the *if* 'part' (the antecedent clause) is satisfied; this is why we think of such productions / conditionals as ⟨precondition, action⟩ pairs. For example, agreement on a verb can be (abstractly) expressed as follows: *if* the subject number in the sentence currently under construction is *sg* (precondition), *then* check that the verb number in the sentence is *sg* (action). Of course, 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 specifying agreement that misses an

important generalization: the two rules are really just one agreement rule with two distinct values for the number morphology. 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 that same value in the action part, i.e., 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: *if* the subject number in the sentence currently under construction is =x, *then* check that the number specification on the (main) verb of the sentence is also =x.

2.4 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 details of declarative knowledge in ACT-R and the implementation of those details in `pyactr`. We will then turn to a discussion of modules and buffers, which are the building blocks of the mind in ACT-R (section §2.5). After this, we can finally turn to the second type of knowledge in ACT-R: procedural knowledge / productions (section §2.6).

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

```
[py2] >>> import pyactr as actr                                    1
```

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 are clear from the start about 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", "phonology, meaning, category, number")   1
```

The function, a.k.a. method, `chunktype` creates a type `word` with four slots: `phonology`, `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 knowledge about the noun *car*.

```
[py4] >>> carLexeme = actr.makechunk(nameofchunk="car",\                1
      ...                         typename="word",\                    2
```

```
... phonology="/kar/",\ 3
... meaning="[[car]]",\ 4
... category="noun",\ 5
... number="sg") 6
>>> print(carLexeme) 7
word(category= noun, meaning= [[car]], number= sg, phonology= /kar/) 8
```

The chunk is created using the method `makechunk`, which has two required arguments: `nameofchunk`, provided on line 1 in [py4], and `typename` (line 2). Other than these two slots (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 all the slots that a chunk of a particular type should have; 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 `phonology` first (line 3), but that slot appears last 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'. 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 and the computational model itself. For this reason, if we don't specify a chunk type before declaring a chunk of that type, `pyactr` will print a warning message. Among other things, this helps us debug our code – e.g., 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.[3]

It is also recommended that you only use attributes already defined in your chunk type declaration – or when you first used a chunk of a particular type. However, you can always add new attributes 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 attributes. For example, imagine we realize half-way through our modeling session that it would be useful to specify what syntactic function 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 `synfunction`. We will assume that the `synfunction` value of `carLexeme2` is `subject`, as shown on line 7 in [py5]:

```
[py5] >>> carLexeme2 = actr.makechunk(nameofchunk="car2",\ 1
... typename="word",\ 2
... phonology="/kar/",\ 3
... meaning="[[car]]",\ 4
... category="noun",\ 5
... number="sg",\ 6
... synfunction="subject") 7
```

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

```
>>> print(carLexeme2)                                                    8
word(category= noun, meaning= [[car]], number= sg, phonology= /kar/,    9
   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 attributes`.

Another, more intuitive way of specifying a chunk uses 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, separated by commas. 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
...     phonology '/kar/'                                                3
...     meaning '[[car]]'                                                4
...     category 'noun'                                                  5
...     number 'sg'                                                      6
...     synfunction 'subject'""")                                        7
>>> print(carLexeme3)                                                    8
word(category= noun, meaning= [[car]], number= sg, phonology= /kar/,    9
   synfunction= subject)                                                10
```

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. These signal to Python that the string can appear on more than one line. The first slot-value pair ([py6], line 2) 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 listed on lines 9-10 of [py6] is the same as before.

Defining chunks as feature structures / AVMs induces a natural notion of identity and 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 attributes and the same values for those attributes. 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. The `pyactr` library overloads standard comparison operators for these tasks, as shown below:

[py7]
```
>>> carLexeme2 == carLexeme3                                            1
True                                                                    2
>>> carLexeme == carLexeme2                                             3
False                                                                   4
>>> carLexeme <= carLexeme2                                             5
True                                                                    6
>>> carLexeme < carLexeme2                                              7
True                                                                    8
>>> carLexeme2 < carLexeme                                              9
False                                                                   10
```

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")
>>> noun = actr.makechunk(nameofchunk="noun",
...                       typename="syncat",
...                       category="noun")
>>> noun < carLexeme
True
>>> noun < 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. Chunks are not indexed in any way and cannot be accessed via their index / memory address: the only way to access a chunk is by specifying a cue / pattern, 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.[4]

2.5 Modules and buffers

Chunks do not live in a vacuum, they are always part of an 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 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. And while the memory module supports massively parallel processes – basically all chunks can be simultaneously checked against a pattern / cue, 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.

For ACT-R, the human mind is 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. Basically, productions

[4] 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. Basically, if C_1 is atomic / primitive, e.g., the number value *sg*, then C_1 subsumes C_2 iff C_2 is also atomic / primitive with the same atom. If C_1 is complex, i.e., it is a set of slot-value pairs, 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 subsumes the value of the corresponding slot in C_2.

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 / control module for human cognition so it does not have to be explicitly declared because is always assumed to be part of any 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 cognitive. Similarly, declarative memory is a module, and it stores chunks. The buffer associated with the declarative memory module is called the retrieval buffer.

Let's build 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()                                    1
```

The mind we intend to build is simply supposed to check for number agreement between the main verb and 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 for declarative memory, for example:

```
[py10] >>> agreement.decmem                                               1
       {}                                                                2
```

`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.

```
[py11] >>> agreement.goal                                                1
       set()                                                             2
       >>> agreement.retrieval                                          3
       set()                                                             4
```

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

```
[py12] >>> dm = agreement.decmem                                         1
```

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)
>>> print(dm)
{word(category= noun, meaning= [[car]], number= sg, phonology= /kar/,
   synfunction= subject): array([ 0.])}
```

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 / started the model simulation, so that time is 0 (line 4 of [py13]).

2.6 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 action that is 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 the arrow.

Let's add some productions to our model to simulate a basic form of verb agreement.[5] Our model of subject-verb agreement will be very elementary, but the point is learning how to assemble the basic architecture of the model / mind rather than building 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 stores the subject of the clause and the current verb is already present in the goal buffer, where it is being actively assembled/specified.

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 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="""
...     =g>
...     isa goal_lexeme
...     category 'verb'
...     task agree
...     ?retrieval>
...     buffer empty
```

[5] The full model is provided in the appendix to this chapter.

```
...     ==>                                                                    8
...     =g>                                                                    9
...     isa goal_lexeme                                                        10
...     task trigger_agreement                                                 11
...     category 'verb'                                                        12
...     +retrieval>                                                            13
...     isa word                                                               14
...     category 'noun'                                                        15
...     synfunction 'subject'                                                  16
...     """)                                                                   17
{'=g': goal_lexeme(category= verb, task= agree), '?retrieval': {'buffer':      18
   'empty'}}                                                                   19
==>                                                                            20
{'=g': goal_lexeme(category= verb, task= trigger_agreement), '+retrieval':     21
   word(category= noun, meaning= , number= , phonology= , synfunction=         22
   subject)}                                                                   23
```

The preconditions (left hand side of the rule / antecedent of the conditional) and the actions (right hand side of the rule / consequent of the conditional) are separated by ==> – see line 8 of [py14] above. The rule has two preconditions. The first one starts on line 2: =g> indicates that this precondition will check that the chunk currently stored in the goal buffer (that's what g encodes) is (that's what = encodes) of a particular kind: the chunk has to be a goal_lexeme (line 3) of category 'verb' (line 4), and the current task for this lexeme should be agree (line 5). The second precondition starts on line 6: ?retrieval> indicates that this precondition will check whether the retrieval buffer is in a certain state (that's what ? encodes). The state 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 whether the goal buffer is full (whether it carries a chunk); '?retrieval> state busy' checks if the retrieval buffer is working on retrieving a chunk; and '?retrieval> state error' checks if the last retrieval has failed (no chunk has been found).

If these two preconditions are met, the rule triggers two actions. The first action is stated starting on line 9 of [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 – here, the goal_lexeme chunk – remain the same.

The triggered agreement on the goal_lexeme chunk needs to identify a subject noun so that it can agree with that noun in number, which leads us to the second action. This action is stated starting on line 13: +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's 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'.[6]

Memory cues always consist of chunks, i.e., feature structures, and the retrieval process

[6] 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 just be flushed/emptied first, and the memory cue would then be placed in it.

asks the declarative memory module to provide a 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 of type `word`, its category should be `'noun'` and its syntactic category should be `'subject'`.

We are now in a state in which a subject noun will be retrieved from declarative memory and placed in the retrieval buffer, and the goal lexeme is in an 'active' state of triggered agreement. The second production rule performs the agreement:

```
[py15] >>> agreement.productionstring(name="agree", string="""                    1
       ...     =g>                                                               2
       ...     isa goal_lexeme                                                   3
       ...     task trigger_agreement                                            4
       ...     category 'verb'                                                   5
       ...     =retrieval>                                                       6
       ...     isa word                                                          7
       ...     category 'noun'                                                   8
       ...     synfunction 'subject'                                             9
       ...     number =x                                                         10
       ...     ==>                                                               11
       ...     =g>                                                               12
       ...     isa goal_lexeme                                                   13
       ...     category 'verb'                                                   14
       ...     number =x                                                         15
       ...     task done                                                         16
       ...     """)                                                              17
       {'=g': goal_lexeme(category= verb, task= trigger_agreement), '=retrieval': 18
          word(category= noun, meaning= , number= =x, phonology= , synfunction=  19
          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 (=) a `goal_lexeme` of category `'verb'` that is in an active state of agreeing (the current `task` is `trigger_agreement`)
- lines 6-10: the chunk in the retrieval buffer is (=) a `word` of category `'noun'` that is a `'subject'` and that has a number specification `=x` (more precisely: take that number specification and assign it to the variable `=x`)

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 maintained 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. Since this

completes the agreement operation, the `task` slot of the agreeing goal lexeme should also be updated and marked as `done`.

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

[py16]
```
>>> agreement.productionstring(name="done", string="""
...     =g>
...     isa goal_lexeme
...     task done
...     ==>
...     ~g>
...     """)
{'=g': goal_lexeme(category= , number= , task= done)}
==>
{'~g': None}
```

2.7 Running our first model

To start running the agreement model, we just have to add an appropriate chunk to the goal buffer. Recall that the ACT-R view of higher cognition is that it is goal-driven: if there is no goal, no productions will fire and the mind will not change state. We do this in [py17] below: we first declare our `goal_lexeme` type (line 1 in [py17]) and then add one such chunk to the goal buffer (lines 2-6; chunk 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")
>>> agreement.goal.add(actr.chunkstring(string="""
...     isa goal_lexeme
...     task agree
...     category 'verb'
...     """))
>>> agreement.goal
{goal_lexeme(category= verb, number= , task= agree)}
```

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()
>>> agreement_sim.run()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: retrieve')
(0.05, 'PROCEDURAL', 'RULE FIRED: retrieve')
```

```
(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, meaning= [[car]],      11
   number= sg, phonology= /kar/, 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) 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. Every cognitive step in the model takes by default 50 ms, i.e., 0.05 seconds – 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., if the current state of the mind satisfies the preconditions of any production rule. If the preconditions of multiple rules are satisfied, we have a conflict because only one production rule can fire at any given time. In that case, we need to select one rule ('resolve the conflict').

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 conditions 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 parallel 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 goal

lexeme has entered the active trigger_agreement state) 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 chapter 6, but we want to keep our first model simple so we use the default 50 ms retrieval time here.

At the 100 ms point, the memory retrieval process has been completed and 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 & 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 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
{goal_lexeme(category= verb, number= sg, task= done): array([ 0.2]),
   word(category= noun, meaning= [[car]], number= sg, phonology= /kar/,
   synfunction= subject): array([ 0.])}
```

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, a.k.a. 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 that point in a formal semantics intro course where the semantics for classical first order logic (FOL) has 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.

2.8 Appendix: The agreement model

File **ch2_agreement.py**:

```
"""
A basic model that simulates subject-verb agreement.
We abstract away from syntactic parsing, among other things.
"""

import pyactr as actr
import random

actr.chunktype("word", "phonology, meaning, category, number, syncat")
actr.chunktype("goal_lexeme", "task, category, number")

carLexeme = actr.makechunk(
    nameofchunk="car",
    typename="word",
    phonology="/ka:/",
    meaning="[[car]]",
    category="noun",
    number="sg",
    syncat="subject")

agreement = actr.ACTRModel()

dm = agreement.decmem
dm.add(carLexeme)

agreement.goal.add(actr.chunkstring(string="""
    isa goal_lexeme
    task agree
    category 'verb'"""))

agreement.productionstring(name="retrieve", string="""
    =g>
    isa goal_lexeme
    category 'verb'
    task agree
    ?retrieval>
    buffer empty
    ==>
```

```
    =g>
    isa goal_lexeme
    task trigger_agreement
    category 'verb'
    +retrieval>
    isa word
    category 'noun'
    syncat 'subject'
    """)

agreement.productionstring(name="agree", string="""
    =g>
    isa goal_lexeme
    task trigger_agreement
    category 'verb'
    =retrieval>
    isa word
    category 'noun'
    syncat 'subject'
    number =x
    ==>
    =g>
    isa goal_lexeme
    task done
    category 'verb'
    number =x
    """)

agreement.productionstring(name="done", string="""
    =g>
    isa goal_lexeme
    task done
    category 'verb'
    number =x
    ==>
    ~g>""")

if __name__ == "__main__":
    agreement_sim = agreement.simulation()
    agreement_sim.run()
    print("\nDeclarative memory at the end of the simulation:")
    print(dm)
```