Computing Dynamic Meanings

Adrian Brasoveanu, Jakub Dotlačil[1]

February 2, 2017

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

Contents

List of Figures

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
```

[1]

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 Python 3 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 `pyactr`, which comes with a variety of ways of working interactively with Python (IDE with Spyder as the editor, `anaconda` notebooks etc.). But none of this is required to run `ipython` and the code in this book.

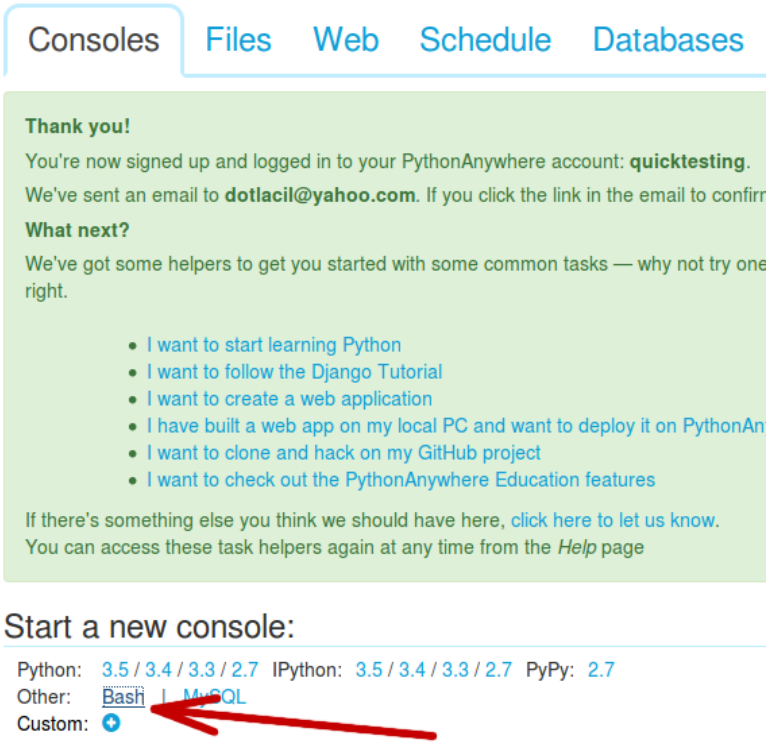
a. Go to www.pythonanywhere.com and sign up there.

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 1.1: 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:

```
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:

```
import pyactr
```

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 (Python 3) and

```
~/.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.")
```

on your computer and running scripts directly there.

Part I

Basics of ACT-R and modeling syntactic processing in self-paced reading tasks

Chapter 2

Basics of ACT-R

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 Python (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, since Python is currently much more widespread than Lisp and has a much larger and more diverse ecosystem of libraries, coding parts that do not directly pertain to the ACT-R model (like data manipulation / data munging, interactions with the operating system, displaying simulation results, incorporating them into tex / pdf documents etc.) are much better supported than in Lisp. Because of this, the programming language and programming-related issues stand less in the way of learning ACT-R. You can therefore focus on doing cognitive modeling for linguistic applications, examining and evaluating your models, and communicating your results, rather than spending a significant amount of time on issues having to do with the computational tools you need to run.

This book and the cognitive models we build and discuss are not intended as a comprehensive introduction and/or reference manual for ACT-R. For learning the theory behind ACT-R and its main applications in cognitive psychology, consider Anderson (1990); Anderson and Lebiere (1998); Anderson et al. (2004); Anderson (2007) among others. 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 / output and link them

[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.

to various components of the ACT-R theory or of the linguistic phenomenon or linguistic analysis we are modeling.

For example, when we want to discuss the code, we will display it as:

(1)
```
pyactr                                                                    1
```

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`. `3 + 2 == 6`.

When we want to discuss both the code and its output, we will display the code and output in the same way they would appear in your 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. – are numbered as shown in (1) above. Blocks of python code meant to be run interactively, together with their associated output, are 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. [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 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 by providing a detailed explanation of those constructs, they are no longer neutral. [...] At first glance, 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, spanning various points on the competence-performance spectrum; then they have to be empirically and methodologically evaluated in specific ways, as accounts of the specific phenomena they target.

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. 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 more 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 that formalize explicit

[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.

(quantitative) connections between semantic theorizing and experimental data. Our book should also be of interest to cognitive scientists other than linguists who are 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.

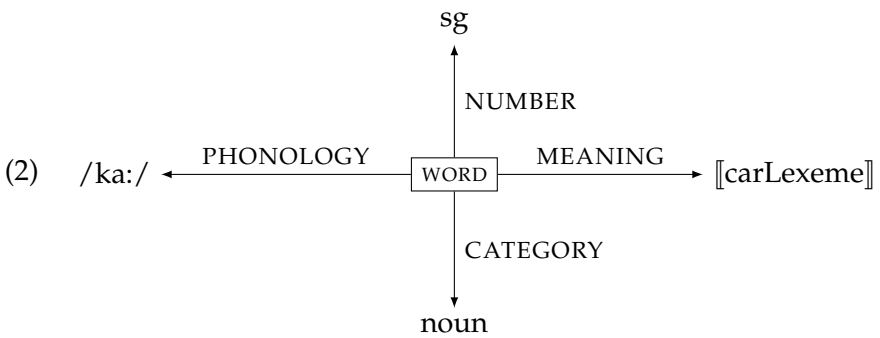
Procedural knowledge is knowledge that we display in our behavior (cf. Newell 1973). It is often the case that our procedural knowledge is internalized, we are aware that we have it but we would be hard pressed to explicitly and precisely describe it. Driving, swimming, riding a bicycle are examples 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.

2.3.1 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). 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 /ka:/ for the slot *phonology*, the value ⟦carLexeme⟧ 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 /ka:/, ⟦carLexeme⟧, *noun* and *sg*, respectively. Chunks (complex, non-primitive elements) are boxed, whereas primitive elements are sim-

ple 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 **??**). 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) $\underset{\text{WORD}}{\left[\text{PHONOLOGY: /ka:/ meaning: [\![carLexeme]\!] category: noun number: sg}\right]}$

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' specify preconditions and if these preconditions are true, then 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 a generalization: the fact is that the two rules are really just one 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 in fact 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 that (and given the convention that variables are signaled in ACT-R using '='), we can write: *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 `2 + 2 == 4`: 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 §**??**). After this, we can finally turn to the second type of knowledge in ACT-R: procedural knowledge / productions.

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

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

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

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 that chunk type has. This is just good housekeeping: by declaring the type and attributes associated with the type, we are clear from the start about the kind of objects we assume declarative memory stores.

Let's create a chunk type that will correspond to our lexical knowledge. 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")
```

The function pyactr creates a type chunktype, which consists of the following slots: word, phonology, meaning, category. The type, namely number, is the first argument of the function; the list of slots, with the slots separated by commas, is the second argument. After declaring the chunk type, we can create new chunks of this type.

```
[py4] >>> carLexeme = actr.makechunk(nameofchunk="car",\
      ...                         typename="word",\
      ...                         phonology="/ka:/",\
      ...                         meaning="[[car]]",\
      ...                         category="noun",\
      ...                         number="sg")
      >>> print(carLexeme)
      word(category= noun, meaning= [[car]], number= sg, phonology= /ka:/)
```

The chunk is created using the function "word". Every makechunk has two fixed arguments: makechunk ([py4], line 1) and nameofchunk ([py4], 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 in [py4], lines 3-6. 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 in [py4], line 7. Note that the order of slot-value pairs is different than the one we used when we declared the chunk: we defined typename 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 phonology, 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 chunk of that type; declaring types clarifies what kind of AVMs are needed in your model and provide a clear link between the phenomena and generalizations we are trying to model and the computational model itself. For this reason, if we don't specify the chunk type before declaring a chunk, pyactr will print a warning message. Among other things, this might help you debug your code – e.g., if you accidentally named your chunk type pyactr instead of the type "morphreme" you previously declared, you 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]

[3] See the "morpheme" and Python 3 documentation for more on warnings.

It is also recommended that you only use attributes you 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 sesssion that it would be useful to specify what syntactic function a word has (something like the argument structure attribute ARG-ST in HPSG). We didn't have that slot in our pyactr chunk. So let's create a new chunk carLexeme, which is like carLexeme2 except it adds this extra piece of information in the slot carLexeme. We will assume that the syncat value of syncat is carLexeme2:

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

Line 7 in [py5] is the new part. We are adding a new slot "subject", and assign it the value syncat. The command goes through successfully, as shown by the fact that we can print "subject"), but a warning message is issued (not displayed above): UserWarning: Chunk type word is extended with new attributes.

Another, perhaps more intuitive, way of specifying a chunk uses the function carLexeme2. When declaring chunks with chunkstring, the chunk type is provided as the value of the chunkstring-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 '/ka:/'                                              3
      ...     meaning '[[car]]'                                              4
      ...     category 'noun'                                                5
      ...     number 'sg'                                                    6
      ...     syncat 'subject'""")                                           7
      >>> print(carLexeme3)                                                 8
      word(category= noun, meaning= [[car]], number= sg, phonology= /ka:/,   9
         syncat= subject)                                                    10
```

The function isa provides the same functionality as chunkstring. The argument makechunk defines what the chunk consists of. The value pairs are written as a plain string. Notice that we use three quote marks, rather than one. 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 chunk, and a special slot is used for this, `string`. Notice that the resulting chunk is identical to the previous one, as shown on [py6], line 8.

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. PYACTR overloads standard comparison operators for these tasks, as shown below:

[py7]
```
>>> carLexeme2 == carLexeme2
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 that's an essential feature of ACT-R works: 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("synlabel", "category")
>>> noun = actr.makechunk(nameofchunk="noun",
...                       typename="synlabel",
...                       category="noun")
>>> noun < carLexeme
True
>>> noun < carLexeme2
True
```

!!! add content addressable memory to motivate chunk identity !!! add parallel vs. serial to motivate serial production application and single-chunk buffers !!! why a new implementation in Python: LISP implementation not easy to learn (Python is much more common) and not easy to interface (many more libraries in the Python ecosystem); also, ACT-R is a mathematical theory, not a black-box like software package; it can be easily reimplemented; the Python 2 ACT-R version wasn't trivial to update / refine, it wasn't aligned with the basic LISP ACT-R syntax and it's subsymbolic systems were not fully behaving in the LISP ACT-R way

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 is a limited throughput capacity: at any given time, it can carry only one chunk.

For ACT-R, the human mind is a specific system of modules and associated buffers within and across which chunks are stored and transacted. This flow of information is driven by productions. In this chapter, we will be concerned with only two major components of (the ACT-R architecture for) the human mind: procedural memory and declarative memory. Procedural memory stores productions. It 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 (the ACT-R view of human higher cognition is that it is fundamentally goal driven). Declarative memory stores chunks, and the buffer associated with the declarative memory module is 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()
```

The mind we intend to build is simply supposed to check for number agreement, hence the name of the variable for the ACT-R model (`pyactr`). 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
{}
```

`agreement` is an attribute of our `decmem` ACT-R model, and it stores the declarative memory module. The `agreement` and `retrieval` attributes store the retrieval and the goal buffer, respectively.

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

It is convenient to have a shorter alias for the declarative memory module, which we can do by introducing a new variable `goal` and assigning it the `dm` module:

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

We might want to add a chunk to our declarative memory, e.g., our `decmem` chunk. We add chunks by invoking the `carLexeme2` 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= /ka:/,
   syncat= subject): {0.0}}
```

Note that when we inspect `add`, we can see the chunk we just added. The chunk encoding is associated with the simulation time of the encoding. Since we have not yet run the model / started the model simulation, that time is 0.

2.6 Writing productions in `dm`

Recall that productions are essentially conditionals (*if*-statements), with the preconditions that need to be satisfied in the antecedent of the conditional and the action that is triggered if the preconditions are satisfied in the consequent. Consequently, productions have two parts: the preconditions precede the double arrow (`pyactr`) and the actions follow the arrow.

Let's now add some productions to our model that simulate a basic form of verb agreement.[4] Our model of subject-verb agreement will be blatantly oversimplified, but for now we focus on assembling and getting off the ground the basic architecture of the model / mind rather than on realistic processing models of linguistic behavior. 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 agreement 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 `==>`, then the number of the verb in the goal buffer should be the same, namely is `=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 `=x` that will make the simulation output more readable. In general, productions are created by the method `"retrieve"` associated with our ACT-R model, and they have two arguments (there is actually a third argument; more on that later): `productionstring` (the name of the production) and `name`, which provides the actual content of the production.

[py14]
```
>>> agreement.productionstring(name="retrieve", string="""
...      =g>
```

[4] See the appendix to this chapter.

```
...     isa goal_lexeme                                                    3
...     category 'verb'                                                    4
...     task agree                                                         5
...     ?retrieval>                                                        6
...     buffer empty                                                       7
...     ==>                                                                8
...     =g>                                                                9
...     isa goal_lexeme                                                    10
...     task trigger_agreement                                             11
...     category 'verb'                                                    12
...     +retrieval>                                                        13
...     isa word                                                           14
...     category 'noun'                                                    15
...     syncat '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= , syncat= subject)} 22
```

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 `string` on line 8 of [py14] above. The rule has two preconditions. The first one starts on line 2: `==>` indicates that this precondition will check that the chunk currently stored in the goal buffer (that what `=g>` encodes) is (that's what g encodes) of a particular kind: the chunk has to be a `=` (line 3) of category `goal_lexeme` (line 4), and the current task for this lexeme should be `'verb'` (line 5). The second precondition starts on line 6: `agree` indicates that this precondition will check whether the `?retrieval>` buffer is in a certain state (`retrieval`). The state is specified on line 7: the retrieval buffer needs to be `?` (no chunk should be stored there). In general, we can check for a variety of states that buffers could be in, for example: `empty` checks whether the goal buffer is full (whether it carries a chunk), `'?g> buffer full'` checks if the retrieval buffer is working on retrieving a chunk, and `'?retrieval> state busy'` 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 `'?retrieval> state error'` chunk by changing the current task from `goal_lexeme` to `agree`; the other features of the `trigger_agreement` chunk remain the same. And 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, starting on line 13: `goal_lexeme` 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 `+retrieval>` means). This chunk is our memory cue / query: we want to retrieve from declarative memory a chunk of type `+` that is a `word` and a `'noun'`.[5]

[5]Strictly speaking, it is not necessary to ensure that the retrieval buffer is empty before placing a retrieval request. The model would work just as well if the retrieval buffer is non-empty – the buffer would just be flushed first, and the memory cue would then be placed in it.

Memory queries / cues always consist of chunks, i.e., feature structures, and the retrieval process asks the declarative memory module to provide a larger chunk that the cue chunk is a part of. 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 `'subject'`, its category should be `PYACTR` and its syntactic category should be `word`.

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="""
...     =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
...     """)
{'=g': goal_lexeme(category= verb, task= trigger_agreement), '=retrieval':
   word(category= noun, meaning= , number= =x, phonology= , syncat=
   subject)}
==>
{'=g': goal_lexeme(category= verb, number= =x, task= done)}
```

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 (`'subject'`) a = of category `goal_lexeme` that is in an active state of agreeing (the current task is `'verb'`)
- lines 6-10: the chunk in the retrieval buffer is (`trigger_agreement`) a = of category `word` that is a `'noun'` and that has a number specification; call that number specification (whatever it is) `'subject'` (more precisely: take that number specification and assign it as value to the variable =x; we mark variable names in ACT-R by prefixing their names with x)

After checking 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 (the = sign in = on line 12) and its feature structure should be updated as follows: the type and category should stay the same (=g> and `goal_lexeme`, respectively), but a new number

specification should be added that has the same number specification 'verb' as the subject noun we have retrieved from declarative memory. This completes the agreement operation, so the task on the agreeing goal lexeme should also be updated and marked as =x.

The third and final production rule just mops things up: we are done, so the goal buffer is flushed and our simulation can end.

```
[py16] >>> agreement.productionstring(name="done", string="""                1
       ...     =g>                                                           2
       ...     isa goal_lexeme                                               3
       ...     task done                                                     4
       ...     category 'verb'                                               5
       ...     number =x                                                     6
       ...     ==>                                                           7
       ...     ~g>""")                                                       8
       {'=g': goal_lexeme(category= verb, number= =x, task= done)}           9
       ==>                                                                   10
       {'~g': None}                                                          11
```

The action on line 8 in [py16], namely done, simply discards the chunk present in the goal buffer.

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 ~g> type (line 1 in [py17]) and then add one such chunk to the goal buffer (lines 2-5; chunk are always added to buffers / modules using the method goal_lexeme). We check that the chunk has been added to the goal buffer by printing its contents (line 6); note that the number specification on line 7 is empty.

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

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

```
[py18] >>> simulation = agreement.simulation()                              1
       >>> simulation.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, meaning= [[car]],       11
   number= sg, phonology= /ka:/, syncat= 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 `simulation` command is the temporal trace of our model simulation. Each line specifies three elements: the first element is simulation time (in seconds); the second element is the module that is affected; and the third element is 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.

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. We will talk more about conflict resolution in due time; 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').

'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 `run()` in [**py14**] above; the rules is selected, as shown in [**py18**], line 4. The rule fires, and this takes the ACT-R default time of 50 ms – [**py18**], line 5. The state of our mind has changed as this time, and the following lines report on that new state: the goal buffer has been modified (line 6; the goal lexeme has entered the active `retrieve` state) and the retrieval buffer has started a memory retrieval procedure, which will take time to complete. The procedure module then looks for production rules to apply, and none can be fired in the current mental state (lines 8 and 9).

We therefore advance to the next simulation time (100 ms): at this point, the memory retrieval has been completed (line 10) and the retrieved chunk is reported (lines 11-12). Since

the mind is now in a new state, the conflict resolution procedure (rule collection & rule selection) yields one new rule that can fire (lines 13-14), namely the second production rule we discussed in [py15] above that we named `trigger_agreement`. The `agree` rule takes 50 ms to fire, so after a total simulation time of 150 ms (time 0.15), the rule has fired (line 15) and the chunk in the goal buffer has been modified: 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, so at time 0.2 the goal buffer is cleared, and no further rule can apply (lines 19-22).

When the goal buffer is cleared, the information stored in there does not disappear. The ACT-R architecture specifies that the cleared information is automatically transferred to the declarative memory. This is also the case here: our past goals become our present (newly acquired) memory facts. We check the final state of the declarative memory to see that this is indeed the case:

[py19]
```
>>> dm                                                                    1
{goal_lexeme(category= verb, number= sg, task= done): {0.2}, word(category= 2
   noun, meaning= [[car]], number= sg, phonology= /ka:/, syncat= subject):  3
   {0.0}}                                                                 4
```

And that's it: at its core, this is a simple framework for building process models. It is overly simplistic in many ways, but the main point is: we can now build explicit computational models for linguistic processes and behaviors. It is very similar to classical first and higher order logic: they are simple systems, and in many ways overly simplistic; but we can now start thing about natural language meaning and interpretation.

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__":
    x = agreement.simulation()
    x.run()
```

Bibliography

Anderson, John R. 1990. *The adaptive character of thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, John R. 2007. *How can the human mind occur in the physical universe?*. Oxford University Press.

Anderson, John R., Daniel Bothell, and Michael D. Byrne. 2004. An integrated theory of the mind. *Psychological Review* 111:1036–1060.

Anderson, John R., and Christian Lebiere. 1998. *The atomic components of thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Carpenter, Bob. 1992. *The logic of typed feature structures*. New York, NY, USA: Cambridge University Press.

Davies, Martin. 2001. Knowledge (explicit and implicit): Philosophical aspects. In *International encyclopedia of the social and behavioral sciences*, ed. N. J. Smelser and B. Baltes, 8126–8132. Elsevier.

Forster, Kenneth I. 1990. *Lexical processing.*. The MIT Press.

Frazier, Lyn, and Janet Dean Fodor. 1978. The sausage machine: A new two-stage parsing model. *Cognition* 6:291–325.

Fuchs, Albert. 1971. The saccadic system. *The control of eye movements* 343–362.

Hale, John. 2011. What a rational parser would do. *Cognitive Science* 35:399–443.

Hale, John T. 2014. *Automaton theories of human sentence comprehension*. Stanford: CSLI Publications.

Just, Marcel A., and Patricia A. Carpenter. 1980. A theory of reading: From eye fixations to comprehension. *Psychological Review* 87:329–354.

Just, Marcel A., Patricia A. Carpenter, and Jacqueline D. Woolley. 1982. Paradigms and processes in reading comprehension. *Journal of Experimental Psychology: General* 111:228–238.

Lewandowsky, S., and S. Farrell. 2010. *Computational modeling in cognition: Principles and practice*. Thousand Oaks, CA, USA: SAGE Publications.

Lewis, Richard, and Shravan Vasishth. 2005. An activation-based model of sentence processing as skilled memory retrieval. *Cognitive Science* 29:1–45.

Marslen-Wilson, William. 1973. Linguistic structure and speech shadowing at very short latencies. *Nature* 244:522–523.

Meyer, David E, and David E Kieras. 1997. A computational theory of executive cognitive processes and multiple-task performance: Part i. basic mechanisms. *Psychological review* 104:3.

Murray, Wayne S, and Kenneth I Forster. 2004. Serial mechanisms in lexical access: the rank hypothesis. *Psychological Review* 111:721.

Newell, A. 1990. *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Newell, Alan. 1973. Production systems: Models of control structures. In *Visual information processing*, ed. W.G. Chase et al., 463–526. New York: Academic Press.

Polanyi, Michael. 1967. *The tacit dimension*. London: Routledge and Kegan Paul.

Poore, Geoffrey M. 2013. Reproducible documents with pythontex. In *Proceedings of the 12th Python in Science Conference*, ed. Stéfan van der Walt, Jarrod Millman, and Katy Huff, 78–84.

Rayner, Keith. 1998. Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin* 124:372–422.

Reichle, Erik D, Alexander Pollatsek, Donald L Fisher, and Keith Rayner. 1998. Toward a model of eye movement control in reading. *Psychological review* 105:125.

Resnik, Philip. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the Fourteenth International Conference on Computational Linguistics*. Nantes, France.

Ryle, Gilbert. 1949. *The concept of mind*. London: Hutchinson's University Library.

Salvucci, Dario D. 2001. An integrated model of eye movements and visual encoding. *Cognitive Systems Research* 1:201–220.

Schilling, Hildur EH, Keith Rayner, and James I Chumbley. 1998. Comparing naming, lexical decision, and eye fixation times: Word frequency effects and individual differences. *Memory & Cognition* 26:1270–1281.

Staub, Adrian. 2011. Word recognition and syntactic attachment in reading: Evidence for a staged architecture. *Journal of Experimental Psychology: General* 140:407–433.

Steedman, Mark. 2001. *The syntactic process*. Cambridge, MA: MIT Press.

Tanenhaus, M. K., M. J. Spivey-Knowlton, K. M. Eberhard, and J. C. Sedivy. 1995. Integration of visual and linguistic information in spoken language comprehension. *Science* 268:1632–1634.