Language, Cognition, and Mind

Adrian Brasoveanu
Jakub Dotlačil

Computational Cognitive Modeling and Linguistic Theory

Springer Open

Language, Cognition, and Mind

Volume 6

Adrian Brasoveanu · Jakub Dotlačil

Computational Cognitive Modeling and Linguistic Theory

Springer Open

Adrian Brasoveanu
University of California Santa Cruz
Santa Cruz, CA, USA

Jakub Dotlačil
Utrecht University
Utrecht, The Netherlands

ISSN 2364-4109 ISSN 2364-4117 (electronic)
Language, Cognition, and Mind
ISBN 978-3-030-31844-4 ISBN 978-3-030-31846-8 (eBook)
https://doi.org/10.1007/978-3-030-31846-8

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword and Acknowledgments

We want it all. And so should you.

We want it all: this book used to have a very long subtitle—'Integrating generative grammars, cognitive architectures and Bayesian methods.' It was a mouthful, so we dropped it. But this very long subtitle was trying to summarize the main contribution of this book, which is to provide a formally and computationally explicit way to build theories that integrate generative grammars and cognitive architectures: integrated competence-performance theories for formal syntax and semantics. Not only that: once this rich, expansive space of linguistic theories opens up, we want to be able to *quantitatively* check their predictions against experimental data that is standard in psycholinguistics (forced choice experiments, self-paced reading, eye-tracking, etc.). We also want to be able to do a quantitative comparison for arbitrary linguistic and processing theories. And this is where Bayesian methods for parameter estimation and model comparison come in.

And so should you: this book is our best argument that linguists *can* actually have it all. Maybe not exactly (or not even nearly) in the form outlined in this book. That's OK. We are taking a formal and computational step on the path to a richer theoretical and empirical space for generative linguistics. And we hope you will join us in our building effort.

In our heart of hearts, we are formal semanticists, and we think of this book as taking some steps towards addressing one of the key challenges for formal semantics that Barbara Partee mentioned in her 2011 address titled *The Semantics Adventure*, namely "how to build formal semantics into real-time processing models—whether psychological or computational—that involve the integration of linguistic and not-specifically linguistic knowledge." (Partee 2011, p. 4)

One way to begin answering this challenge is to build a framework for mechanistic processing models that integrates work in the formal semantics tradition that started roughly with Montague (1970, 1973), and work on cognitive architectures—broad, formally explicit and unified theories of human cognition and cognitive behavior—a cognitive psychology research tradition that was explicitly established around the same time (Newell 1973a, b). This book is our first comprehensive attempt at building such a framework, and we see ourselves as following directly in the footsteps of Hans Kamp's original goal for Discourse Representation Theory. The classic Kamp (1981) paper begins as follows:

> Two conceptions of meaning have dominated formal semantics of natural language. The first of these sees meaning principally as that which determines conditions of truth. [...] According to the second conception meaning is, first and foremost, that which a language user grasps when he understands the words he hears or reads. [...] these two conceptions [...] have remained largely separated for a considerable period of time. This separation has become an obstacle to the development of semantic theory [...] The theory presented here is an attempt to remove this obstacle. It combines a definition of truth with a systematic account of semantic representations. (Kamp 1981, p. 189)

We are grateful to Chris Barker, Dylan Bumford, Sam Cumming, Donka Farkas, Morwenna Hoeks, Margaret Kroll, Dan Lassiter, Rick Nouwen, Abel Rodriguez, Amanda Rysling, Edward Shingler, Shravan Vasishth, Matt Wagers and Jan Winkowski, and to the participants in the UCSC LaLoCo lab in Spring 2017, the participants in the UCSC Semantics Seminar of Spring 2018 and the participants in our ESSLLI 2018 course for discussing with us various issues related to this book, and giving us feedback about various parts of the book. We are also grateful to the Editor of the Springer LCAM series Chungmin Lee, the Senior Publishing Editor for Springer Language Education & Linguistics Jolanda Voogd, and the Assistant Editors for Springer Language Education and Linguistics Helen van der Stelt and Anita Rachmat—this book would not have been possible without their continued help and support. We want to thank two anonymous reviewers for their comments on an earlier draft of this book. Finally, we want to thank the UCSC Socs-Stats cluster administrators, particularly Doug Niven, without whose support the computing-intensive parts of this research would not have been possible. This document has been created with LaTeX (Lamport 1986) and PythonTex (Poore 2013). This research was partially supported by a Special Research Grant awarded to Adrian Brasoveanu by the Committee on Research from UC Santa Cruz, by the NWO VENI 275-80-005 grant awarded to Jakub Dotlačil and by the NWO VC. GW17.122 grant. The NWO VC.GW17.122 grant and a grant from the Utrecht University library enabled us to provide open access to this book..

Finally, we want to thank Maria Bittner, Hans Kamp and Shravan Vasishth for their support of this project, which has been a long time coming. Maria Bittner kept reminding us that making a contribution to semantics that only we can make is one of the most important things to which we can aspire, and that having an idea is only half the work—the other half is spreading the word. Hans Kamp has been an outstanding mentor and role-model, providing much-needed encouragement at crucial junctures during this project. His continued emphasis on the importance of a representational level for natural language interpretation has constantly guided the work we report on here. Shravan Vasishth provided extremely helpful and supportive feedback on an earlier version of the book, and helped us identify a suitable title that is both descriptive and concise. Shravan's work on computational cognitive models for sentence comprehension was one of the main sources of inspiration for us, and his support means a lot.

The usual disclaimers apply.

We dedicate this book to our children J. Toma Brasoveanu, Willem Dotlačil and Klaartje Dotlačil, whose births and early childhoods overlapped with the birth and maturation of this project.

In memoriam: we also want to acknowledge that discussions with friend and mentor Ivan Sag (1949–2013) and his work on competence-performance issues in generative grammar (Sag 1992; Sag and Wasow 2011) were a major source of inspiration for this work.

Keywords: Semantics • Processing • Computational psycholinguistics • ACT-R • Discourse representation theory • Cognitive modeling • Bayesian inference • Python

Contents

Chapter 1
Introduction

In this brief chapter, we summarize the background knowledge needed to be able to work through the book (Sect. 1.1). After that, we provide an overview of the remainder of the book (Sect. 1.2).

1.1 Background Knowledge

The present book interweaves approaches that are often treated separately, namely cognitive modeling, (Bayesian) statistics, (formal) syntax and semantics, and psycholinguistics. Given the wide range of frameworks and approaches, we try to presuppose as little possible, so that readers coming from different fields can work through (almost) all the material. That said, the book is mainly geared towards linguists, so readers are expected to have a basic grasp of formal syntax and semantics. The overwhelming majority of the cognitive phenomena that we discuss and model in this book are associated with natural language (English) comprehension, and we will generally presuppose the reader is familiar with the basic linguistic representations and operations involved in modeling these phenomena.

We take a hands-on approach to cognitive modeling in this book: we discuss theories and hypotheses, but we also focus on actually implementing the models (in Python). While it is possible to read the book without developing or running any code, we believe that going through the book this way misses important aspects of learning cognitive modeling. For this reason, we strongly encourage readers to run and modify our code, as well as develop their own models as they proceed. Cognitive modeling, like any other technical endeavor, is not a spectator sport: learning is doing. But doing cognitive modeling from scratch can be a daunting task. To simplify this, we created a Python package, `pyactr`, that will help readers focus only on those features of the implementation of cognitive models that are theoretically relevant.

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_1

Instructions for how to install the package, as well as other practical details regarding programming and Python are discussed here[1]:

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

This book is not an introduction to programming, in general or in Python. Whenever it is possible, we briefly cover concepts needed to understand code snippets presented in the book. However, readers should keep in mind that such explanations are included merely to make the process of going through the text a little smoother. In order to gain a deeper understanding, it will be necessary to consult Python textbooks (or online courses). Downey (2012) is a good starting point to learn Python; see Ramalho (2015) for a slightly more advanced discussion. We chose Python for this book because it is beginner-friendly and it is currently (as of 2019) the most popular language for general data wrangling, data visualization and analysis, machine learning and scientific computing. Python's ease-of-use and library ecosystem for scientific computing is currently unrivaled.[2]

In sum, we believe it is possible to read the book without any knowledge of Python. But understanding Python will provide better insight into the models we build, and it will enable our readers to use the concepts and tools we develop here in their own research.

1.2 The Structure of the Book

The book is structured as follows.

Chapter 2 introduces the ACT-R cognitive architecture and the Python3 implementation `pyactr` we use throughout the book. We end with a basic ACT-R model for subject-verb agreement.

Chapter 3 introduces the basics of syntactic parsing in ACT-R. We build a top-down parser and learn how we can extract intermediate stages of `pyactr` simulations. This enables us to inspect detailed snapshots of the cognitive states that our processing models predict.

Chapter 4 introduces a psycholinguistically realistic model of syntactic parsing (left-corner parsing). We also introduce the vision and motor modules. These mod-

[1]If you encounter any issues with the package and/or the code discussed in this book, please go the public forum associated with the `pyactr-book` repository and open an issue there. The forum is located here:

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

[2]But see this blog post, for example, for a more nuanced—and ultimately different—opinion:

https://github.com/matloff/R-vs.-Python-for-Data-Science.

Chances are good that sooner or later, one will have to become familiar with both Python and R if one works in a field connected to data science (in its broadest sense, e.g., as characterized here: https://cra.org/data-science/).

ules enable our cognitive models to interact with the environment just as human participants do in a psycholinguistic experiment. This is an important contribution to the current psycholinguistics literature, which focuses almost exclusively on modeling the declarative memory contribution to natural language processing. Instead, our models make use of the full ACT-R cognitive architecture, and explicitly include (i) the procedural memory module, which is the backbone of all cognitive processes, as well as (ii) the interface modules, motor and vision specifically.

Chapter 5 introduces the basics of Bayesian methods for data analysis and parameter estimation, and the main computational tools we will use for Bayesian modeling in Python3. Bayesian modeling enables us to estimate the subsymbolic parameters of ACT-R models for linguistic phenomena, and our uncertainty about these estimates. Applying Bayesian methods to ACT-R cognitive models is a contribution relative to the current work in the psycholinguistic ACT-R modeling literature, and ACT-R modeling more generally. Parameters in ACT-R models are often tuned manually by trial and error, but the availability of the new `pyactr` library introduced in the present monograph, in conjunction with already available, excellent libraries for Bayesian modeling like `pymc3`, should make this practice obsolete and replace it with the modeling and parameter-estimation workflow now standard in statistical modeling communities.

Chapter 6 introduces the (so-called) subsymbolic components needed to have a realistic model of human declarative memory, and shows how different cognitive models embedded in Bayesian models can be fit to the classical forgetting data from Ebbinghaus (1913). In addition to estimating the parameters of these models and quantifying our uncertainty about these estimates, we are also able to compare these models based on how good their fit to data is. We limit ourselves to plots of posterior predictions and informal model comparison based on those plots.

Chapter 7 brings together the Bayesian methods introduced in Chap. 5 and the subsymbolic components of the ACT-R architecture introduced in Chap. 6 to construct and compare a variety of ACT-R models for the lexical decision data in Murray and Forster (2004). We begin by comparing two ACT-R models that abstract away from the full ACT-R architecture and focus exclusively on the way declarative memory modulates lexical decision. Once the better model is identified, we show how it can be integrated into three different end-to-end models of lexical decision in `pyactr`. These models incorporate the full ACT-R architecture and are able to realistically simulate a human participant in lexical decision tasks, from the integration of visual input presented on a virtual screen to providing the requisite motor response (key presses). Crucially, these three Bayes+ACT-R models differ in symbolic (discrete, non-quantitative) ways, not only in subsymbolic (quantitative) ways. Nonetheless, our Bayes+ACT-R framework enables us to fit them all to experimental data and to compute quantitative predictions (means and credible intervals) for all of them. That is, we have a general procedure to quantitatively compare fully formalized qualitative (symbolic) theories. The chapter also discusses predictions of the ACT-R left-corner parser from Chap. 4 for the Grodner and Gibson (2005) processing data. This provides another example of how the framework enables us to consider distinct symbolic hypotheses about linguistic representations and parsing processes, formalize them and quantitatively compare them.

Chapters 8 and 9 build the first (to our knowledge) fully formalized and computationally implemented psycholinguistic model of the human semantic parser/interpreter that explicitly integrates formal semantics theories and an independently-motivated cognitive architecture (ACT-R), and fits the resulting processing models to experimental data. Specifically, we show how Discourse Representation Theory (DRT; Kamp 1981; Kamp and Reyle 1993[3]) can be integrated into the ACT-R cognitive architecture.

Chapter 8 focuses on the organization of Discourse Representation Structures (DRSs) in declarative memory, and their storage in and retrieval from declarative memory. The chapter argues that the fan effect (Anderson 1974; Anderson and Reder 1999) provides fundamental insights into the memory structures and cognitive processes that underlie semantic evaluation, which is the process of determining whether something is true or false relative to a database of known facts, i.e., a model in the parlance of model-theoretic semantics.

Chapter 9 builds on the model in Chap. 8 and formulates an explicit parser for DRSs that works in tandem with a syntactic parser and that has visual and motor interfaces. The resulting model enables us to fully simulate the behavior of participants in self-paced reading tasks targeting semantic phenomena. We use it to account for the experiments reported in Brasoveanu and Dotlačil (2015a), which study the interaction between (i) cataphoric pronouns and cataphoric presuppositions on one hand, and (ii) the dynamic meanings of sentential connectives, specifically, conjunctions versus conditionals, on the other hand.

An extreme, but clear way to state the main theoretical proposal made in Chap. 9 is the contention that anaphora, and presupposition in general, are properly understood as *processing-level* phenomena that guide and constrain memory retrieval processes associated with incremental interpretation. That is, they guide and constrain the cognitive process of integration, or linking, of new and old semantic information. Anaphora and presupposition have semantic effects, but are not exclusively, or even primarily, semantics. The proper way to analyze them is as a part of the processing component of a broad theory of natural language interpretation. This proposal is very close in spirit to the DRT account of presupposition proposed in van der Sandt (1992); Kamp (2001a, b), among others. Kamp (2001b), with its extended argument for and extensive use of *preliminary representations*—that is, meaning representations that explicitly include unresolved presuppositions—is a particularly close idea.

Finally, Chap. 10 outlines several directions for future research.

[3]See also File Change Semantics (FCS; Heim 1982) and Dynamic Predicate Logic (DPL; Groenendijk and Stokhof 1991).

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

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

2.1 Cognitive Architectures and ACT-R

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

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

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

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

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_2

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 ACT-R in Cognitive Science and Linguistics

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

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

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

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

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

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

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

2.3 ACT-R Implementation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4 Knowledge in ACT-R

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

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

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

2.4.1 Declarative Memory: Chunks

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

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

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

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

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

2.4.2 Procedural Memory: Productions

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

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

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

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

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

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

2.5 The Basics of `pyactr`: Declaring Chunks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.6 Modules and Buffers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.7 Writing Productions in pyactr

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.8 Running Our First Model

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

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

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

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

[py18]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.9 Some More Models

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

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

2.9.1 The Counting Model

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We can now run the counting model:

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

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

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

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

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

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

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

2.9.2 *Regular Grammars in ACT-R*

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

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

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

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

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

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

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

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

We need only three rules:

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

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

[py29]
```
>>> regular_grammar.productionstring(name="NP ==> N NP", string="""      1
...     =g>                                                             2
...     isa         goal_chunk                                          3
...     mother      NP                                                  4
...     daughter1   None                                                5
...     daughter2   None                                                6
...     state       rule                                                7
...     ==>                                                             8
...     =g>                                                             9
...     isa         goal_chunk                                          10
...     daughter1   N                                                   11
...     daughter2   NP                                                  12
...     state       show                                                13
... """)                                                                14
{'=g': goal_chunk(daughter1= None, daughter2= None, mother= NP, state= rule)}15
==>                                                                     16
{'=g': goal_chunk(daughter1= N, daughter2= NP, mother= , state= show)}  17
```

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

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

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

[py31]
```
>>> regular_grammar.productionstring(name="get new mother", string="""   1
...     =g>                                                             2
...     isa         goal_chunk                                          3
...     daughter2   =x                                                  4
...     daughter2   ~None                                               5
...     state       rule                                                6
...     ==>                                                             7
...     =g>                                                             8
...     isa         goal_chunk                                          9
...     mother      =x                                                  10
...     daughter1   None                                                11
...     daughter2   None                                                12
... """)                                                                13
{'=g': goal_chunk(daughter1= , daughter2= =x~None, mother= , state= rule)}14
==>                                                                     15
{'=g': goal_chunk(daughter1= None, daughter2= None, mother= =x, state= )} 16
```

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

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

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

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

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

2.9.3 Counter Automata in ACT-R

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

All the code discussed in this chapter is available on GitHub as part of the repository https://github.com/abrsvn/pyactr-book. If you want to examine it and run it, install pyactr (see Chap. 1), download the files and run them the same way as you would any other Python3 script. Links to the specific files that contain the models discussed in this chapter are provided below:

File **ch2_agreement.py**:

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

File **ch2_count.py**:

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

File **ch2_regular_grammar.py**:

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

File **ch2_counter_automaton.py**:

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

Chapter 3
The Basics of Syntactic Parsing in ACT-R

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

3.1 Top-Down Parsing

Now that the basic ACT-R cognitive architecture is in place and we're more familiar with its specific implementation in `pyactr`, let us build a basic model of syntactic parsing. Specifically, we will build a top-down parser, i.e., a parser that uses the grammar to make predictions about the sentential structure of the upcoming input.

There are three properties of the human parser that we want our model to capture (Marslen-Wilson 1973, Frazier and Fodor 1978, Tanenhaus et al. 1995, Steedman 2001, Hale 2011 among others):

i. the parser is *incremental*: syntactic parsing and semantic interpretation do not lag significantly behind the perception of individual words;
ii. the parser is *predictive*: the processor forms explicit representations of words and phrases that have not yet been heard;
iii. finally, the parser *satisfies the competence hypothesis*: understanding a sentence/discourse involves the recovery of the structural description of that sentence/discourse on the syntax side, and of the meaning representation on the semantic side.

A top-down parser satisfies these conditions and it has the pedagogical advantage of being very simple (too simple, in fact, to be cognitively plausible). It is, therefore, a good place to start.

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

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

(1)

S	→	NP VP
NP	→	ProperN
VP	→	V NP
ProperN	→	Mary
ProperN	→	Bill
V	→	likes

For simplicity, we assume that we have only two proper names in our language and one transitive verb. Our goal is to build a top-down parser that is able to analyze the sentence *Mary likes Bill*. We assume the sentence is presented to the comprehender one word at a time in the manner of self-paced reading tasks (Just et al. 1982). In such tasks, the words are hidden and only one word is uncovered at a time with a spacebar press. The human reader decides when to press the spacebar to uncover the next word (which automatically hides the current word), hence the name of self-paced reading. So reading our sentence *Mary likes Bill* will happen in four successive stages. In one such version of self-paced reading (the so-called non-cumulative moving-window paradigm), the whole process would look as in (2) below.

(2)

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

Self-paced reading tasks mimic an essential aspect of naturally-occurring language comprehension with auditory stimuli: the signal is strictly linearly and strictly incrementally presented one word at a time. Just as in naturally-occurring verbal interactions, and unlike in normal reading situations, the linguistic signal cannot be 'rewound' to previous words—we cannot just look back and reread previous parts of the text—or 'fast-forwarded' to subsequent words—we cannot jump ahead to parts of the text that do not immediately follow the word currently being read.

With the empirical task fully characterized as a self-paced reading task, we can proceed to the characterization of our processing model. A top-down parser can be thought of as a push-down automaton, i.e., an automaton that has a basic form of memory represented as a stack. The stack stores parsing goals and subgoals in a strict, total order and these goals are accomplished one at a time by accessing the top of the stack. In our case, the parsing goals are simply syntactic categories that have to be parsed, i.e., that have to be identified in the incoming string.

For example, when we start the parsing process, we push the initial goal of parsing an S node onto the stack. The stack has now only one goal in it, namely 'parse an S', and the goal sits at the top of the stack.

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

We pop goals off the stack one at a time: we can only look at the top of the stack and remove the current top goal when this goal is (i) accomplished or (ii) broken down exhaustively into subgoals. For example, we will pop the ‘parse an S’ goal off the stack when we apply the first grammar rule in (1) above and replace this goal with two subgoals: first parse an NP (i.e., identify an NP in the incoming word input), then parse a VP. The resulting stack will now have two goals: the top one is ‘parse an NP’, and the one below it is ‘parse a VP’.

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

The parser works by modifying the contents of its stack based on two pieces of information: the top element on the stack and, possibly, the current word that has to be parsed (the leftmost word in the incoming string of words).

We can sum up top-down parsing as a parsing strategy that applies two algorithm schemata, *expand* and *scan*, in this order (see Hale 2014 among others for an introduction):

(5) Top-down parsing rules:

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

b. *scan*: if the top of the stack has a terminal symbol (a symbol like *V* or *ProperN* that rewrites to a lexical item, that is, a part of speech) and w, the leftmost word to be parsed, is of that part of speech, then pop the terminal symbol off the stack and remove w from the word string that is to be parsed.

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

3.2 Building a Top-Down Parser in `pyactr`

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

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

We should now specify the types of chunks we need. We will have one type for parsing goals. The parsing goal will keep track of:

- the stack content: we only need two positions in the stack for our current purposes—the top and the bottom of the stack; this is a consequence of the fact that our grammar (1) generates at most binary branching trees with no left recursion (cf. Resnik 1992);
- the current word being parsed (if any);

- the current task of the parser, that is, the current state our parsing model is in—basically, 'parsing' if the parse is still ongoing, and 'done' if the parsing is finished.

[py2]
```
>>> actr.chunktype("parsing_goal",
...                "stack_top stack_bottom parsed_word task")
```

The second chunk type we need to declare is one that will enable us to represent the incoming sentence, i.e., the word string to be parsed. This might seem counter-intuitive: why should we represent the sentence to be parsed in a chunk? The sentence is external to the agent, it's what the agent reads or hears. However, at this point we have no way of representing the surrounding environment and the basic input/output interfaces between the mind and the environment. We therefore have to represent a sentence internally as a chunk. When we introduce the vision and motor modules in Chap. 4, we will be able to develop a more intuitive and elegant solution.

The chunk type for sentences only needs to store three words, since our target sentence is that long:

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

3.2.1 *Modules, Buffers, and the Lexicon*

Let us now initialize the model and set up more convenient ways of accessing the declarative memory module and the goal buffer:

[py4]
```
>>> parser = actr.ACTRModel()
>>> dm = parser.decmem
>>> g = parser.goal
```

The goal buffer will store a `parsing_goal` chunk, which carries the information that drives the parsing process, and which is updated throughout that process. But we also need to store the word sequence that we need to parse, so we will create a second buffer that is similar to the goal buffer and that will store the sentence to be parsed.

Having two goal-like buffers is not uncommon in ACT-R. The first buffer is the actual goal buffer, which keeps track of the information driving the cognitive process. The other one is the *imaginal* buffer. This buffer is associated with the imaginal module and maintains an internal image of the information associated with the current cognitive process, thereby providing contextual information relevant for the current task. Thus, storing the sentence to be parsed in the imaginal buffer is an acceptable approximation of the cognitive behavior we're trying to model.

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

In [py5], we create a new goal buffer, the `imaginal` buffer. The string `"imaginal"` sets the name under which the model will recognize and access the buffer (e.g., in production rules). The delay attribute of the imaginal buffer is the time needed to encode/set a chunk in the buffer, which is 0.2 s (200 ms). This is the default ACT-R value for this buffer, in contrast to the goal buffer which sets a chunk immediately. Finally, **[py5]** assigns this new buffer to a variable imaginal so that we can access it more easily in the Python interpreter.

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

Thus, the cognitive context in the sense of 'the current state of the buffers' has a function similar to variable assignments in first-order logic. Assignments in first-order logic provide the current context of interpretation relative to which upcoming expressions are interpreted. Similarly, the state of the buffers in an ACT-R model of the mind provide the context for the next step in the cognitive process.

We can even extend this analogy to models, i.e., to the other parameter that the interpretation function in first-order logic is relativized to. The ACT-R counterpart of a first-order logic model is the content of the modules, particularly the facts stored in declarative memory and the rules stored in procedural memory.

We can now add chunks to the goal and imaginal buffers:

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

The goal buffer switches to an active parsing state/task, and the current parsing goal, i.e., the top of the stack, is set to parsing a sentence (S). In the imaginal buffer, we set the sentence to be parsed to *Mary likes Bill*.

We are now ready to start answering the main question of the chapter: how do we implement the top-down parser itself? We will assume that the grammar and associated parsing rules are part of the procedural module, i.e., they are encoded in production rules. This contrasts with lexical information, which is commonly encoded in declarative memory. See Lewis and Vasishth (2005) for more discussion and arguments for this division of labor between declarative and procedural memory when encoding the lexicon and the grammar and parser.

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

[py7]
```
>>> actr.chunktype("word", "form cat")
>>> dm.add(actr.chunkstring(string="""
...     isa  word
...     form Mary
...     cat  ProperN
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa  word
...     form Bill
...     cat  ProperN
... """))
>>> dm.add(actr.chunkstring(string="""
...     isa  word
...     form likes
...     cat  V
... """))
>>> dm
{word(cat= ProperN, form= Mary): array([0.]),
 word(cat= ProperN, form= Bill): array([0.]),
 word(cat= V, form= likes): array([0.])}
```

3.2.2 *Production Rules*

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

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

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

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

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

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

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

The third production rule expands VP into V and NP:

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

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

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

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

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

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

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

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

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

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

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

[py14]
```
>>> parser.productionstring(name="print parsed word", string="""   1
...     =g>                                 2
...     isa  parsing_goal                   3
...     task printing                       4
...     =imaginal>                          5
...     isa   sentence                      6
...     word1 ~None                         7
...     ==>                                 8
...     !g>                                 9
...     show parsed_word                    10
...     =g>                                 11
...     isa         parsing_goal            12
...     task        parsing                 13
...     parsed_word None                    14
... """)                                    15
{'=g': parsing_goal(parsed_word= , stack_bottom= ,     16
             stack_top= , task= printing),             17
 '=imaginal': sentence(word1= ~None, word2= , word3= )}   18
==>                                                    19
{'!g': ([(['show', 'parsed_word'], {})], {}),          20
 '=g': parsing_goal(parsed_word= None, stack_bottom= , 21
             stack_top= , task= parsing)}              22
```

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

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

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

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

3.3 Running the Model

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

[py16]
```
>>> parser_sim = parser.simulation()
>>> parser_sim.run()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: expand: S ==> NP VP')
(0.05, 'PROCEDURAL', 'RULE FIRED: expand: S ==> NP VP')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP ==> ProperN')
(0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP ==> ProperN')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Mary)')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: scan: word')
(0.25, 'PROCEDURAL', 'RULE FIRED: scan: word')
```

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

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

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

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

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

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

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

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

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

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

3.4 Failures to Parse and Taking Snapshots of the Mind When It Fails

We can run the parser on ungrammatical sentences to see if, and how exactly, it fails. Let's try to parse the word sequence *Bill Mary likes*. The parser should fail while parsing the second word *Mary* because the noun does not match the parser's expectation to see a verb.

We add the relevant chunks to the goal and imaginal buffers and start a new simulation. Note that, in general, it is recommended to reset the declarative memory module (and various buffers) before rerunning a model simulation.

A simple way to reset the model is to reinitialize it from scratch, that is, restart with `parser = actr.ACTRModel()` etc. You can take a look at the code for the more advanced models in Chaps. 7, 8 and 9 to see how to reset the state of a model without restarting it from scratch, so that multiple simulations with the same initial model state can be run.

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

Just as before, our goal is to parse a sentence `S` (**[py18]**, line 4), namely *Bill Mary likes* (lines 8–10). The parser correctly parses the first word *Bill* and prints it (line 40). But the parsing process stops after 450 ms because the word *Mary* retrieved from declarative memory is of category ProperN (line 55). The top of the goal stack, however, stores the category V, which is what the parser was expecting to retrieve (lines 48–49).

To facilitate the inspection of simulations and models, `pyactr` provides a way to advance simulations one step at a time, rather than letting them run from beginning to end. This makes it easy to check the internal state of the buffers, as well as to diagnose/debug our models, e.g., if the model gets stuck in an infinite loop. Let's run the simulation in **[py18]** again and go through it step by step.

[py19]
```
>>> g.add(actr.chunkstring(string="""
...     isa       parsing_goal
...     task      parsing
...     stack_top S
... """))
>>> imaginal.add(actr.chunkstring(string="""
...     isa   sentence
...     word1 Bill
...     word2 Mary
...     word3 likes
... """))
>>> parser_sim3 = parser.simulation()
>>> parser_sim3.step()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
```

Very little happens in the first step: the parser simply enters a 'conflict resolution' state in which it identifies the rules that can be fired given the initial cognitive state (that is, the initial state of the buffers).

Let's go through some more steps. To do that, we use the method `steps` with a parameter that provides the exact number of steps the simulation should advance through. In **[py20]**, we advance 10 steps, as reflected in the 10 lines of simulation output.

[py20]
```
>>> parser_sim3.steps(10)
(0, 'PROCEDURAL', 'RULE SELECTED: expand: S ==> NP VP')
(0.05, 'PROCEDURAL', 'RULE FIRED: expand: S ==> NP VP')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP ==> ProperN')
(0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP ==> ProperN')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
```

Let's now advance our simulation to the point where the rule `"scan: word"` has just fired. To be able to do that, we have to be able to check the current event, i.e., the most recent step taken in the simulation, and stop when this event is a `"scan: word"`-rule firing.

The current event is an attribute of the simulation. For example, the current event in our simulation is a ProperN retrieval:

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

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

- `time`—the simulation time at which the event occurred (150 ms in our case),
- `proc`—the module or buffer that is affected (procedural memory in our case), and
- `action`—the cognitive action that has taken place.

Let us now advance to the first firing of the `"scan: word"` rule. We do this by running a `while` loop in the Python interpreter: the command on line 2 in [py22] below, i.e., advance one step through the simulation, should be executed while the condition on line 1 is satisfied. That condition says that the `action` attribute of the current event should *not* be a `"scan: word"` firing. Note that `!=` is non-identity in Python; `!` is customarily used for negation in programming languages, and it is distinct from ACT-R negation `~`.

[py22]
```
1  >>> while parser_sim3.current_event.action != 'RULE FIRED: scan: word':
2  ...     parser_sim3.step()
3  ...
4  (0.15, 'g', 'MODIFIED')
5  (0.15, 'retrieval', 'START RETRIEVAL')
6  (0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
7  (0.15, 'PROCEDURAL', 'NO RULE FOUND')
8  (0.2, 'retrieval', 'CLEARED')
9  (0.2, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Bill)')
10 (0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
11 (0.2, 'PROCEDURAL', 'RULE SELECTED: scan: word')
12 (0.25, 'PROCEDURAL', 'RULE FIRED: scan: word')
```

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

[py23]
```
>>> g
{parsing_goal(parsed_word= Bill, stack_bottom= None,
              stack_top= VP, task= printing)}
>>> imaginal
{sentence(word1= Bill, word2= Mary, word3= likes)}
>>> parser.retrieval
{word(cat= ProperN, form= Bill)}
```

Let us now advance to the point where the parsing process failed. We will step through the simulation until the `action` attribute of the current event starts with the string `'RETRIEVED'`. That will be the point where the second word in our string, namely *Mary* has been retrieved:

[py24]
```
>>> while not parser_sim3.current_event.action.startswith('RETRIEVED'):
...     parser_sim3.step()
...
(0.25, 'g', 'MODIFIED')
(0.25, 'imaginal', 'MODIFIED')
(0.25, 'retrieval', 'CLEARED')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: print parsed word')
(0.3, 'PROCEDURAL', 'RULE FIRED: print parsed word')
parsed_word Bill
(0.3, 'g', 'EXECUTED')
(0.3, 'g', 'MODIFIED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
```

```
(0.3, 'PROCEDURAL', 'RULE SELECTED: expand: VP ==> V NP')
(0.35, 'PROCEDURAL', 'RULE FIRED: expand: VP ==> V NP')
(0.35, 'g', 'MODIFIED')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: retrieve: V')
(0.4, 'PROCEDURAL', 'RULE FIRED: retrieve: V')
(0.4, 'g', 'MODIFIED')
(0.4, 'retrieval', 'START RETRIEVAL')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.45, 'retrieval', 'CLEARED')
(0.45, 'retrieval', 'RETRIEVED: word(cat= ProperN, form= Mary)')
```

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

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

And the cause of the parsing failure is apparent: the retrieval buffer stores a ProperN while the top of the parsing goal stack, i.e., our current parsing expectation/prediction, is a V. The parser therefore halts before the second word in our sentence can be scanned, as shown by the unchanged chunk in the imaginal buffer.

3.5 Top-Down Parsing as an Imperfect Psycholinguistic Model

It is, however, not enough for our parser to correctly parse grammatical sentences and fail for ungrammatical ones. Our top-down ACT-R parser is not simply an implementation of an arbitrary parsing algorithm that is satisfactory as long as it works correctly. This parser is meant to be a limited, but realistic model of a certain kind of human cognitive behavior, namely syntactic parsing in comprehension-like tasks (self-paced reading). Is our parser even remotely adequate as a psycholinguistic model?

One of the empirical adequacy desiderata for our parser is that the temporal trace of parsing a sentence should correspond to the temporal trace of an average human participant completing the same task. For example, we see that our parser takes 800 ms to parse the sentence *Mary likes Bill*. This is roughly correct.

But there are various other properties of our parser that are more worrying. For one, the parser requires this much time while abstracting away from what human participants have to do during an actual self-paced reading task: internalizing visual information, projecting sentence meaning, executing motor actions (pressing keys) etc., so ultimately 800 ms might be too much given the very narrow amount of work our parser actually does.

Another issue is that retrieving lexical information always takes 50 ms in our current models and simulations, but this is hardly realistic. We know that lexical retrieval is dependent on various factors: word frequency, priming etc. These factors are completely ignored here.

Finally, top-down parsers work well for right-branching structures like the sentence *Mary likes Bill*, but they have significant difficulties with left branching structures. For such structures, the parser would have to store as many symbols on the stack as there are levels of embedding. Since every expansion of a grammar rule takes 50 ms, we expect left branching structures with n levels of embedding to take $50 * n$ ms. This is at odds with actual human performance (see Johnson-Laird 1983; Abney and Johnson 1991; Resnik 1992).[1] The main reason for this is that our parser generates predictions about syntactic structure exclusively based on the grammar and completely ignores the actual evidence (the sentence to be parsed) until it reaches a terminal on the leftmost branch.

In fact, purely top-down parsers consult the evidence (the word string) only after they predict all the way to lexical items. That is, such pure top-down parsers would place memory retrieval requests based on the terminal at the top of the parsing goal stack. For example, if a ProperN is at the top of the stack, they would retrieve an arbitrary ProperN from declarative memory and only after that, they check whether the form of the retrieved ProperN matches the leftmost word to be parsed. If not, a new retrieval request would be placed for a new ProperN in hopes that the form of that new chunk would match the word to be parsed. In the worst case, such a purely top-down parser would retrieve all chunks of category ProperN one at a time from declarative memory and, finally, identify the one whose form matches the current word to be parsed.

The temporal trace of such a parser would be very far from the temporal trace of an average human participant completing the same task: if the lexicon contains 20 chunks of ProperN category, and a retrieval takes around 50 ms, it would take a full second to parse the first word in the sentence *Mary likes Bill* in the worst-case scenario. And this ignores the time needed to verify that 19 of the retrieved chunks are mismatches, and then the time needed to backtrack and restart the retrieval process.

Thus, a more plausible human parser would consult the evidence, i.e., the word string to be parsed, earlier and more often in the parsing process. Our top-down parsing strategy needs to be complemented by a bottom-up parsing strategy. In principle, we could switch from a purely top-down parser to a purely bottom-up parser that is completely driven by the evidence. Such a parser would be incremental, but it would not be predictive in the same way that the human parser seems to be. We will therefore not explore purely bottom-up (shift-reduce) parsers and instead move directly to left-corner parsers, which combine top-down and bottom-up features: they can be thought of as predictive top-down parsers with incremental bottom-up filtering. The next chapter introduces left-corner parsers and models them in ACT-R and `pyactr`.

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

3.6 Appendix: The Top-Down Parser

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

File **ch3_topdown_parser.py**:

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

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

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

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

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

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

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

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

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_4

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.1 The Visual Module

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

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

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

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

The same applies to the `screen_y` slot.

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

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

4.1.2 The Motor Module

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

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

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

4.2 The Lexical Decision Model: Productions

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

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

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

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

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

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

[py6]

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

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

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

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

[py7]
```
>>> lex_decision.productionstring(name="retrieving", string="""        1
...     =g>                                                             2
...     isa     goal                                                    3
...     state   retrieving                                              4
...     =visual>                                                        5
...     isa     _visual                                                 6
...     value   =val                                                    7
...     ==>                                                             8
...     =g>                                                             9
...     isa     goal                                                   10
...     state   retrieval_done                                         11
...     +retrieval>                                                    12
...     isa     word                                                   13
...     form    =val                                                   14
... """)                                                               15
{'=g': goal(state= retrieving), '=visual': _visual(cmd= , color= ,     16
 screen_pos= , value= =val)}                                           17
==>                                                                    18
{'=g': goal(state= retrieval_done), '+retrieval': word(form= =val)}    19
```

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

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

[py9]
```
>>> lex_decision.productionstring(name="no lexeme found", string="""
...     =g>
...     isa     goal
...     state   retrieval_done
...     ?retrieval>
...     buffer  empty
...     state   error
...     ==>
...     =g>
...     isa     goal                                                        10
...     state   done                                                        11
...     +manual>                                                            12
...     isa     _manual                                                     13
...     cmd     press_key                                                   14
...     key     F                                                           15
... """)                                                                    16
{'=g': goal(state= retrieval_done),                                         17
 '?retrieval': {'buffer': 'empty', 'state': 'error'}}                       18
==>                                                                         19
{'=g': goal(state= done), '+manual': _manual(cmd= press_key, key= F)}       20
```

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

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

4.3 Running the Lexical Decision Model and Understanding the Output

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

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

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

We are now ready to initialize the simulation:

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

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

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

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

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

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

The simulation can now be run:

[py12]

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

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

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

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

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

4.3.1 Visual Processes in Our Lexical Decision Model

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

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

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

But the two processes proceed independently of each other.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3.2 *Manual Processes in Our Lexical Decision Model*

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

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

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

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

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

4.4 A Left-Corner Parser with Visual and Motor Interfaces

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

As discussed at the end of the previous chapter, left-corner parsers combine top-down and bottom-up features: they can be thought of as predictive top-down parsers with incremental bottom-up filtering.

Left-corner parsing differs from top-down parsing with respect to the amount of evidence necessary to trigger a production rule. A grammar rule cannot be triggered without any evidence from the incoming signal/string of words, as it would be in a top-down parser. But we do not need to accumulate complete evidence, that is, all the necessary words, to trigger a rule, as we would in bottom-up parsing. For example, we do not need both words in *Mary sleeps* to trigger the `S` → `NP VP` rule.

Thus, in left-corner parsing, partial evidence is necessary (in contrast to top-down parsing), and also sufficient (in contrast to bottom-up parsing). For example, having evidence for the very first category on the right-hand side of the rule, namely NP in the sentence *Mary sleeps*, which is described as having evidence for the *left corner* of the `S` → `NP VP` rule, is sufficient to trigger it.

Following Hale (2014, Chap. 3), we summarize the left-corner parsing strategy in the 'project' and 'project and complete' rules below (see (6a) and (6b)). The only difference between them is the context in which the left-corner rule is triggered. If the mother node, e.g., `S` in our example above, is not expected in context, it is added to the context as a 'found' symbol (this is the simple 'project' rule). But if the mother node is already expected in context, we check off that expectation as satisfied. Finally, the 'shift' rule in (6c) takes words one at a time from the incoming string of words and adds them to the top of the stack to be parsed.

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

a. **Project**: if the symbol Y is at the top of the stack, and there is a grammar rule X → Y Z whose right-hand side starts with Y, then replace Y with two new symbols: a record that X has been found and an expectation for the remaining right-hand side symbol(s) Z.

b. **Project and complete**: if the symbol Y is at the top of the stack and right below it is an expectation of finding symbol X, and there is a grammar rule X → Y Z, then replace both Y and X with an expectation for the remaining right-hand side symbol(s) Z.

c. **Shift**: if the next word of the sentence is a terminal symbol of the grammar, push it on the top of the stack.

The distinction between the two different kinds of left-corner projection—projection *tout court* and projection plus a completion step—was proposed in Resnik (1992), who argues that projection and completion is necessary to keep the stack depth reasonably low when parsing both left-branching and right-branching structures.

Most of our rules will be project and complete rules, with the exception of NPs projected by ProperNs. If the ProperN is in subject position, it will trigger a simple projection rule for the NP dominating it since we do not have an NP expectation at that point. But if the ProperN is in object position, the previous application of the `VP -> V NP` rule will have introduced an NP expectation to the context, so we can both project and complete the NP at the same time.

Let's build a left-corner parser in ACT-R. We start by importing `pyactr` and setting the position on the virtual screen where the words in our example—the simple sentence *Mary likes Bill*—will be displayed one at a time.

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

We then declare the chunk types we need:

- `parsing_goal` chunks will be stored in the goal buffer and they will drive the parsing cognitive process;
- `parse_state` chunks will be stored in the imaginal buffer and they will provide intermediate internal snapshots of the parsing process, befitting the kind of information the imaginal buffer stores;
- finally, `word` chunks will be stored in declarative memory and encode lexical information (in our case, just phonological form and syntactic category) for the words in our target example.

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

The `parsing_goal` chunk type in **[py14]** has the same slots as the type we used for the top-down parser discussed in Chap. 3, with the addition of a `right_frontier` slot. The right-frontier slot will be used to record the attachment points for NPs: the `S` node for subject NPs and the `VP` node for object NPs. Specifically, whenever we will store a `parse_state` chunk in the imaginal buffer that will contain information about an NP that has just been parsed, we will take the value in the `right_frontier` slot and record it as the value of the `mother` node for the NP in the imaginal buffer.

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

- the node_cat slot records the syntactic category of the current node, i.e., the node that has just been parsed;
- the mother slot records the mother node of the current node;
- the daughter1 and daughter2 slots record the daughter nodes of the current node;
- finally, the lex_head slot records the lexical head of the current phrasal projection.

The parse_state chunk type gives us a window into how much ACT-R constrains theories of 'high-level' cognitive processes. The goal of the parsing cognitive process can be characterized as incrementally building an unobservable hierarchical tree structure (a structural description) for the target sentence. But there are strict limits on how the partially-built structure is maintained and accessed during the cognitive process: we can only store one chunk at a time in any given buffer (goal, imaginal, retrieval).

This means that the mind never has a global view of the syntactic tree it is constructing. Instead, the structure is viewed through a limited, moving window that can 'see' only a part of the under-construction structure.

Furthermore, the values stored in the slots of a chunk can encode only 'descriptive' content, not specific memory addresses or uniquely identifiable time stamps for specific nodes in the tree. This is particularly constraining for phrases like NPs, which are repeatedly instantiated in a given structure. Their position in the hierarchical structure can be identified only if we encode additional information in their corresponding chunks.

Specifically, we need to record the lexical head associated with an NP to be able to identify which word it dominates, otherwise the NP might end up dominating any ProperN that has already been built/parsed—hence the need for the lex_head slot. We also need to record the point where the full NP is attached in the larger tree, otherwise we might end up attaching the direct object NP to the S node as if it were a subject—hence the need for the mother slot.

These two slots of the parse_state chunk type, namely lex_head and mother, will be exclusively needed for NPs in the left-corner parser introduced in this section. There is no deep reason for this. For simplicity, we only focus on simple mono-clausal target sentences, so only NP and ProperN nodes will be multiply instantiated in any given tree. When we scale up the parser to include multi-clausal sentences and/or multi-sentential discourses, we will end up using these slots for other node types, e.g., VP and S.

We can now initialize the parser model and set up separate variables for the declarative memory module (dm), the goal buffer (g) and the imaginal buffer (imaginal). In [py15], we set a delay of 0 ms for the imaginal buffer, going against its default setting of 200 ms. This default setting is motivated by non-linguistic cognitive processes that are structurally much simpler than language comprehension, and the 200 ms encoding delay provides a better fit to the reaction time data associated with those processes.

In contrast, we believe that a low-delay or even no-delay setting is necessary when modeling language comprehension in ACT-R, because this requires rapidly building complex hierarchical representations that are likely to extensively rely on imaginal chunks. In general, it is reasonable to expect that the systematic modeling of language processing in ACT-R—still very much a nascent endeavor—will occasionally require such departures from received ACT-R wisdom.

[py15]

```
>>> parser = actr.ACTRModel(environment, motor_prepared=True)    1
>>> dm = parser.decmem                                           2
>>> g = parser.goal                                              3
>>> imaginal = parser.set_goal(name="imaginal", delay=0)         4
```

We are ready to add lexical entries to declarative memory. Just as in the case of our top-down parser, we keep the lexical information to a minimum and store only phonological forms and syntactic categories, as shown in **[py16]** below. We also specify the goal chunk needed to get the parsing process started: the initial goal is to read the first word (line 18: the `task` is `read_word`), and to try to parse the whole sentence (line 19: `stack_top` is `S`).

[py16]

```
>>> dm.add(actr.chunkstring(string="""             1
...     isa  word                                  2
...     form Mary                                  3
...     cat  ProperN                               4
... """))                                          5
>>> dm.add(actr.chunkstring(string="""             6
...     isa  word                                  7
...     form Bill                                  8
...     cat  ProperN                               9
... """))                                          10
>>> dm.add(actr.chunkstring(string="""             11
...     isa  word                                  12
...     form likes                                 13
...     cat  V                                     14
... """))                                          15
>>> g.add(actr.chunkstring(string="""              16
...     isa             parsing_goal               17
...     task            read_word                  18
...     stack_top       S                          19
...     right_frontier  S                          20
... """))                                          21
```

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

[py17]

```
>>> parser.productionstring(name="press spacebar", string="""    1
...     =g>                                                      2
...     isa             parsing_goal                             3
...     task            read_word                                4
...     stack_top       ~None                                    5
...     ?manual>                                                 6
...     state           free                                     7
...     ==>                                                      8
...     =g>                                                      9
...     isa             parsing_goal                             10
...      task            encode_word
...      +manual>
...      isa             _manual
...      cmd             'press_key'
...      key             'space'
... """)
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,
 stack_top= ~None, task= read_word), '?manual': {'state': 'free'}}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,
                    stack_top= , task= encode_word),
 '+manual': _manual(cmd= press_key, key= space)}
```

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

[py18]
```
>>> parser.productionstring(name="encode word", string="""
...      =g>
...      isa             parsing_goal
...      task            encode_word
...      =visual>
...      isa             _visual
...      value           =val
...      ==>
...      =g>
...      isa             parsing_goal
...      task            get_word_cat
...      parsed_word    =val
...      ~visual>
... """)
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,
                    stack_top= , task= encode_word),
 '=visual': _visual(cmd= , color= , screen_pos= , value= =val)}
==>
{'=g': parsing_goal(parsed_word= =val, right_frontier= , stack_bottom= ,
 stack_top= , task= get_word_cat), '~visual': None}
```

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

[py19]
```
>>> parser.productionstring(name="retrieve category", string="""
...      =g>
...      isa             parsing_goal
...      task            get_word_cat
...      parsed_word     =w
...      ==>
...      +retrieval>
...      isa             word
...      form            =w
...      =g>
...      isa             parsing_goal
...      task            retrieving_word
... """)
{'=g': parsing_goal(parsed_word= =w, right_frontier= , stack_bottom= ,
                    stack_top= , task= get_word_cat)}
==>
{'+retrieval': word(cat= , form= =w),
 '=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= ,
                    stack_top= , task= retrieving_word)}
```

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

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

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

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

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

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

[py22]
```
>>> parser.productionstring(                                                1
...     name="project and complete: NP ==> ProperN",                        2
...     string="""                                                          3
...         =g>                                                             4
...         isa             parsing_goal                                    5
...         stack_top       ProperN                                         6
...         stack_bottom    NP                                              7
...         right_frontier  =rf                                             8
...         parsed_word     =w                                              9
...         ==>                                                            10
...         =g>                                                            11
...         isa             parsing_goal                                   12
...         task            read_word                                      13
...         stack_top       None                                           14
...         stack_bottom    None                                           15
...         +imaginal>                                                     16
...         isa             parse_state                                    17
...         node_cat        NP                                             18
...         daughter1       ProperN                                        19
...         mother          =rf                                            20
...         lex_head        =w                                             21
... """)                                                                   22
{'=g': parsing_goal(parsed_word= =w, right_frontier= =rf, stack_bottom=    23
                   NP, stack_top= ProperN, task= )}                        24
==>                                                                        25
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None,   26
 stack_top= None, task= read_word), '+imaginal': parse_state(daughter1=    27
ProperN, daughter2= , lex_head= =w, mother= =rf, node_cat= NP)}            28
```

Now that we implemented the NP projection rules, we can turn to the S and VP grammar rules, implemented in [py23] and [py24] below. Both of these rules are project-and-complete rules because, in both cases, we have an expectation for the mother node. We expect an S because that is the default starting goal of all parsing-model runs. And we expect a VP because the `"project and complete: S ==> NP VP"` rule in [py23] always adds a VP expectation to the stack.

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

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

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

[py24]
```
>>> parser.productionstring(                                                    1
...     name="project and complete: VP ==> V NP",                               2
...     string="""                                                              3
...         =g>                                                                 4
...         isa             parsing_goal                                        5
...         task            parsing                                             6
...         stack_top       V                                                   7
...         stack_bottom    VP                                                  8
...         ==>                                                                 9
...         =g>                                                                 10
...         isa             parsing_goal                                        11
...         task            read_word                                           12
...         stack_top       NP                                                  13
...         stack_bottom    None                                                    14
...         +imaginal>                                                              15
...         isa             parse_state                                             16
...         node_cat        VP                                                      17
...         daughter1       V                                                       18
...         daughter2       NP                                                      19
... """)                                                                            20
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= VP,              21
                    stack_top= V, task= parsing)}                                   22
==>                                                                                 23
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None,            24
 stack_top= NP, task= read_word), '+imaginal': parse_state(daughter1= V,            25
 daughter2= NP, lex_head= , mother= , node_cat= VP)}                                26
```

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

[py25]

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

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

[py26]

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

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

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

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

At the very beginning of the simulation (0 ms), the first word of the sentence (*Mary*) is already displayed on the screen (line 13), so the visual module immediately encodes its location (lines 14–15) and retrieves the visual value (word form) at that location soon after that (lines 16–18). At the same time, the `"press spacebar"` rule is selected, which will ultimately trigger a key press that will reveal the second word of the sentence. This rule fires 50 ms later (line 19), initiating a manual process in the motor module.

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

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

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

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

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

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

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

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

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

At the same time, the motor module prepares and initiates the second spacebar press much more quickly: the preparation is instantaneous (line 73) and the initiation takes 50 ms (line 76), so the space key is pressed again at 550 ms. The final word of the sentence (*Bill*) is displayed on the screen (line 89), triggering the same visual-location encoding and visual-value retrieval steps as before.

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

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

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

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

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

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

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

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

We see here that the ACT-R architecture places significant constraints on the construction of deep hierarchical structures like syntactic trees: there is no global view of the constructed tree in declarative memory, only local snapshots recording immediate domination relations, or at the most, two stacked immediate domination relations (when the `mother` slot is specified). As external observers, we can assemble a global view of the syntactic tree based on the `parse_state` chunks stored in declarative memory and their time stamps, but this syntactic tree is never present as such in declarative memory.

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

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

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

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

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

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

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

4.5 Appendix: The Lexical Decision Model

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

File **ch4_lexical_decision.py**:

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

File **ch4_leftcorner_parser.py**:

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

Chapter 5
Brief Introduction to Bayesian Methods and `pymc3` for Linguists

In this chapter, we introduce the basics of Bayesian statistical modeling. Bayesian methods are not specific to ACT-R, or to cognitive modeling. They are a general framework for doing plausible inference based on data—both categorical ('symbolic') and numerical ('subsymbolic') data.

Introducing Bayesian methods at this point in a book about cognitive modeling for linguistics might seem like an unnecessary detour into a complex topic that is only tangentially related to our main goal here. Why would Bayesian methods be a necessary component of building fully formalized and integrated competence-performance/processing theories for generative linguistics in general, and formal semantics in particular?

They are not. However, we believe this detour to be necessary for a different reason. One of the two main goals of this book is to argue for integrated, fully formalized theories of competence and performance: theories that provide full mathematical formalizations that explicitly link the theoretical constructs generative linguists postulate in their analyses and the experimental data generated by widely used psycholinguistic methodologies.

But the second main goal of this book is to enable our readers to build these kinds of theories for themselves and use them in their own work. We want our readers to understand that linguistic theorizing can be more than what it is traditionally taken to be. But equally importantly, we want our readers to be able to *do* more and build more encompassing, fully formalized and experimental-evidence based linguistic theories.

This is why we always provide all the gory details of our Python code throughout the book, although much of that is not immediately relevant to our main, linguistics-centered enterprise: we want our readers to be able to do everything we do, so that they can start using these ideas, frameworks and tools in their own work.

We are now at an important juncture in this book. In the next chapter, we will start introducing the subsymbolic components of ACT-R, which come with a good number of numerical parameters/'knobs'. These parameters need to be dialed in to

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_5

specific settings based on (numerical) experimental data. For simplicity and clarity of exposition, we could choose to pull the correct settings and values out of thin air, and hand-wave in the general direction of statistical inference for the proper way to obtain these specific values. But this will satisfy only our first goal for this book. We would present a formalized theory of competence and performance, but our readers would not be able to reproduce our results, and would ultimately not be able to build better theories and 'tune' their numerical parameters on their own. Similarly, it would be impossible to compare several theories by tuning their parameters and finding out which theory better accounts for the data under consideration.

The Bayesian inference framework introduced in this chapter will enable us to: (i) learn the best settings for numerical parameters from the data, (ii) explicitly quantify our uncertainty about these settings, and (iii) do empirically-driven theory comparison. We will then be able to introduce the subsymbolic components of ACT-R, set the values of the numerical parameters associated with these components based on linguistic data, and numerically compare and evaluate different linguistic theories.

A final word for our readers with a largely non-technical background: it might seem that numbers, equations and code start flooding the pages at this point, and you might feel that the difficulties are insurmountable. They are not. You have to understand that the math we will use in this and the following chapters is actually pretty simple, and it is possible to follow the text by slowing down and googling around a bit. If you find things heavy going, just be enterprising and look online for the right kind of resource for you—the resource that will enable you to bridge the specific gaps you have in your background math knowledge. There are many such resources available now, from blog posts to videos of lectures/tutorials and fully-fledged online courses.

Another thing that you, dear reader, should keep in mind is that you do not have to fully understand everything we show you at this point. Understanding the main ideas and the general process of theory and model development is good enough for a first (or second) pass through the book. That's why we want you to have the full code and be able to run it, modify it, play with it, and grasp how things work by seeing how the results change.

The approach that both of us (i.e., the authors) have gravitated towards more and more as teachers is: your daily activity as a budding (or mature, for that matter) researcher is tinkering within the horizon of a big question. The big question gives you a compass for where the research should be going, and how to change strategies when you get stuck. That's where all the big words ('developing integrated, fully formalized linguistic theories of competence and performance') come in. The tinkering aspect is what makes research fun, doable and hands-on, and keeps it exciting and fundamentally open-minded and exploratory. That's where the code and the instantaneous interaction with the Python interpreter come in.

5.1 The **Python** Libraries We Need

We first load the relevant `Python` libraries:

- `numpy` provides fast numerical and vectorial operations;
- `matplotlib` and `seaborn` provide plotting facilities; we adjust a variety of settings for `matplotlib` in **[py1]**, but the default ones are very good too, and more useful if working interactively in the terminal, in which case you should comment out/ignore lines 4–17;
- `pandas` provides data frames, i.e., data structures well suited for data analysis, basically Excel sheets on steroids; similar to R data frames;
- finally, `pymc3` is the library for Bayesian modeling—Monte Carlo (MC) methods for `Python3`

[py1]

```
>>> import numpy as np                                                         1
                                                                               2
>>> import matplotlib as mpl                                                   3
>>> mpl.use("pgf")                                                             4
>>> pgf_with_pdflatex = {"text.usetex": True, "pgf.texsystem": "pdflatex",    5
...                      "pgf.preamble": [r"\usepackage{mathpazo}",            6
...                                       r"\usepackage[utf8x]{inputenc}",     7
...                                       r"\usepackage[T1]{fontenc}",         8
...                                       r"\usepackage{amsmath}"],            9
...                      "axes.labelsize": 8,                                  10
...                      "font.family": "serif",                               11
...                      "font.serif":["Palatino"],                            12
...                      "font.size": 8,                                       13
...                      "legend.fontsize": 8,                                 14
...                      "xtick.labelsize": 8,                                 15
...                      "ytick.labelsize": 8}                                 16
>>> mpl.rcParams.update(pgf_with_pdflatex)                                     17
>>> import matplotlib.pyplot as plt                                            18
>>> plt.style.use('seaborn')                                                   19
>>> import seaborn as sns                                                      20
>>> sns.set_style({"font.family":"serif", "font.serif":["Palatino"]})         21
                                                                               22
>>> import pandas as pd                                                        23
>>> import pymc3 as pm                                                         24
```

5.2 The Data

We introduce Bayesian estimation methods by using a very simple data set from Brasoveanu and Dotlačil (2015c), which consists of reading times (RTs) associated with two quantifiers, namely *every* and *each*. The data is available in the `pyactr-book` github repo—see the 'data' folder.[1]

We load the file using the `read_csv` method provided by `pandas`—see line 1 in **[py2]**) below.[2] Line 2 in **[py2]** specifies that the `"quant"` (quantifier) variable should be considered categorical.

[1] At this address: https://github.com/abrsvn/pyactr-book/blob/master/data/every_each.csv.

[2] If you are using Windows, you might have to modify the way you provide the path to the csv file. Similarly, you might need to modify the path if the relative path from your current working

We can look at the shape of our data (line 3 in **[py2]**) and we can list the first 3 rows of the data (line 5). We see that the data consists of 347 observations (rows) with respect to two variables (columns): `"logRTresid"` (residualized log-transformed RTs) and `"quant"`. We can also select several different rows/observations by using the `iloc` (integer-based location) method (line 10): we select rows `[0, 8, 18, 31]`, and we display the values in all the columns (`:`).

[py2]

```
>>> every_each = pd.read_csv("./data/every_each.csv")                      1
>>> every_each["quant"] = every_each["quant"].astype('category')           2
>>> every_each.shape                                                       3
(347, 2)                                                                   4
>>> every_each.head(n=3)                                                   5
   logRTresid  quant                                                       6
0    0.056128   each                                                       7
1    0.241384   each                                                       8
2    0.056128  every                                                       9
>>> every_each.iloc[[0, 8, 18, 31], :]                                     10
    logRTresid  quant                                                      11
0     0.056128   each                                                      12
8     0.869077  every                                                      13
18   -0.073706  every                                                      14
31   -0.187536   each                                                      15
```

This data is part of the results of Experiment 2 (a self-paced reading experiment) reported in Brasoveanu and Dotlačil (2015c). The experiment investigates the hypothesis formulated in Tunstall (1998) that the distinct scopal properties of *each* and *every* are, at least to some extent, the consequence of an event-differentiation requirement contributed by *each*. By scopal properties, we mean the preference of these quantifiers to take wide scope over another quantifier in the same sentence.

Consider the examples in (1) and (2) below:

(1) A helper dyed every shirt without thinking about it.

(2) A helper dyed each shirt without thinking about it.

The quantifier phrases *every/each shirt* can take wide or narrow scope relative to the indefinite *a helper* in subject position. On the wide scope reading, the sentences in (1)/(2) are taken to mean that every/each shirt was dyed by a possibly different helper. We also call this reading the *inverse scope* reading because the scope of the quantifiers is the inverse of their surface order. On the narrow scope reading, also known as the *surface scope* reading (for obvious reasons), the sentences in (1)/(2) are taken to mean that the same helper dyed every/each shirt.

On the face of it, both *every* and *each* have the same meaning: they contribute so-called universal quantificational force—as opposed to indefinites like *a* or *some*, which contribute existential quantificational force. However, Tunstall (1998) (see also references therein) notices that *each*, but not *every*, require a separate event for each element it quantifies over.

For example, the sentence *Jake photographed every student in the class, but not separately* is perfectly acceptable, while the minimally different sentence *Jake*

directory to the csv file is different. Please look online for more information about files paths in Python 3; for example, search for "Python 3 file paths on Windows and Linux."

photographed each student in the class, but not separately, where *each* is substituted for *every*, is definitely less acceptable.

Based on contrasts like this, Tunstall (1998, 100) proposes that *each* contributes a *differentiation condition* to the effect that "[e]ach individual object in the restrictor set of the quantified phrase must be associated with its own subevent, in which the predicate applies to that object, and which can be differentiated in some way from the other subevents."

There are many ways in which events can be differentiated from one another, but one way, relevant for our sentences in (1)/(2) above, is for *each* to take inverse scope. In that case, each shirt is dyed by a (possibly) different helper, which ensures that each shirt-dyeing event is differentiated from all others because of the different person doing the dyeing.

Thus, if *each* contributes an event-differentiation requirement, unlike *every*, we expect it to have a higher preference for inverse scope than *every*. And since inverse scope is known to lead to processing difficulties (Kurtzman and MacDonald 1993; Tunstall 1998; Anderson 2004; Pylkkänen and McElree 2006 among many others), which manifest themselves as increased RTs, we expect to see higher RTs for sentence (2) relative to (1).

In their Experiment 2, Brasoveanu and Dotlačil (2015c) test this prediction using a moving-window self-paced reading task (Just et al. 1982). Because the experiment included a separate manipulation, the most important regions of interest (ROIs) were the spillover words immediately following the universal quantifier phrase. Specifically, in examples (1)/(2) above, these ROIs were the words *without*, *thinking* and *about*. The data set we have just loaded in Python and assigned to the `every_each` variable contains measurements collected for the third ROI *about*.

The RTs collected for the ROI *about* were transformed in a couple of ways. Raw reading times in self-paced reading experiments are roughly between 300 and 600 ms per word. These raw reading times were first log-transformed, which yields log RTs roughly between 5 and 7—see lines 1–4 in ([py3]) below. We discuss log transformation/log compression in more detail in the next chapter.

In addition, following Trueswell et al. (1994), Brasoveanu and Dotlačil (2015c) residualized the log RTs by factoring out the influence of word length and word position. This yields residualized log RTs that are roughly between −3 and 3. In fact, in this particular case, they fall just between −1 and 2, as we can see when we inspect the minimum and maximum of the residualized log RTs in our data—see lines 5–8 in [py3].

[py3]
```
>>> np.log(300)                              1
5.703782474656201                            2
>>> np.log(600)                              3
6.396929655216146                            4
>>> np.min(every_each["logRTresid"])         5
-0.678407840683957                           6
>>> np.max(every_each["logRTresid"])         7
1.19278354190761                             8
```

The main question we want to ask of this data set is the following: are the reading times, specifically in the form of residualized log RTs, different for the two quantifiers *every* versus *each*?

That is, we will model RTs as a function of quantifier. One way to model RTs as a function of quantifier is to estimate the two means for the two quantifiers: we can estimate the means and our uncertainty about them, that is, we estimate two full probability distributions, one for each of the means.

But estimating the mean RTs for the two quantifiers will not give us a direct answer to our question: is there a difference in RTs between the two quantifiers? In a Bayesian framework, we could still answer the question given a two-mean model, but it is more straightforward (and closer to the way frequentist estimation would be done) to estimate the difference between the two quantifiers directly.

Thus, in our model, we estimate the mean RT for *every* (together with our uncertainty about it), and instead of estimating the mean RT for *each*, we estimate the mean difference between the *every* RTs and *each* RTs (together with our uncertainty about it). We can still obtain our mean RT for *each* by starting with the mean for *every* and adding to it the mean difference in RTs between the two quantifiers.

If we want to answer our question—are the RTs different for *every* versus *each*?—we basically look at our probability distribution for the difference in RTs and check if 95% of the probability mass is away from 0.[3]

It is important to pause at this point and realize that the reasoning we are going through here has a different structure than the kind of reasoning and arguments linguists are familiar with. From very early on in our linguistic training, we are presented with some data, *we automatically assume there is a pattern in the data*, and we try to identify the pattern and build a theory to capture it. In contrast, our first job as empirically-driven statistical modelers is to ask: is there really a pattern in the data? how sure are we that we're not hallucinating regularities/signal in what is actually pure noise?

This kind of skepticism is actually familiar to linguists in other forms. For example, it is never clear at the outset whether a meaning-related phenomenon (e.g., licensing negative polarity items like *any* or *ever*) should receive a syntactic analysis (Klima 1964), which might seem the 'obvious' way to go, or a semantic one (Ladusaw 1979). As linguists, we know all too well that it is important to be skeptical about the assumptions we make as we build theories.

But it is equally important to be skeptical about the assumptions we make when we identify 'obvious' generalizations and patterns in the data. All data (even introspective data) is ultimately behavioral data, i.e., a product of a performance system, never a direct expression of the unobservable competence system hypothesized to be at the 'core' of the performance system. As such, we need to be reasonably skeptical about all the generalizations and patterns we think we see in the behavior of the system.

[3]This is an oversimplification. See, for example, Kruschke (2011), Nicenboim and Vasishth (2016) and references therein for more discussion of posterior distributions, credible intervals, hypothesis testing and related issues in a Bayesian framework.

Therefore, our question about the quantifier data set is unpacked as follows: (i) can we actually show with enough credibility that the RTs actually differ between the two quantifiers (*every* and *each*)? Assuming we can, (ii) what is the magnitude of the difference[4] and what is our uncertainty about that magnitude? The answer to question (ii) should be of the form: given both our prior knowledge about the phenomenon under discussion and the experimental data, the mean difference between the RTs of the two quantifiers is x_{mean}, and we're 95% certain that the magnitude of the difference is somewhere in the interval $(x_{\text{lowerlimit}}, x_{\text{upperlimit}})$.

Let's now turn to specifying the actual model, and more of this will start making sense. The model we are about to specify is called a t-test, or a linear regression with one binary categorical predictor.

5.3 Prior Beliefs and the Basics of `pymc3`, `matplotlib` and `seaborn`

The way Bayesian estimation works is basically as follows. We start with a prior belief about the quantities of interest, in our case: the RTs for *every*, and the difference in RTs between *each* and *every*. 'Prior' beliefs means that these are our beliefs before we see the data. Furthermore, these beliefs take the form of full probability distributions: we say what values are possible for the quantities of interest, and in addition, we say which of the possible values are plausible (*before* we see the data).

We then update these prior beliefs with the data—specifically, the data stored in the `"logRTresid"` and `"quant"` columns in our data set. Upon exposure to the data, we shift/update our prior beliefs in the direction of the data, and the result of this update consists of two *posterior* probability distributions, one for the mean RT for *every*, and the other for the difference in RTs between the two quantifiers.

Our posterior beliefs/posterior probability distributions are a weighted average of our prior beliefs, on one hand, and the data, on the other hand. If our prior beliefs are very strong (not the case here; see below), the posterior beliefs will reflect the data to a smaller extent. If we have a lot of data, and the data has low variability, the posterior beliefs will reflect the data to a larger, or even overwhelming, extent.

We have very weak prior beliefs about the quantities of interest. Let's characterize them. Before even looking at the specific residualized log RTs associated with *each* and *every*, we know that self-paced reading RTs are usually between 300 and 600 ms, since participants read about 3 words per second. After log-transforming and residualizing these RTs with respect to word length and word position, we get very small values clustered around 0 and spanning the $(-3, 3)$ interval (actually, the $(-2, 2)$ interval) most of the time.

In sum, residualized log RTs for word-by-word self-paced reading are very rarely, if ever, larger than 3 in absolute value. Note that we derived these limits from considerations about word-by-word self-paced reading experiments in general, and the

[4]In residualized log ms, admittedly a non-intuitive temporal unit, which we will omit from now on.

various transformations we apply to the resulting raw RTs (namely, log + residualization).

Therefore, a very reasonable—in the sense of very weakly constraining/very low information—prior for the mean RT for *every* would be a normal (Gaussian) distribution centered at 0 and with a standard deviation (which is a measure of dispersion) of 10. This prior belief effectively says that the mean RT for *every* (after log transformation and residualization, as we already indicated) is, as far as we are concerned and before seeing the data, anywhere between about -30 and 30 (± 3 standard deviations from the mean), most likely somewhere between -20 and 20. Values below -30 or above 30 are possible, but really unlikely.

We call this prior a very low information prior in the sense that it very weakly constrains the range of data we expect to see when we finally look at the experimental data. From prior considerations, we know that this data is very likely in the $(-3, 3)$ interval, and our prior very liberally allows for values in the $(-30, 30)$ interval.

Let's plot a normal distribution with a standard deviation of 10. We'll take this opportunity to introduce the basics of `pymc3` models. In **[py4]** below, we first create a model `every_each_model` using the `pm.Model()` method (line 1).

Then, we start specifying the model components. In our case, we are simply interested in a normally distributed variable called `'normal_density'`. We create this variable with the corresponding `pymc3` probability density function (line 3). When we call this function, we need to specify the name of the variable (needed for `pymc3`-internal purposes), and then provide the parameters for the probability density function: the mean of the normal is set to 0 and the standard deviation is set to 10.

We then ask `pymc3` to sample 5000 draws from this distribution using some sampling magic that we won't worry about,[5] and save the results to a separate database `db` (lines 7–9 in **[py4]**). We specify the type of database that will store the draws to be `Text`. To save and load this type of database, we need to import two additional convenience functions from `pymc3` (lines 5–6 in **[py4]**).

Lines 7–9 in **[py4]** are commented out because we do not want to execute them. Executing them takes a while with the default sampling procedure for `pymc3`. Instead, we load the samples from a previous run of the model, as shown on lines 12–13. The samples we load here are available in the `pyactr-book` github repository, as are the samples (a.k.a. draws, or traces, or posterior estimates) for all the other Bayesian models in this book.[6] But you can also generate your own samples; to do that, simply uncomment lines 7–9 and run them.

[py4]
```
>>> every_each_model = pm.Model()                                        1
>>> with every_each_model:                                               2
...     normal_density = pm.Normal('normal_density', mu=0, sd=10)        3
...                                                                      4
>>> from pymc3.backends import Text                                      5
```

[5]See Lynch (2007); Kruschke (2011); Lambert (2018) among others for detailed and clear discussions.

[6]The traces/posterior samples for all the models in the book are available in the 'data' folder here: https://github.com/abrsvn/pyactr-book/tree/master/data. The samples for the model under discussion are available here: https://github.com/abrsvn/pyactr-book/tree/master/data/normal_trace.

```
>>> from pymc3.backends.text import load                                     6
>>> #with every_each_model:                                                   7
>>>     #db = Text('./data/normal_trace')                                     8
>>>     #trace = pm.sample(draws=5000, trace=db, n_init=500)                  9
                                                                             10
>>> # we load the results / trace of a previous run                          11
>>> with every_each_model:                                                   12
...     trace = load('./data/normal_trace')                                  13
...                                                                          14
>>> def generate_normal_prior_figure():                                      15
...     fig, ax = plt.subplots(ncols=1, nrows=1)                             16
...     fig.set_size_inches(5.5, 3.5)                                        17
...     sns.distplot(trace['normal_density'], hist=True, ax=ax)              18
...     ax.set_xlabel('Normal density, mean = 0, standard deviation = 10')   19
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)                      20
...     plt.savefig('./figures/normal_prior.eps')                            21
...     plt.savefig('./figures/normal_prior.png')                            22
...     plt.savefig('./figures/normal_prior.pdf')                            23
...                                                                          24
>>> generate_normal_prior_figure()                                           25
```

Finally, we turn to plotting the results. We start by defining a plotting function on line 15 in **[py4]**. On line 16, we initialize the figure by indicating that we will have one plot, or 'axis system', which we call ax. This plot will be displayed in a grid consisting of 1 column and 1 row (ncols=1, nrows=1); this specification is superfluous here, but it is useful when we have more than one plot to display. The total figure size is set to 5.5 by 3.5 in. Lines 18–19 plot the histogram of the normal samples and label the plot accordingly. Lines 20–23 tighten up/suitably shrink all the blank margins in the figure and save it in various formats.

As we already mentioned, this entire procedure is assembled under a function generate_normal_prior_figure, which is called on line 25 of **[py4]**. The result is provided in Fig. 5.1. As expected for a normal density centered at 0 and with a standard deviation of 10, most of the probability mass (area under the curve) is spread over the $(-30, 30)$ interval, i.e., within $\pm$ 3 standard deviations of the mean. We will use this normal density as our very weak, noncommittal prior for the mean RT for *every*.

We can now turn to specifying the prior for the difference in RTs between *every* and *each*. This difference could be positive (the mean RT for *each* is greater than the one for *every*), negative (the mean RT for *each* is less than the one for *every*), or 0 (the mean RTs for *each* and *every* are the same). Whether 0, negative or positive, this difference cannot be larger than 6 in absolute value.

To see this, recall that log RT residual values are roughly between -3 and 3 (and are usually very close to 0). In the unlikely event that the mean RT for *each* happens to be 3 and the one for *every* happens to be -3 (or the other way around), we'll get a difference of 6 in absolute value. This is already very unlikely, and the a priori probability of even more extreme values for the difference in RTs is practically 0.

It is therefore reasonable to, once again, specify our prior for the difference in RTs as a normal distribution with a mean of 0 and a standard deviation of 10.

One final remark about the plot in Fig. 5.1. The shaded area under the dark-blue curve is probability mass and the curve itself plots probability *density*. That is, for any point on the x-axis, the height of the curve at that point (i.e., the corresponding value on the y-axis) does *not* plot the probability of that x-axis value: the height of

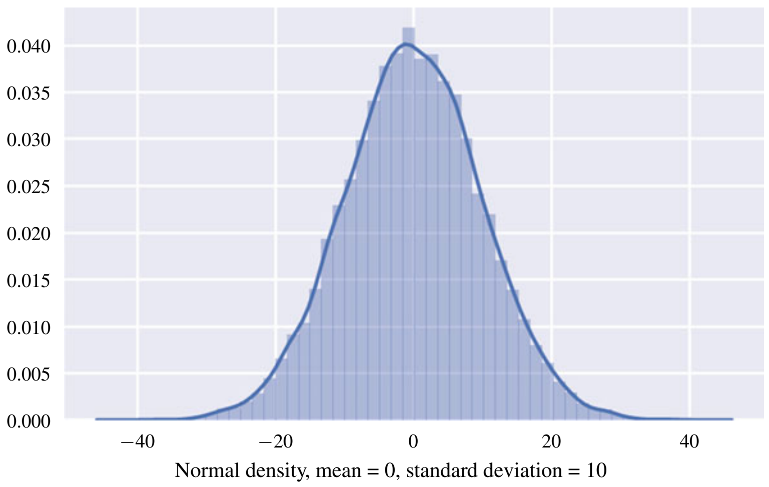


Fig. 5.1 A normal probability density

the curve does *not* provide the probability mass associated with that x-axis value.[7] Since they are density curves, their values can exceed 1 (we will in fact see such a case when we estimate posterior probabilities for our *every/each* model). In contrast, probabilities, i.e., probability masses, can never exceed 1.

5.4 Our Function for Generating the Data (The Likelihood)

Now that we specified our priors, we can go ahead and specify how (we think) nature generated the data. For this purpose, we need to mathematically specify how RT is a function of quantifier.

What we conjecture as our 'data-generating' function (technically, our likelihood function) is that we have two mean RTs for the two quantifiers *every* and *each*. For each of the two quantifiers, the RTs we observe are imperfect reflections of the mean RT for that quantifier, that is, they are somewhere around the mean for that quantifier. More specifically, the observed RTs for a quantifier are composed of the mean RT for

[7]There is 0 probability mass associated with any single point on the x-axis. Just as single points on the real line do not have length, i.e., they have 0 length, single points on the real line have 0 probability associated with them. Only intervals on the real line can have length. Similarly, only intervals can have a non-0 amount of probability mass associated with them.

The height of the curve at an x value does not indicate the probability mass at that value, but the probability density in an infinitesimal interval around that value. That is, we take an infinitesimal interval around the x value, we measure the probability mass sitting on top of that interval and we divide that mass by the length of the interval. In the limit, i.e., as the length of the infinitesimal interval around the x value goes to 0, this ratio, namely $\frac{\text{probability mass}}{\text{interval length}}$, gives us the probability *density* at that point—and this is what is plotted by the dark-blue curves.

that quantifier plus some error, which is caused by our imperfect measurement, the fact that a participant was faster pressing the space bar on one occasion than another etc.

Plotting the RTs by quantifier will make this clearer. On line 1 in [py5] below, we check to see how many observations we have for each quantifier. We see that we have roughly the same number of observations for *every* and *each*—around 170, for a total of 347 observations. We define a function to generate a plot of these 347 observations by quantifier on lines 6–16, and call this function on line 18.

[py5]

```
>>> every_each["quant"].value_counts()                                       1
each      174                                                                2
every     173                                                                3
Name: quant, dtype: int64                                                    4
                                                                             5
>>> def generate_RTs_by_quant_figure():                                      6
...     fig, ax = plt.subplots(ncols=1, nrows=1)                             7
...     fig.set_size_inches(5.5, 3.5)                                        8
...     g = sns.stripplot(x="quant", y="logRTresid", data=every_each,\       9
...                       jitter=True)                                      10
...     g.set_xlabel("Quantifier")                                          11
...     g.set_ylabel("Residualized log RTs")                                12
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)                     13
...     plt.savefig('./figures/RTs_by_quant.eps')                           14
...     plt.savefig('./figures/RTs_by_quant.png')                           15
...     plt.savefig('./figures/RTs_by_quant.pdf')                           16
...                                                                         17
>>> generate_RTs_by_quant_figure()                                          18
```

The plot, provided in Fig. 5.2, shows that the approx. 170 observations collected for *each* are centered somewhere around 0.05, while the approx. 170 observations collected for *every* are centered a little lower, around −0.05. The observations are jittered (`jitter=True` on line 10 in [py5]), that is, they are not plotted on a straight line so that we can distinguish overlapping points in the plot. The observations for *each* are generated from the mean RT for *each* (which is, say, around 0.05) plus some error/noise around that mean RT. The noise is pretty substantial, with observed RTs spread between about −0.75 and 0.75. Similarly, the observations for *every* are generated from the mean RT for *every* (which seems to be around −0.05) plus some error/noise around that mean RT. Once again, the noise is substantial, spreading the observed values mostly between −0.75 and 0.75.

Our job right now is to write this story up in a single formula that will describe how the 347 RTs depend on quantifier. Furthermore, recall that we are interested in the difference between the two quantifiers: we want to estimate it so that we can say whether this difference is likely different from 0, i.e., whether the mean RT for *each* is different from the mean RT for *every*, as Tunstall's differentiation condition would predict. To this end, we will estimate two quantities:

- the mean RT for *every*: RT_{every}
- the mean difference in RT between *each* and *every*: $RT_{each-every}$

With these two quantities in hand, we can obtain the mean RT for *each* in the obvious way: $RT_{each} = RT_{every} + RT_{each-every}$.

We will now use a simple reformulation of the *quant* variable (called 'dummy coding' of the categorical predictor variable `every_each["quant"]`) to be able

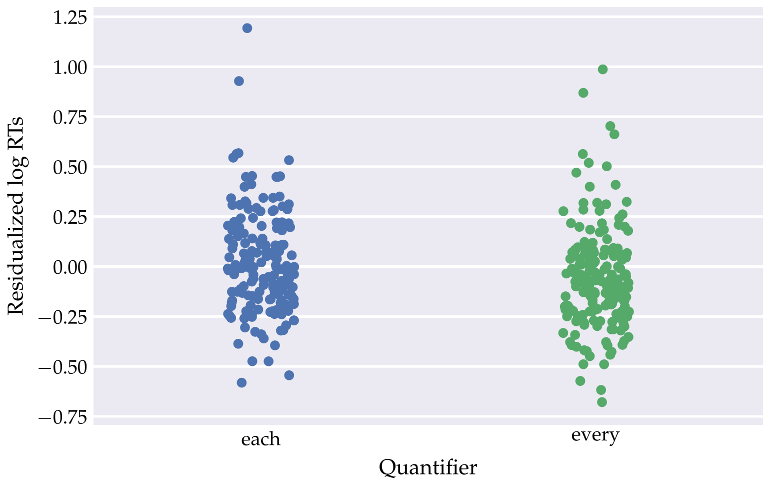


Fig. 5.2 Plot of residualized log RTs by quantifier

to write a *single* formula describing how all 347 RTs are a function of the quantifier they are associated with.

The idea is to rewrite the *quant* variable as taking either a value of 0 or a value of 1, depending on whether the RTs are associated with *every* or *each*. We then multiply this rewritten/dummy-coded *quant* variable with the $RT_{each-every}$ difference as follows:

(3) Formula for RT as a function of quantifier:
$\boxed{RT = RT_{every} + quant \cdot RT_{each-every} + noise}$

a. If *RT* is associated with *every*, our dummy-coding for *quant* says that $quant = 0$. Therefore, the *RT* is generated from the mean RT for *every* plus some noise:
$RT = RT_{every} + 0 \cdot RT_{each-every} + noise = RT_{every} + noise$

b. If *RT* is associated with *each*, our dummy-coding for *quant* says that $quant = 1$. Therefore, the *RT* is generated from the mean RT for *each* plus some noise:
$RT = RT_{every} + 1 \cdot RT_{each-every} + noise = RT_{each} + noise$

The code for the dummy coding of the *quant* variable is a one-liner (line 1 in **[py6]** below). This takes advantage of the vectorial nature of both data and operations in `numpy/pandas`. The resulting `"dummy_quant"` variable, as expected, recodes *each* as 1 and *every* as 0:

[py6]
```
>>> every_each["dummy_quant"] = (every_each["quant"]=="each").      1
astype("int")                                                        2
>>> every_each.head(n=6)                                             3
   logRTresid  quant  dummy_quant                                    4
0    0.056128   each            1                                    5
1    0.241384   each            1                                    6
2    0.056128  every            0                                    7
3    0.037743   each            1                                    8
4   -0.208206  every            0                                    9
5   -0.113990  every            0                                   10
```

We can now use the variable `every_each["dummy_quant"]` and the likelihood function in (3) to generate synthetic datasets. In **[py7]** below, we set our mean RT for *every* to −0.05 and our mean difference in RT to 0.1 (lines 1–2). This will result in a mean RT of 0.05 for *each*. For convenience, we extract the dummy-coded `dummy_quant` variable and store it separately (line 3 in **[py7]**).

We then assemble the means for the 347 synthetic observations we want to generate: line 4 in **[py7]** directly implements the likelihood function in (3). We can look at the first 25 means thus assembled (lines 5–8 in **[py7]** below). For example, the first two are mean RTs associated with *each*, since the first two observations in our original data set are associated with *each* (see lines 5–6 in **[py6]** above). Their mean RT is therefore 0.05. The third observation is associated with *every* since the third observation in our original data set was associated with *every* (see line 7 in **[py6]** above). Its mean RT is therefore −0.05. And so on.

[py7]
```
>>> mean_every = -0.05                                                          1
>>> mean_difference = 0.1                                                       2
>>> quant = np.array(every_each["dummy_quant"])                                 3
>>> synthetic_RT_means = mean_every + quant * mean_difference                   4
>>> synthetic_RT_means[:25]                                                     5
array([ 0.05,  0.05, -0.05,  0.05, -0.05, -0.05,  0.05,  0.05, -0.05,          6
        0.05, -0.05,  0.05,  0.05, -0.05, -0.05,  0.05, -0.05,  0.05,          7
       -0.05, -0.05, -0.05,  0.05,  0.05,  0.05, -0.05])                       8
>>> sigma = 0.25                                                                9
>>> synthetic_RTs = np.random.normal(synthetic_RT_means, sigma)                10
>>> synthetic_RTs.round(2)[:25]                                                11
array([ 0.21,  0.01, -0.05, -0.38,  0.17, -0.31,  0.2 ,  0.02, -0.06,         12
        0.42, -0.41, -0.09, -0.36, -0.57, -0.13, -0.11, -0.27,  0.35,         13
       -0.57,  0.06, -0.08, -0.35,  0.33,  0.02,  0.05])                      14
>>> # compare to the actual RTs in our dataset                                 15
>>> RTs = np.array(every_each["logRTresid"])                                   16
>>> RTs.round(2)[:25]                                                          17
array([ 0.06,  0.24,  0.06,  0.04, -0.21, -0.11, -0.04,  0.02,  0.87,         18
        0.04,  0.09, -0.02,  0.18, -0.49, -0.04,  0.17, -0.28, -0.16,         19
       -0.07, -0.18, -0.13, -0.27,  0.14, -0.34,  0.08])                      20
>>> # repeat to generate a different sample of synthetic RTs                   21
>>> synthetic_RTs = np.random.normal(synthetic_RT_means, sigma)                22
>>> synthetic_RTs.round(2)[:25]                                                23
array([-0.21, -0.01,  0.26,  0.24, -0.15,  0.08,  0.21, -0.53, -0.15,         24
        0.02, -0.38,  0.16, -0.59,  0.02, -0.17, -0.29,  0.07, -0.19,         25
       -0.15, -0.01, -0.4 , -0.03,  0.58, -0.02,  0.25])                      26
```

The likelihood function in (3) has one final component, namely the noise: RTs from a specific quantifier are only imperfect, noisy reflections of the mean RT for that quantifier. The noise comes from variations in the measuring equipment (keyboard etc.), or variations in the way the participants press the space bar at different times, or any other factor that we are not controlling for.

We generate noisy observations by drawing random numbers from a normal distribution: we use the `numpy` function `random.normal` for this purpose (line 10 in

[py7] above). The mean of the normal distribution is the mean RT for one quantifier or the other, and the standard deviation is set to 0.25, which generates noise of about ± 0.75 (lines 9–10 in [py7]). The resulting RTs are randomly generated real numbers (lines 11–14). We can compare them to the actual RTs, extracted and stored in an independent variable `RTs` (lines 16–20). We see that the range of variation in the synthetic data is pretty similar to the actual data.

Finally, if we want to synthesize more RT datasets that are similar to our actual dataset, we can simply do another set of draws from a normal distribution centered at -0.05 or 0.05 (depending on the quantifier) with a standard deviation of 0.25 (lines 22–26 in [py7]).

Now that we understand that the likelihood function has to incorporate some noise, which needs to be estimated from the data, we need to set up a prior for this noise. Reasoning again from our prior knowledge about residualized log RTs, we know that this noise/variability in data cannot really be larger than maybe about 3.

This can be justified as follows. We know that residualized log RTs are between about -3 and 3. Now, if we think of them as being generated from a normal distribution centered somewhere in the interval $(-3, 3)$, a standard deviation (i.e., a noise setting) of about 3 for this normal distribution would very easily cover the interval $(-3, 3)$. This is because a normal distribution spreads 99% of its probability mass within ± 3 standard deviations from its mean.

So, if the normal distribution that generates noisy RTs is centered at 3 and has a standard deviation (noise) of 3, it will spread most of its probability mass between about -6 and 12. This will easily cover the interval $(-3, 3)$. A similar conclusion is reached if we assume that the normal distribution that generates noisy RTs is centered at the other extreme of the interval, namely -3. With a standard deviation of 3, it will spread its probability mass between about -12 and 6, once again easily covering the interval $(-3, 3)$, within which we know from prior considerations that most residualized log RTs live.

Therefore, a very weak and non-committal prior for residualized log RT noise would be a half-normal distribution centered at 0 and with a standard deviation of 10. A half-normal distribution is a normal (Gaussian) distribution centered at 0 and 'folded over' so that all the probability mass over negative values gets transferred to the corresponding positive values. Half-normal distributions correctly require noise/dispersion to be positive.

If we set the standard deviation of this half-normal prior for noise to 10, we place practically no constraints on the actual value of the noise before we see the data: as far as we a priori expect, the noise can be anywhere between 0 and about 30, a very diffuse interval that allows for much larger values than 3, which we already argued would be good enough.

However, since this prior assigns higher probabilities to lower values than to larger values, as we can see in Fig. 5.3, we do expect the noise to be smaller rather than larger. This makes sense: even though values larger than 3 for residualized log RT noise are possible, such values are unlikely and they are more and more unlikely as they get bigger and bigger.

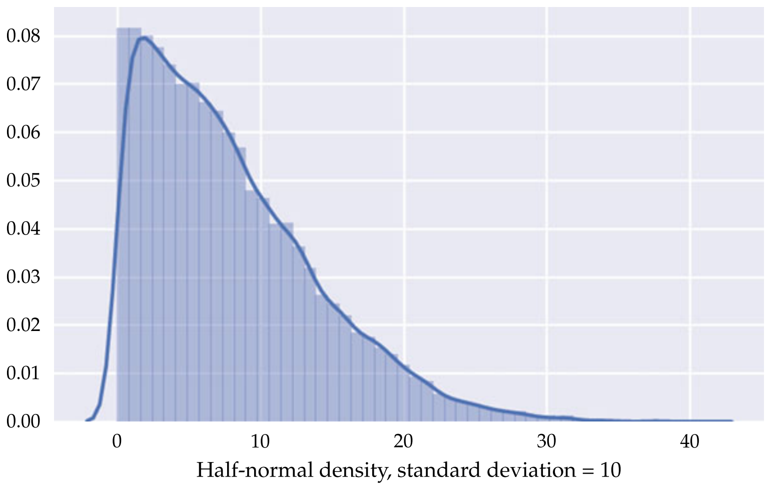


Fig. 5.3 A half-normal probability density

To plot a half-normal prior, we can simulate draws from it in the same way we did for the normal priors for the mean RT for *every* and the mean RT difference between *each* and *every*. We do this in [py8] below, and plot the results in Fig. 5.3.

[py8]
```
>>> every_each_model = pm.Model()
>>> with every_each_model:
...     half_normal_density = pm.HalfNormal('half_normal_density', sd=10)
...
>>> from pymc3.backends import Text
>>> from pymc3.backends.text import load
>>> #with every_each_model:
>>>     #db = Text('./data/half_normal_trace')
>>>     #trace = pm.sample(draws=5000, trace=db, n_init=500)

>>> # we load the results / trace of a previous run
>>> with every_each_model:
...     trace = load('./data/half_normal_trace')
...
>>> def generate_half_normal_prior_figure():
...     fig, ax = plt.subplots(ncols=1, nrows=1)
...     fig.set_size_inches(5.5, 3.5)
...     sns.distplot(trace['half_normal_density'], hist=True, ax=ax)
...     ax.set_xlabel('Half-normal density, standard deviation = 10')
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/half_normal_prior.eps')
...     plt.savefig('./figures/half_normal_prior.png')
...     plt.savefig('./figures/half_normal_prior.pdf')
...
>>> generate_half_normal_prior_figure()
```

5.5 Posterior Beliefs: Estimating the Model Parameters and Answering the Theoretical Question

With the priors and likelihood in hand, we can finally ask pymc3 to give us the posterior distributions for the quantities of interest, namely mean_every and mean_difference. We specify the full model as shown in **[py9]** below:

[py9]
```
>>> every_each_model = pm.Model()                                              1
>>> with every_each_model:                                                     2
...     # priors                                                               3
...     mean_every = pm.Normal('mean_every', mu=0, sd=10)                      4
...     mean_difference = pm.Normal('mean_difference', mu=0, sd=10)            5
...     sigma = pm.HalfNormal('sigma', sd=10)                                  6
...     # likelihood                                                           7
...     observed_RTs = pm.Normal('observed_RTs',                               8
...                        mu=mean_every + quant*mean_difference,              9
...                        sd=sigma,                                          10
...                        observed=RTs)                                      11
...                                                                           12
>>> #with every_each_model:                                                   13
>>>     #db = Text('./data/every_each_model_trace')                           14
>>>     #trace = pm.sample(draws=5000, trace=db, n_init=50000, njobs=4)       15
                                                                              16
>>> with every_each_model:                                                    17
...     trace = load('./data/every_each_model_trace')                         18
...                                                                           19
```

Our assumptions for this model—and for Bayesian models in general—fall into two classes: (i) the assumptions we need to specify the priors, (ii) the assumptions we need to specify the likelihood (how the data is generated). We discuss them in turn.

The priors for our model are specified on lines 4–6 in **[py9]**.

Our prior for the mean RT for *every* is very weak/low information, as already discussed. Recall that in normal and half-normal distributions, 99% of observations fall within 3 standard deviations from the mean, so the prior on line 4 allows for mean (residualized log) RTs anywhere between -30 and 30, an extremely wide range given that log RT residuals are very close to 0 and usually fall within the $(-3, 3)$ interval.

Our prior for the difference in RT between *each* and *every* is similarly weak/low information (line 5): the difference can be positive (higher mean RT for *each* than *every*), negative (lower mean RT for *each* than *every*), or 0 (same mean RT for *each* and *every*). The differences this prior allows for fall anywhere between -30 and 30, again an extremely wide range given that the maximum difference between two residualized log RTs is usually at most 6 in absolute value, and commonly much smaller.

Finally, the prior for our noise distribution on line 6 allows for values anywhere between 0 and 30. This is a very non-committal range since the standard deviation of residualized log RTs is usually around 1, generating a range of residualized log RTs between -3 and 3 if these RTs are centered around 0 and approximately normally distributed.

The likelihood (the data generation part of the model) is specified on lines 8–11 in **[py9]**. We indicate that we model the observed data by explicitly specifying on line 11 that these values are observed, and providing the variable that stores these observed values. In this case, the observed values are stored in the variable `RTs`, which we introduced on line 16 of [py7].

On line 9 of our model specification in [py9], we say that each RT is somewhere near the mean RT for the corresponding quantifier. If the RT is associated with the quantifier *every*, our dummy variable `quant` takes the value 0, so line 9 reduces to `mu=mean_every`. If the RT is associated with the quantifier *each*, the dummy variable `quant` takes the value 1, so line 9 reduces to `mu=mean_every` + `mean_difference`, that is, the mean RT for *each*.

As we already discussed above, each RT is an imperfect, noisy reflection of the mean RT for the corresponding quantifier, so we add some normally distributed noise to that mean RT to obtain the actual RT. This normally distributed noise has a standard deviation `sigma` (line 10). We do not know how large the noise is, so this will be the third parameter we estimate from the data, in addition to our parameters of primary interest `mean_every` and `mean_difference`.

At the end of the day, our assumptions about the priors and the likelihood lead us to using two types of probability distributions: (i) normal distributions, which come with two parameters (mean and standard deviation) that we can tweak to rearrange the way the probability mass is spread over the entire real line; and (ii) half-normal distributions, which are by default assumed to cover the entire positive part of the real line, and which come with only one parameter (standard deviation) that manipulates the spread of probability mass over the positive real numbers.[8]

To summarize, we have three parameters we want to learn about from the data:

i. the mean RT for *every* (`mean_every`),
ii. the mean difference in RT between *each* and *every* (`mean_difference`), and
iii. the magnitude of the noise/dispersion of the actual RTs around the mean RT for the corresponding quantifier (`sigma`).

All these three parameters contribute in essential ways to the likelihood, i.e., to the way we think the observed RT data was generated (lines 8–11 in [py9]). And we need prior distributions for each of these three parameters (lines 4–6 in [py9]), so that we can update the prior distributions with our observed RT data. These prior distributions have a total of five hyperparameters (two means and three standard deviations) that we need to set, and we have set them all to values that are very non-committal. This is why we called them weak, low information priors.

We finally run the model on lines 13–15 in [py9], which means that we ask `pymc3` to compute for us the posterior distributions for the two quantities that are of primary interest to us, namely `mean_every` and `mean_difference`. As before, we do not actually run the model interactively here, but load the samples from a previous run of the model (lines 17–18). And as before, these samples are available in the `pyactr-book` github repo.[9]

[8] We assume that half-normal distributions are always obtained by ‘folding over’ normal distributions centered at 0, so we do not specify a location parameter for them.

[9] See here: https://github.com/abrsvn/pyactr-book/tree/master/data/every_each_model_trace.

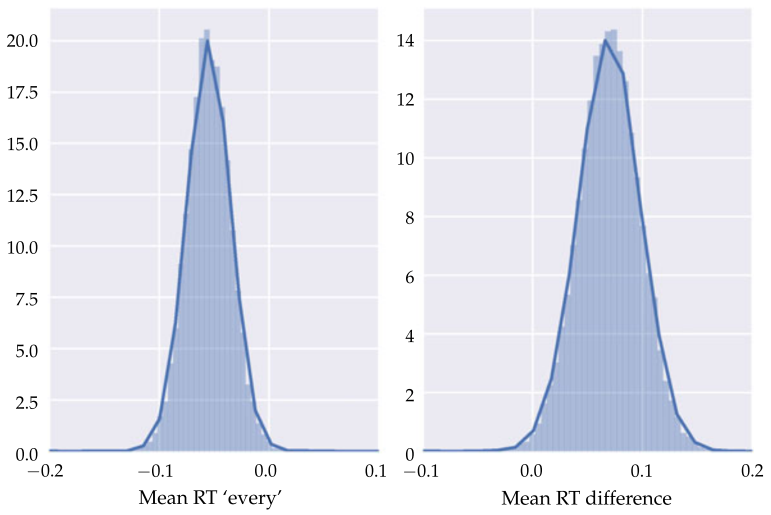


Fig. 5.4 *Every-each* RT model: posterior distributions

We then plot the resulting posterior distributions. The plotting code is provided in [py10] below and the resulting plots are shown in Fig. 5.4. The posterior distributions plotted in Fig. 5.4 are much more constrained than the very 'loose' prior distributions we specified, which indicates that we learned a lot from the data. That is, these posterior distributions mostly reflect the data and not our priors (which were very weak).

[py10]
```
>>> def generate_every_each_model_posteriors_figure():
...     fig, (ax1, ax2) = plt.subplots(ncols=2, nrows=1)
...     fig.set_size_inches(6, 4)
...     plot1 = sns.distplot(trace['mean_every'], hist=True, bins=300,\
...                          ax=ax1)
...     plot1.set_xlim(-0.2, 0.1)
...     ax1.set_xlabel(r"Mean RT 'every'")
...     plot2 = sns.distplot(trace['mean_difference'], hist=True,\
...                          bins=300, ax=ax2)
...     plot2.set_xlim(-0.1, 0.2)
...     ax2.set_xlabel(r'Mean RT difference')
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/every_each_model_posteriors.eps')
...     plt.savefig('./figures/every_each_model_posteriors.png')
...     plt.savefig('./figures/every_each_model_posteriors.pdf')
...
>>> generate_every_each_model_posteriors_figure()
```

Specifically, the posterior mean for the *every* mean RT is around −0.05 and, after seeing the data, we know that this mean RT is roughly between −0.1 and 0. This is much stronger than our prior assumptions, which stated that the mean RT for *every* could be anywhere between −30 and 30.

Similarly, after seeing the data, we know that the RT difference between *each* and *every* is probably around 0.07, and we're very confident this difference is between about 0 and 0.15. That is, we are fairly confident that the mean RT for *each* is larger than the mean RT for *every*. Once again, this is much more constrained than our prior, which countenances differences as small as -30 and as large as 30.

We can ask `pymc3` to examine the posterior distribution of the mean RT difference and tell us precisely where the central 95% portion of the probability mass is[10]:

[py11]
```
>>> mean_difference = trace['mean_difference']
>>> pm.hpd(mean_difference).round(2)
array([0.01, 0.12])
```

The resulting interval is our 95% credible interval: after seeing the data, we are 95% confident that the difference in (residualized log) RTs between *each* and *every* is positive and is somewhere between 0.01 and 0.12.

We can also compute the means and standard deviations (a.k.a. standard errors) of these posterior distributions:

[py12]
```
>>> mean_difference.mean().round(2)
0.07
>>> mean_difference.std().round(2)
0.04

>>> mean_every = trace['mean_every']
>>> mean_every.mean().round(2)
-0.05
>>> mean_every.std().round(2)
0.03

```

We see that these numbers are close to the frequentist ones we would get using a standard function in a standard statistical software environment, e.g., the function `lm()` in R:

(4) Output from R's `lm` function for comparison with our Bayesian estimates:

```
every_each = read.csv("./data/every_each.csv")
every_each$quant = relevel(every_each$quant, ref="every")
print(summary(lm(logRTresid ~ quant, data=every_each)), digits=2)
[...]
                          Estimate   Std. Error   [...]
[mean RT for every]         -0.055        0.019   [...]
[mean RT difference]         0.070        0.028   [...]
[...]
```

The Bayesian and frequentist point estimates are very close, while the standard errors (our uncertainty about these estimates) are smaller, i.e., underestimated, on the frequentist side.

[10]Technically speaking, the `pm.hpd` function returns the 95% highest posterior density (HPD) interval, not the central one. These intervals are very similar for posterior distribution approximations that are basically symmetric and unimodal, like the one under consideration here. For a discussion of the differences between these two types of posterior credible intervals (CRIs), see, for example, Kruschke (2011).

5.6 Conclusion

Bayesian methods for data analysis and cognitive modeling have two advantages, one theoretical and one computational. The theoretical one is that Bayesian methods are very intuitive: they mathematically encode the common-sense idea that we have beliefs about what is plausible and (un)likely to happen in the world, and that we learn from experience, that is, experience/data updates these prior beliefs. To do statistical inference or compute predictions boils down to computing posterior beliefs (i.e., prior beliefs updated with data) and examining these posterior beliefs in various ways.

Computationally, Bayesian methods in general and `pymc3` in particular give us access to a very powerful and flexible way of empirically evaluating linguistic theories. These theories can be faithfully and fairly directly encoded in specific structures for the priors and for the way we think the data is generated (the likelihood). We are not required to take our independently motivated linguistic theories and force them (or parts of them) into a pre-specified statistical inference mold. The opposite actually happens: we take our theory in all its complex and articulated glory and embed it in a fairly direct fashion in a Bayesian model. This, in turn, enables us to empirically evaluate it and do statistical inference about the parameters of the theory in a straightforward way.

Being able to take mathematically specified cognitive models and embed them in Bayesian models to empirically evaluate them will become essential in the next chapter, when we start introducing the subsymbolic components of ACT-R. These subsymbolic components come with a good number of real-valued parameters, and the Bayesian methods introduced in this chapter will enable us to learn the best settings for these parameters from the data, rather than relying on default values that seem to be pulled out of thin air.

Equally importantly, embedding rich cognitive theories in Bayesian models also enables us to do empirically-driven theory comparison. We can take two competing theories for the same phenomenon, collect experimental data, identify the best parameter settings for the two theories, as well as our uncertainty about these parameter settings, and then compare how well the predictions made by these parameter settings fit the data. We will in fact do this at the beginning of the next chapter, where we compare an exponential and a power-law model of forgetting.

5.7 Appendix

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

File **ch5_code.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch5_code.py.

Chapter 6
Modeling Linguistic Performance

The goal of ACT-R is to provide accurate cognitive models of learning and performance, as well as accurate neural mappings of cognitive activities. In this chapter, we introduce the 'subsymbolic' declarative memory components of ACT-R. These are essential to modeling performance, i.e., actual human behavior in experimental tasks. We then build end-to-end models for a variety of psycholinguistic tasks—list recall, lexical decision, (self-paced) reading—and evaluate how closely these models fit the actual data. The models we build are end-to-end in the sense that they include explicit linguistic analyses that are primarily encoded in the production rules (i.e., in procedural memory), together with a realistic model of declarative memory and simple, but reasonably realistic, vision and motor modules.

When studying performance, we are usually interested in two measures: (i) what response people choose given some stimulus, and (ii) how much time it takes them to make that choice, known as reaction time. In linguistics, the first measure often appears as the "Accept–Reject" response when people judge the grammatical or interpretational status of a sentence or a discourse. But other types of experimental tasks also fit here, for example, responses in forced-choice tasks, lexical decision tasks, acceptability judgment tasks etc. The second measure often encodes how much time it takes to choose a particular response, but other options also exist, e.g., how much time it takes to shift eye gaze, to move a mouse, to press the spacebar to reveal the next word etc. This chapter is dedicated to introducing the components of ACT-R that enable us to make realistic predictions with respect to both kinds of measures.

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_6

6.1 The Power Law of Forgetting

The main idea behind the ACT-R declarative memory architecture is that

> human memory is behaving optimally with respect to the pattern of past information presentation. Each item in memory has had some history of past use. For instance, our memory for one person's name may not have been used in the past month but might have been used five times in the month previous to that. What is the probability that the memory will be needed (used) during the conceived current day? Memory would be behaving optimally if it made this memory less available than memories that were more likely to be used but made it more available than less likely memories. (Anderson and Schooler 1991, 396)

In particular, the availability of a specific chunk stored in declarative memory, i.e., its *activation*, which determines both the probability that it will be successfully retrieved and its retrieval time/latency, is a function of the past use of that memory chunk (among other things; more about other factors later).

To see how this is actually formalized in ACT-R, let's examine the well-known Ebbinghaus (1913) retention data, presented in his Chap. 7 and shown here in **[py13]** below. The stimulus materials used by Ebbinghaus consisted of nonsense CVC syllables, about 2300 in number. They were mixed together and then, syllables were randomly selected to construct lists of different lengths that needed to be memorized.

The method used to memorize them was 'learning to criterion': Ebbinghaus repeated the list as many times as necessary to reach a prespecified level of accuracy (e.g., one perfect reproduction of the list). The retention measure was 'percent savings', which was computed as follows. First, a list was learned to criterion, and Ebbinghaus counted the number of times he needed to repeat the list until that happened. Let's say he needed to repeat the list 10 times until he could reproduce it perfectly. He waited one day, and then he tried to see how much he still remembered. The way Ebbinghaus chose to measure 'how much' is by seeing how many times he needed to repeat the list on the second day until he learned it to criterion again. Let's say that he needed to repeat the list only 7 times on the second day. So, he saved 3 list repetitions compared to the first day, that is, he had 30% savings ($\frac{10-7}{10}$) after one day.

[py13]
```
>>> # loading the data                                                       1
>>> import pandas as pd                                                      2
>>> ebbinghaus_data = pd.read_csv('./data/ebbinghaus_retention_data.csv')    3
>>> ebbinghaus_data                                                          4
   delay_in_hours  percent_savings                                           5
0            0.33             58.2                                           6
1            1.00             44.2                                           7
2            8.80             35.8                                           8
3           24.00             33.7                                           9
4           48.00             27.8                                          10
5          144.00             25.4                                          11
6          744.00             21.1                                          12
```

In **[py13]**, we load the Ebbinghaus data from a `csv` file using the `pandas` library (lines 2–3). The data is displayed on lines 5–12. We see that there are 7 data points/observations (7 rows, numbered 0 through 6). The first column in the data is the independent variable (time): it records the delay in hours that the relearning of the syllable series took place relative to the initial learning-to-criterion time. The second column is the dependent variable (the measure of activation in memory): it records the percent savings observed for the corresponding delay in hours, i.e., the reduction in repetitions of the target series of syllables needed to relearn it to criterion.

For completeness, we provide the summary of the Ebbinghaus data in [py14] below. This gives us the total number of observations (7), the means for the delay in hours and savings percentages (138.59 and 35.17 respectively), the standard deviation etc.

[py14]
```
>>> ebbinghaus_data.describe()
       delay_in_hours  percent_savings
count         7.00000         7.000000
mean        138.59000        35.171429
std         271.65549        12.663690
min           0.33000        21.100000
25%           4.90000        26.600000
50%          24.00000        33.700000
75%          96.00000        40.000000
max         744.00000        58.200000
```

A much better way to develop an intuitive understanding of this data is to plot it. In [py15] below, we load the visualization (plotting) libraries `matplotlib` and `seaborn`, and specify a variety of options for them (lines 2–20). If working in the terminal, you should probably simply load these libraries and accept their default settings. We can now define a function to plot the data, and then call it (lines 22–57 in [py15]). The resulting 3 plots are provided in Fig. 6.1.

[py15]
```
>>> # settings for data visualization                                            1
>>> import matplotlib as mpl                                                     2
>>> mpl.use("pgf")                                                               3
>>> pgf_with_pdflatex = {"text.usetex": True, "pgf.texsystem": "pdflatex",       4
...                      "pgf.preamble": [r"\usepackage{mathpazo}",              5
...                                       r"\usepackage[utf8x]{inputenc}",       6
...                                       r"\usepackage[T1]{fontenc}",           7
...                                       r"\usepackage{amsmath}"],              8
...                      "axes.labelsize": 8,                                    9
...                      "font.family": "serif",                                 10
...                      "font.serif":["Palatino"],                              11
...                      "font.size": 8,                                         12
...                      "legend.fontsize": 8,                                   13
...                      "xtick.labelsize": 8,                                   14
...                      "ytick.labelsize": 8}                                   15
>>> mpl.rcParams.update(pgf_with_pdflatex)                                       16
>>> import matplotlib.pyplot as plt                                              17
>>> plt.style.use(’seaborn’)                                                     18
>>> import seaborn as sns                                                        19
>>> sns.set_style({"font.family":"serif", "font.serif":["Palatino"]})            20
                                                                                 21
>>> def generate_ebbinghaus_data_figure():                                       22
...     fig, (ax1, ax2, ax3) = plt.subplots(ncols=1, nrows=3)                    23
...     fig.set_size_inches(5.5, 6.3)                                            24
...     # plot 1                                                                 25
...     ax1.plot(ebbinghaus_data[’delay_in_hours’],                              26
...             ebbinghaus_data[’percent_savings’],                              27
...             marker=’o’, linestyle=’--’)                                      28
...     ax1.set_title(’a. Non-transformed data’)                                 29
...     ax1.set_xlabel(’Delay (hours)’)                                          30
...     ax1.set_ylabel(’Savings (\\%)’)                                          31
...     # plot 2                                                                 32
...     ax2.plot(ebbinghaus_data[’delay_in_hours’],                              33
...             ebbinghaus_data[’percent_savings’],                              34
...             marker=’o’, linestyle=’--’)                                      35
...     ax2.set_title(’b. Log performance (log percent savings), base 10’)       36
...     ax2.set_xlabel(’Delay (hours)’)                                          37
...     ax2.set_ylabel(’Savings (log \\%)’)                                      38
```

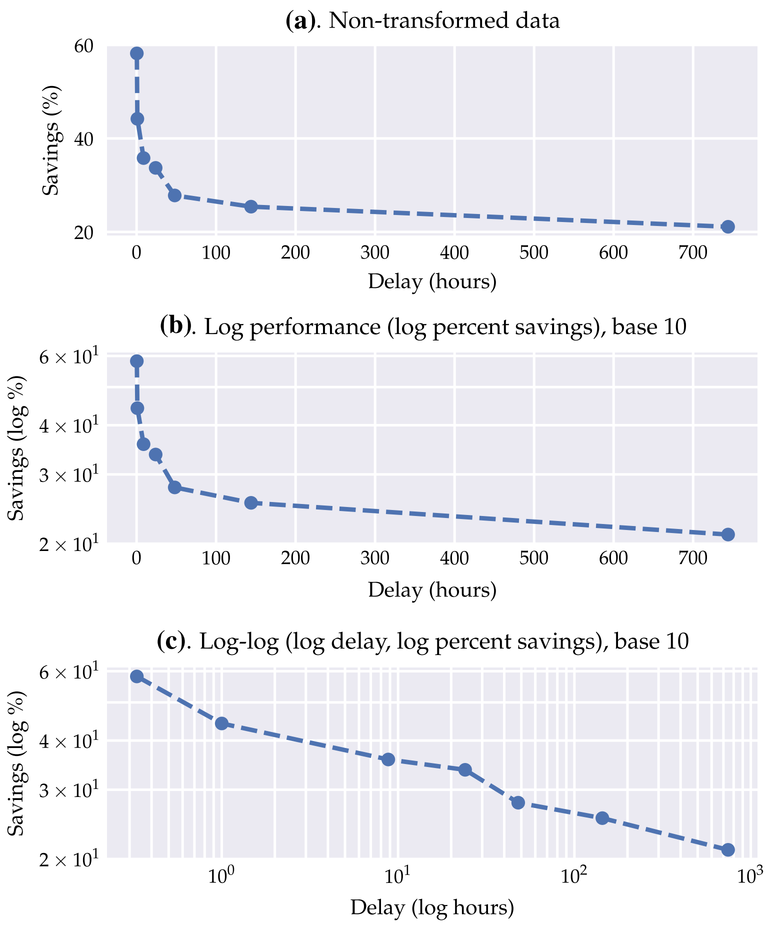


Fig. 6.1 Ebbinghaus retention data

```
...     ax2.set_yscale('log', basey=10)
...     ax2.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # plot 3
...     ax3.plot(ebbinghaus_data['delay_in_hours'],
...             ebbinghaus_data['percent_savings'],
...             marker='o', linestyle='--')
...     ax3.set_title('c. Log-log (log delay, log percent savings), base 10')
...     ax3.set_xlabel('Delay (log hours)')
...     ax3.set_xscale('log', basex=10)
...     ax3.set_ylabel('Savings (log \\%)')
...     ax3.set_yscale('log', basey=10)
...     ax3.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # clean up and save
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/ebbinghaus_data.eps')
...     plt.savefig('./figures/ebbinghaus_data.png')
...     plt.savefig('./figures/ebbinghaus_data.pdf')
...
>>> generate_ebbinghaus_data_figure()
```

The three panels in Fig. 6.1 plot the retention data in its non-transformed form (panel a), with a logarithmically compressed y axis (panel b: we plot log percent savings), and with both axes logarithmically compressed (panel c). We see that a linear relation emerges in the final log-log plot, which indicates that forgetting (decay of chunk activation in declarative memory) has a particular functional form—to which we now turn. (The log tick marks are in base 10 for readability, although we always work with the natural logarithm, i.e., log base e, in what follows.)

The forgetting curve in plot (a) of Fig. 6.1 is sometimes taken to reflect an underlying negative exponential forgetting function of the form:

(1) $P = \alpha e^{-\beta T}$, where:

- P is the memory-related performance measure (percent savings in the Ebbinghaus data),
- T is the time delay since presentation (since initial learning to criterion in our case), and
- α, β are the free parameters of the model, to be fit to the data.

But this predicts that performance should be a linear function of time if we log-transform the performance P:

(2) $\log(P) = \log(\alpha) - \beta T$, i.e., a linear function of time T with intercept $\log(\alpha)$ and negative slope $-\beta$

One way to intuitively think about logarithmic transformation/logarithmic compression is to think about a series of evenly spaced trees that you can see on the side of a long straight road as you look up the road. The distances between the trees appear smaller and smaller as the trees are further and further away, until the trees basically become one tree as the gaze approaches the horizon. The further away two trees are from us, the smaller the distance between them seems to be.

Similarly, the larger two numbers are, the more the difference between them is compressed: the difference between 4 and 2 is compressed much less than the difference between 14 and 12 under the log transform. Equivalently, the larger a number is, the higher its compression under the log transform. This is shown on lines 3–6 in **[py16]** below, as well as in the plot of the log transform (for $x \geq 1$) in Fig. 6.2.

[py16]

```
>>> import numpy as np                                      1
                                                            2
>>> np.log(4) - np.log(2)                                   3
0.6931471805599453                                          4
>>> np.log(14) - np.log(12)                                 5
0.15415067982725805                                         6
                                                            7
>>> def generate_log_figure():                              8
...     fig, ax = plt.subplots(ncols=1, nrows=1)            9
...     fig.set_size_inches(5.5, 3)                        10
...     x = np.arange(1, 15, 0.01)                         11
...     ax.plot(x, np.log(x), linestyle='-')               12
...     #ax.set_xlim(left=1)                               13
...     ax.set_xlabel(r'$x$')                              14
...     ax.set_ylabel(r'$\log(x)$')                        15
```

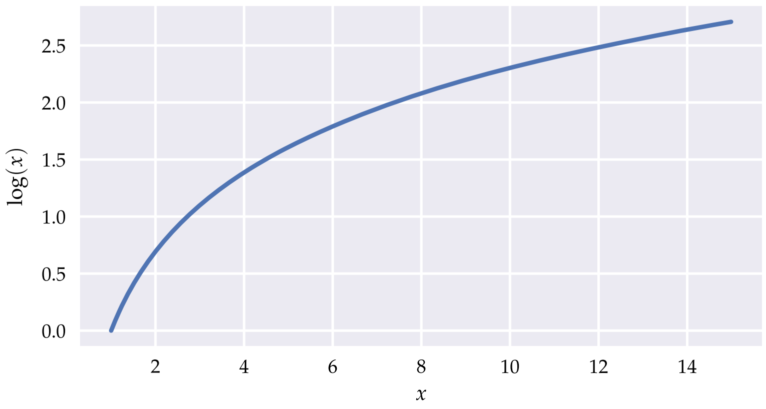


Fig. 6.2 Plot of the log transform (for $x \geq 1$)

```
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/log_plot.eps')
...     plt.savefig('./figures/log_plot.png')
...     plt.savefig('./figures/log_plot.pdf')
...
>>> generate_log_figure()
```

Throughout this book, when we use logarithm *simpliciter* without explicitly specifying the base, we always mean the natural logarithm function, i.e., log base e. We will therefore abbreviate the natural logarithm of a number x simply as $\log(x)$ or, dropping parentheses, $\log x$ (rather than $\ln x$ or $\log_e x$).

The idea behind the exponential model of forgetting is that, once we logarithmically compress performance, log performance will be a linear function of time. However, panel (b) of Fig. 6.1 shows that this is not the case: the relationship between the delay on the x-axis, measured in hours, and savings on the y-axis, measured in log-transformed percentages, is still not linear.

We can use our recently acquired knowledge of Bayesian modeling with pymc3 to compare the actual observations and the predictions made by a theory that hypothesizes that performance (forgetting) is an exponential function of time.

In **[py17]**, we import the relevant libraries and then store the delay and savings data in separate variables for convenience (lines 1–6). We then write up the exponential model directly from the equation in (2): the likelihood function defined on lines 15–17 says that log savings (i.e., log performance) is a linear function of delay (with two free parameters intercept and slope), plus some normally distributed noise with standard deviation sigma.

The hypothesis that log savings are a linear function of delay is tantamount to saying that, if we plot the mean mu of log savings for any given delay, we obtain a line. A line is standardly characterized in terms of an intercept and a slope (line 15 in **[py17]**): mu is a deterministic function of delay, given parameters intercept and slope. The intercept corresponds to $\log(\alpha)$ in formula (2) above, and the slope corresponds to $-\beta$.

Lines 11–13 in **[py17]** provide low information priors for the intercept, slope, and noise. The priors have forms familiar from the previous chapter. We set the standard deviations for all priors to 100, which is very non-committal since the response/dependent variable is measured in log-percent units.

Once the priors and likelihood are specified, we can run the model. We save the result, i.e., our posterior estimates for the parameters `intercept`, `slope` and `sigma`, in the variable `trace`.[1]

[py17]
```
 1  >>> import pymc3 as pm
 2  >>> from pymc3.backends import Text
 3  >>> from pymc3.backends.text import load
 4
 5  >>> delay = ebbinghaus_data['delay_in_hours']
 6  >>> savings = ebbinghaus_data['percent_savings']
 7
 8  >>> exponential_model = pm.Model()
 9  >>> with exponential_model:
10  ...     # priors
11  ...     intercept = pm.Normal('intercept', mu=0, sd=100)
12  ...     slope = pm.Normal('slope', mu=0, sd=100)
13  ...     sigma = pm.HalfNormal('sigma', sd=100)
14  ...     # likelihood
15  ...     mu = pm.Deterministic('mu', intercept + slope*delay)
16  ...     log_savings = pm.Normal('log_savings', mu=mu, sd=sigma,
17  ...                             observed=np.log(savings))
18  ...
19  >>> #with exponential_model:
20  >>>     #db = Text('./data/exponential_model_trace')
21  >>>     #trace = pm.sample(draws=5000, trace=db, n_init=50000, njobs=4)
22
23  >>> with exponential_model:
24  ...     trace = load('./data/exponential_model_trace')
25  ...
```

With the posterior distributions for our exponential model in hand, we can compare the predictions made by the model against the actual data to see how close the predictions are. The predictions are stored in the variable `mu`. These are predicted log savings. If we exponentiate them, we obtain predicted savings.

Furthermore, if we look at the 95% credible intervals for the predicted savings, we can see the range of predictive variability/uncertainty in the predictions made by the exponential model. If the actual savings fall within these credible intervals, we can take the model to be empirically adequate. The code in **[py18]** below generates two plots that enable us to empirically evaluate the exponential model. The two plots are provided in Fig. 6.3.

[py18]
```
>>> mu = trace["mu"]

>>> def generate_ebbinghaus_data_figure_2():
...     fig, (ax1, ax2) = plt.subplots(ncols=1, nrows=2)
...     fig.set_size_inches(5.5, 4.5)
...     # plot 1
...     ax1.plot(delay, savings, marker='o', linestyle='--')
...     ax1.plot(delay, np.median(np.exp(mu), axis=0), color='red', linestyle='-')
...     ax1.set_title('b. Log performance (blue) and exponential model estimates (red)')
```

[1]Recall that the traces, i.e., posterior estimates, for all the models in the book are available in the 'data' folder of the `pyactr-book` github repo. For example, the trace for the exponential model under consideration is available here: https://github.com/abrsvn/pyactr-book/tree/master/data/exponential_model_trace.

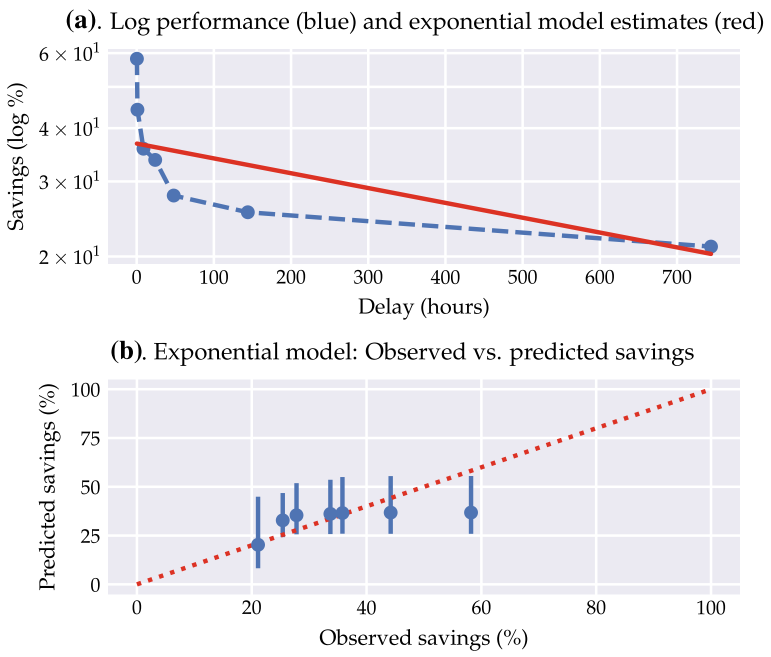


Fig. 6.3 Ebbinghaus retention data and the exponential forgetting model

```
...     ax1.set_xlabel('Delay (hours)')
...     ax1.set_ylabel('Savings (log \\%)')
...     ax1.set_yscale('log', basey=10)
...     ax1.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # plot 2
...     yerr = [np.median(np.exp(mu), axis=0) - pm.hpd(np.exp(mu))[:, 0],
...             pm.hpd(np.exp(mu))[:, 1] - np.median(np.exp(mu), axis=0)]
...     ax2.errorbar(savings, np.median(np.exp(mu), axis=0), yerr=yerr,
...                  marker='o', linestyle='')
...     ax2.plot(np.linspace(0, 100, 10), np.linspace(0, 100, 10),
...              color='red', linestyle=':')
...     ax2.set_title('Exponential model: Observed vs. predicted savings')
...     ax2.set_xlabel('Observed savings (\\%)')
...     ax2.set_ylabel('Predicted savings (\\%)')
...     ax2.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # clean up and save
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/ebbinghaus_data_2.eps')
...     plt.savefig('./figures/ebbinghaus_data_2.png')
...     plt.savefig('./figures/ebbinghaus_data_2.pdf')
...
>>> generate_ebbinghaus_data_figure_2()
```

The plot in the top panel of Fig. 6.3 reproduces the middle panel of Fig. 6.1, together with the line of best fit predicted by the exponential model. It is clear that the line does not match the actual data very well.

This lack of empirical adequacy is also visible in the second plot of Fig. 6.3, which plots the percent savings predicted by the exponential model on the y-axis against the observed percent savings on the x-axis. The red diagonal line indicates the points where the predictions would be exactly equal to the observed values.

We see that the median predicted savings are not very close to the observed values, especially for higher savings (associated with a short delay). Some of the points are pretty far from the diagonal line, and some of the 95% intervals do not cross the diagonal line at all, or barely cross it.

The fact that the points are pretty far from the diagonal line indicates that the exponential model makes incorrect predictions. The fact that some of the 95% intervals around those median predictions do not cross the diagonal line, or barely cross it, indicates that the exponential model is not only wrong, but it is also pretty confident about some of its incorrect predictions.

We can now fairly confidently conclude that memory performance (forgetting) is not a negative exponential function of time. Instead, plot (c) in Fig. 6.1 shows that performance is a *power function* of time. That is, performance is a linear function of time only if both performance and time are log-transformed:

(3) $\log(P) = \log(\alpha) - \beta\log(T)$

(4) **The power law of forgetting**: $\boxed{P = \alpha T^{-\beta}}$
(final form of the forgetting function, obtained by exponentiating both sides of (3))

A line fits the log-log (log savings-log delay) data very well. Once again, we can set up a Bayesian model that directly implements the formula in (3), and then examine its predictions. The code for the power law model is provided in **[py19]**, and the code generating two plots parallel to the plots we generated for the exponential model is provided in **[py20]**. The resulting plots are provided in Fig. 6.4.

[py19]
```
>>> power_law_model = pm.Model()
>>> with power_law_model:
...     # priors
...     intercept = pm.Normal('intercept', mu=0, sd=100)
...     slope = pm.Normal('slope', mu=0, sd=100)
...     sigma = pm.HalfNormal('sigma', sd=100)
...     # likelihood
...     mu = pm.Deterministic('mu', intercept + slope*np.log(delay))
...     log_savings = pm.Normal('log_savings', mu=mu, sd=sigma,
...                             observed=np.log(savings))
...
>>> #with power_law_model:
>>>     #db = Text('./data/power_law_model_trace')
>>>     #trace = pm.sample(draws=5000, trace=db, n_init=50000, njobs=4)

>>> with power_law_model:
...     trace = load('./data/power_law_model_trace')
...
```

[py20]
```
>>> mu = trace["mu"]
>>> def generate_ebbinghaus_data_figure_3():
...     fig, (ax1, ax2) = plt.subplots(ncols=1, nrows=2)
...     fig.set_size_inches(5.5, 4.5)
...     # plot 1
...     ax1.plot(delay, savings, marker='o', linestyle='--')
...     ax1.plot(delay, np.median(np.exp(mu), axis=0), color='red', linestyle='-')
...     ax1.set_title('c. Log-log plot (blue) and power law model estimates (red)')
...     ax1.set_xlabel('Delay (log hours)')
...     ax1.set_xscale('log', basex=10)
...     ax1.set_ylabel('Savings (log \\%)')
...     ax1.set_yscale('log', basey=10)
...     ax1.grid(b=True, which='minor', color='w', linewidth=1.0)
```

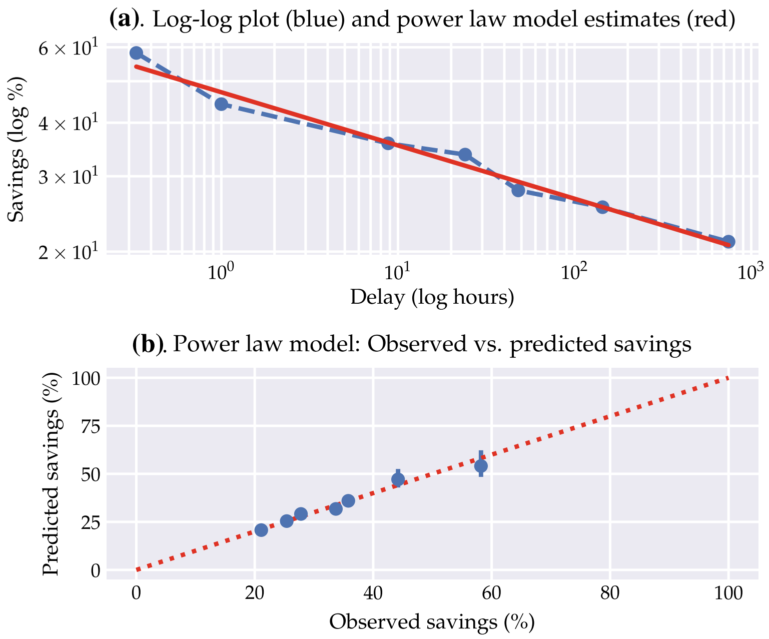


Fig. 6.4 Ebbinghaus retention data and a power forgetting function

```
...     # plot 2
...     yerr = [np.median(np.exp(mu), axis=0) - pm.hpd(np.exp(mu))[:, 0],
...             pm.hpd(np.exp(mu))[:, 1] - np.median(np.exp(mu), axis=0)]
...     ax2.errorbar(savings, np.median(np.exp(mu), axis=0), yerr=yerr,
...                  marker='o', linestyle='')
...     ax2.plot(np.linspace(0, 100, 10), np.linspace(0, 100, 10),
...              color='red', linestyle=':')
...     ax2.set_title('Power law model: Observed vs. predicted savings')
...     ax2.set_xlabel('Observed savings (\\%)')
...     ax2.set_ylabel('Predicted savings (\\%)')
...     ax2.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # clean up and save
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/ebbinghaus_data_3.eps')
...     plt.savefig('./figures/ebbinghaus_data_3.png')
...     plt.savefig('./figures/ebbinghaus_data_3.pdf')
...
>>> generate_ebbinghaus_data_figure_3()
```

The top plot in Fig. 6.4 reproduces the log-log plot in the third panel of Fig. 6.1, together with the line of best fit predicted by the power law model of forgetting. We see that the model predictions match the data very well.

This is further confirmed by the second plot in Fig. 6.4. The points are almost perfectly aligned with the diagonal. That is, the savings predicted by the power law model are very close to the observed savings.

Furthermore, the confidence intervals around most predictions are so tight that that we do not see any segments extending outward from the plotted points. That is, the power law model makes correct predictions, and it is furthermore highly confident about the predictions it makes (with somewhat less confidence for higher savings).

We conclude that the better model for the Ebbinghaus forgetting data is a power law model, and not an exponential one.

6.2 The Base Activation Equation

The ACT-R base activation equation in (5) directly reflects the power law of forgetting:

(5) ACT-R base activation: $\boxed{B_i = \log\left(\sum_{k=1}^{n} t_k^{-d}\right)}$.

Equivalently (exponentiating both sides): $e^{B_i} = \sum_{k=1}^{n} t_k^{-d} = t_1^{-d} + t_2^{-d} + \cdots + t_n^{-d}$

The base activation B_i of a chunk i in declarative memory is a log-transformed measure of performance. So the actual measure of performance is e^{B_i}, and e^{B_i} is a power function of the times t_k since the chunk was presented. 'Presentation' in ACT-R really means two things: (i) the chunk was created for the first time, for example, because the human (or the model) was confronted with a new fact, or (ii) the chunk was re-created. Re-creating most often happens when the human (or the model) correctly recalls a chunk, after which the chunk is stored again in memory.

Memory-related performance e^{B_i} on a specific chunk i at a specific time of retrieval from memory t_{now} is the sum of t_k^{-d}, for all k presentations of the chunk, where k varies from 1 to the total number of chunk presentations n. For each presentation k, t_k is the period of time elapsed between the time of presentation k and the time of retrieval t_{now}. That is, t_k is the same as the *delay* variable in the Ebbinghaus data. The negative exponent $-d$ (decay) is the equivalent of the $-\beta$ slope parameter in our log-log (power-law) model of the Ebbinghaus data.

The basic intuition behind the base activation equation in (5) is that

> at any point in time, memories vary in how likely they are to be needed and the memory system tries to make available those memories that are most likely to be useful. The memory system can use the past history of use of a memory to estimate whether the memory is likely to be needed now. This view sees human memory in some sense as making a statistical inference. However, it does not imply that memory is explicitly engaged in statistical computations. Rather, the claim is that whatever memory is doing parallels a correct statistical inference. (Anderson and Schooler 1991, 400)

What memory is inferring is *activation*, which reflects "need probability": the probability that we will need a particular chunk now. The basic assumption, already developed in Anderson (1990), is that chunks (facts in the declarative memory) are considered in order of their need probabilities until the need probability of a chunk is so low that it is not worth trying to retrieve that chunk anymore—i.e., the chunk's activation is below a retrieval threshold.

This description of what declarative memory does when it retrieves chunks is serial, but the actual retrieval process is formalized as a parallel process in which chunks are simultaneously accessed, and the one with the highest activation is retrieved (if the activation exceeds the threshold).

Crucially, this theory of declarative memory derives specific predictions about the relationship between activation, which is an unobserved quantity reflecting need probability, and observable/measurable quantities: recall latency (how long retrieving a fact takes) and recall accuracy (what is the probability of a successful retrieval).

The key to understanding the connection between activation on one hand, and recall latency and accuracy on the other hand, is to understand the specific way in which activation reflects need probability. The statement in (6) below is left rather implicit in Anderson (1990) and Anderson and Schooler (1991):

(6) Activation B_i is the logit (log odds) transformation of need probability: $B_i = \log(o_i)$.
Thus, exponentiated activation e^{B_i}, which is the actual measure of performance, is the need odds o_i of chunk i:[2] $\boxed{e^{B_i} \text{ is the odds that chunk } i \text{ is needed}}$.

Summarizing, the base-level activation equation in (5) says that exponentiated activation e^{B_i}, which encodes the 'need odds' of chunk i (the odds that chunk i is needed at the time of retrieval t_{now}), is a power function of time. This power function has two components, $\sum\limits_{k=1}^{n}$ and t_k^{-d}, which formalize the following:

$\sum\limits_{k=1}^{n}$: individual presentations 1 through n of a chunk i have a strengthening impact on the need odds of chunk i; a presentation k additively increases the previous need odds for chunk i

– these impacts are summed up to produce a total strength/total need odds for chunk i;

t_k^{-d}: the strengthening impact of a presentation k on the total need odds for chunk i is a power function of time t_k^{-d}, where t_k is the time elapsed since presentation k

– that is, t_k is the delay, i.e., the period of time elapsed between the time of presentation k and the time of retrieval t_{now};
– raising the delay t_k to the $-d$ power (the decay) produces the power law of forgetting.

The parameter d in the base activation equation is usually set to $\frac{1}{2}$, so the equation simplifies to:

[2]Recall that odds are a deterministic function of probability: $o_i = \frac{p_i}{1-p_i}$, where p_i is the 'need probability' of chunk i, i.e., the probability that chunk i is needed at the retrieval time t_{now}.

(7) Base-level learning equation (simplified: $d = 0.5$): $B_i = \log\left(\sum_{k=1}^{n} t_k^{-\frac{1}{2}}\right) = \log\left(\sum_{k=1}^{n} \frac{1}{\sqrt{t_k}}\right)$.

Let's work through an example. Assume we have the following chunk of type `word` in declarative memory, repeated from Chap. 2, and represented in both graph and AVM form:

(8)
$$\begin{tikzcd}
 & \text{sg} & \\
\text{car} & \boxed{\text{WORD } (B_i)} \arrow[l, "\text{FORM}"'] \arrow[r, "\text{MEANING}"] \arrow[u, "\text{NUMBER}"'] \arrow[d, "\text{CATEGORY}"] & [\![\text{car}]\!] \\
 & \text{noun} &
\end{tikzcd}$$

(9)
$$ {}_{\text{WORD } (B_i)}\left[\begin{array}{ll} \text{FORM:} & \text{car} \\ \text{MEANING:} & [\![\text{car}]\!] \\ \text{CATEGORY:} & \text{noun} \\ \text{NUMBER:} & \text{sg} \end{array}\right] $$

Assume this chunk is presented 5 times, once every 1.25 s, starting at time 0 s. We want to plot its base-level activation for the first 10 s.

In [py21] below, we define a `base_activation` function. Its inputs are the vector of presentation times for the chunk (`pres_times`—the first argument of the function), and also the vector consisting of the moments of time at which we want to compute the activation (`moments`—the second argument of the function). You can think of these `moments` of time as potential retrieval times. The output of the `base_activation` function is the vector `base_act` of base-activation values at the corresponding `moments` of time.

- line 2: we initialize the base activation `base_act`: we set it to be a long vector of 0s, as long as the number of moments we want to compute the activation for;
- line 3: the `for` loop on lines 3–6 in [py21] computes the actual activation: for every point `idx` (short for 'index') at which we want to compute the activation, we do several things, discussed below;
- line 4: we identify the moment in time at which we should compute the activation, namely `moments[idx]`; we identify the presentation times that precede this moment, namely `pres_times<moments[idx]`, since they are the only presentations contributing to base activation at this moment in time; we retrieve these presentation times and store them in the variable `past_pres_times`;

- lines 5–6: with these past presentation times in hand, we compute base level activation following the base level equation in (7); first, we compute the time intervals since those past presentations: `moments[idx]−past_pres_times`; then we take the square root of these intervals `np.sqrt(...)` and then, the reciprocal of those square roots `1/np.sqrt(...)`; finally, we sum all those reciprocals `np.sum(...)`;
- lines 7–9: now, the vector `base_act` stores exponentiated activations; to get to the actual activations, we need to take the log of the quantities currently stored in `base_act`; since log(0) is undefined, we identify the non-0 quantities in `base_act` (line 7), take the log of those quantities and replace them with their logs (lines 8–9).

[py21]
```
>>> def base_activation(pres_times, moments):                                1
...     base_act = np.zeros(len(moments))                                    2
...     for idx in range(len(moments)):                                      3
...         past_pres_times = pres_times[pres_times<moments[idx]]            4
...         base_act[idx] = \                                                5
...             np.sum(1/np.sqrt(moments[idx] - past_pres_times))            6
...     non_zero_activations = np.not_equal(base_act, 0)                     7
...     base_act[non_zero_activations] = \                                   8
...         np.log(base_act[non_zero_activations])                           9
...     return base_act                                                      10
...                                                                          11
>>> pres_times = np.linspace(0, 5000, 5)/1000                                12
>>> pres_times                                                               13
array([0.  , 1.25, 2.5 , 3.75, 5.  ])                                        14
>>> moments = np.arange(10000)/1000                                          15
>>> moments                                                                  16
array([0.000e+00, 1.000e-03, 2.000e-03, ..., 9.997e+00, 9.998e+00,           17
       9.999e+00])                                                           18
>>> base_act = base_activation(pres_times, moments)                          19
>>> base_act                                                                 20
array([0.        , 3.45387764, 3.10730405, ..., 0.62436509, 0.6242921 ,     21
       0.62421912])                                                          22
```

On line 12 in **[py21]**, we generate a vector of 5 presentation times evenly spaced between 0 and 5000 ms. As shown on line 14, these presentation times are at 0, 1.25, 2.5, 3.75 and 5 s. On line 15, we generate a vector of the moments in time at which we want to compute the activation: we want to see the ebbs and flows of activation for the first ten seconds, and we want to see this every ms, so we generate a vector with 10000 numbers—from 1 to 10000 ms (lines 17–18). Finally, we compute the base activation relative to these moments and presentation times using our `base_activation` function.

We can now plot the result: the code for the plot is provided in **[py22]** below, and the plot itself is provided in Fig. 6.5.

[py22]
```
>>> def generate_base_activation_figure():
...     fig, ax = plt.subplots(ncols=1, nrows=1)
...     fig.set_size_inches(5.5, 3)
...     ax.plot(moments, base_act, linestyle='-')
...     ax.plot(pres_times, np.ones(5) * -0.3, 'ro')
...     ax.set_title('Base activation (blue) and 5 presentations (red)')
...     ax.set_xlabel('Time (s)')
...     ax.xaxis.set_major_locator(mpl.ticker.MultipleLocator(1))
...     ax.set_ylabel('Base activation (logits)')
...     #plt.xticks(rotation=45)
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/base_activation.eps')
```

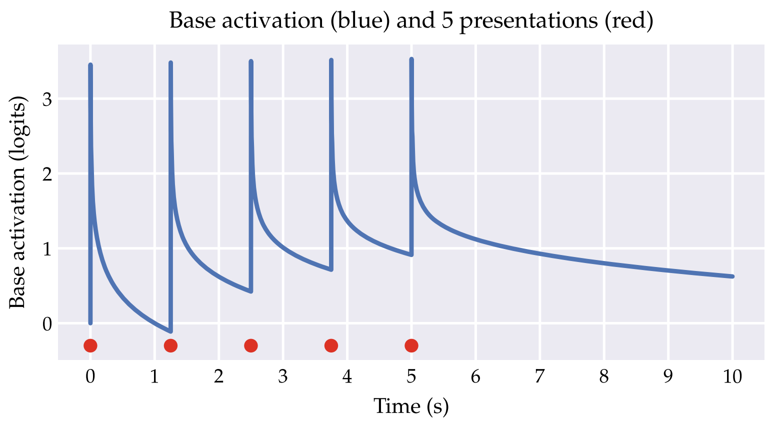


Fig. 6.5 Base activation as a function of time (10 s, 5 presentations)

```
...     plt.savefig('./figures/base_activation.png')
...     plt.savefig('./figures/base_activation.pdf')
...
>>> generate_base_activation_figure()
```

We see that the activation of the chunk spikes after each presentation and then drops as a power function of time until the next presentation/spike. We also see that the maximum activation (the height of the spikes) slowly increases with each presentation, just as the decay of activation becomes more and more mild. After the fifth presentation, the activation of the chunk decreases pretty slowly, and even in the long term (at 10 s), its activation is higher than the activation it had shortly after the first presentation (say, at 500 ms). Thus, after repeated presentations, we can say that the chunk has been retained in 'long-term' memory.

What Fig. 6.5 shows is that forgetting + rehearsal is an essential part of remembering. The model captures the common observation that cramming for an exam never works long term (the activation of the newly learned facts decreases very steeply after the first presentation), while properly spaced rehearsals or practice lead to long-term retention.

To conclude this section, we note that ACT-R does not distinguish between short-term and long-term memory. Both of them are distinct from working memory, which can be thought of as the state of the buffers at any given time. Modeling memory as a power-function of time generates the proper short-term memory behavior (after one presentation), as well as the proper long-term memory behavior (after a series of presentations).

6.3 The Attentional Weighting Equation

In addition to base activation, a chunk's activation depends on the context in which it is needed. What counts as "context" within the ACT-R cognitive architecture? Context for cognitive processes is the information that is instantaneously available to the procedural module: all the buffers and the chunks that reside in them (basically, working memory).[3]

We know that chunks consist of slot-value pairs. To capture the role of context, ACT-R assumes that any chunk V that appears as the value of some slot in a buffer spreads activation to (i) chunks in declarative memory that have V as one of their values, and (ii) chunks in declarative memory that are content-identical to V, i.e., they consist of the same set of slot-value pairs as V. This context-driven boost in activation for chunks in declarative memory is known as *spreading activation*.

An example will help shed more light on the workings of spreading activation. Suppose that only one buffer carries a chunk, say, the imaginal buffer. And the chunk in the imaginal buffer is the representation of the word *car*. We assume that the chunk has four slots: FORM, MEANING, CATEGORY and NUMBER. Each of these slots, in turn, has a chunk as its value: the form, the interpretation, the syntactic category and the morphological number, respectively. Each of these values comes with a weight, as shown in (10).

(10)
$$ {}_{\text{WORD}}\left[\begin{array}{ll} \text{FORM:} & \text{car } (W_1) \\ \text{MEANING:} & [\![\text{car}]\!] \ (W_2) \\ \text{CATEGORY:} & \text{N } (W_3) \\ \text{NUMBER:} & \text{sg } (W_4) \end{array}\right] $$

- the form *car* has weight W_1
- the meaning $[\![\text{car}]\!]$ has weight W_2
- the syntactic category N has weight W_3
- the number specification sg has weight W_4.

Any chunk i in declarative memory that shares values with the imaginal chunk (10)[4] receives spreading activation proportional to ($\propto$) the weights W_j (for $j \in \{1, 2, 3, 4\}$) of the values that chunk i has in common with the imaginal chunk.

That is, chunk i receives an activation boost just by virtue of containing any of the four values in (10), i.e., the form *car* (W_1), $[\![\text{car}]\!]$ (W_2), N (W_3) or sg (W_4). Intuitively, sharing a value with a context chunk (like the *car* chunk in the imaginal buffer)

[3]This is sometimes called "the general context". Besides the general context, there is also a specific context, relevant for partial matching. Partial matching is available in `pyactr`, but it is not covered in this book. See Lebiere (1999) for a detailed discussion of partial matching and an example of how it can be applied to cognition and learning.

[4]Or that consists of the same slot-value pairs, but we ignore this case for expositional simplicity.

'connects' chunk i in declarative memory to the context chunk. Activation can now spread/flow along this connection, and this spreading activation is proportional to the weight W_j (in symbols: $\propto W_j$) of the connecting value.

Note that these values are themselves chunks, but we will continue to refer to them as values and explicitly call 'chunk' only the chunk in declarative memory that receives spreading activation, and the context chunk that is the source of the spreading activation.

We keep insisting that spreading activation is proportional to a weight W_j ($\propto W_j$), but not identical to it, because chunk i in declarative memory does not simply add W_j to its activation. Every weight W_j, or *source activation*, is scaled by an *associative strength* S_{ji}, and it is the product $W_j \cdot S_{ji}$ that gets added to the activation of chunk i.

Intuitively, we can think of this associative strength as the strength (or the resistance, if you will) of the connection between chunk i (the activation-receiving chunk in declarative memory) and the context chunk that is the source of spreading activation. Every value shared between chunk i in declarative memory and the context chunk 'creates' a connection along which activation W_j can spread/flow, but this connection has a strength/resistance S_{ji} specific to the value j that 'created' the connection and to the activation-receiving chunk i.

In our specific example, we have four weights/source activations W_1, W_2, W_3, W_4 and four corresponding associative strengths S_{1i}, S_{2i}, S_{3i}, S_{4i}.

Associative, or 'connection', strength is basically a measure of how predictive any specific value in a context chunk is of chunk i. This idea of 'predictive' association will make more sense in a moment when we introduce the concept of fan, and will be further clarified in Chap. 8, where we discuss and model the classic fan experiment in Anderson (1974).

In our example, the form *car* has weight W_1, but we don't simply add that to the base activation B_i of our chunk i in declarative memory. Instead, we scale it by the associative strength S_{1i}, which is the strength of the connection created by the value *car* (which resides in the FORM slot of the imaginal buffer) and our chunk i (which resides in declarative memory, and which has the same value *car* in one of its slots). Thus, the activation boost spreading to chunk i in declarative memory from the value *car* in the imaginal buffer is given by the product $W_1 S_{1i}$.

The resulting total activation A_i for any chunk i in declarative memory will therefore be the sum of its base activation B_i (which reflects its past history of usage) and whatever spreading activation it gets from the cognitive context, which in our example is restricted to the imaginal buffer. When activation spreads along all four values of the imaginal chunk (that is, chunk i in declarative memory has all these four values in its slots), we have:

$$(11) \quad A_i = B_i + W_1 S_{1i} + W_2 S_{2i} + W_3 S_{3i} + W_4 S_{4i}$$

How are we to set the weights and the associative strengths? One answer that immediately comes to mind is: empirically. We set some low-information/vague priors over the weights and the associative strengths and infer them from suitable experimental data. This is in fact what we will do in Chap. 8. For now, we will simply discuss some reasonable default values.

Every chunk in a buffer that spreads activation is assumed to have a total source activation W that gets evenly distributed among the values that reside in the slots of that chunk. W is by default set to 1. In our example, this would mean that $W_1 = W_2 = W_3 = W_4 = \frac{1}{4}$.

(12) Default value for source activation: $W_j = \frac{W}{n}$, where:

- j goes from 1 to the number of slots n that carry a value;
- W is by default set to 1.

Let's turn now to the associative strengths S_{ji}, where i is the chunk in declarative memory that receives spreading activation, and j, which varies from 1 to n, is a value in the cognitive context buffer that spreads activation (this buffer has n slots that carry a value). For these associative strengths S_{ji}, we want to capture the intuition that:

- the association should be 0 if j does not associate with i (it is not predictive of i in any way),
- it should be high if j uniquely associates with the chunk i (because j is then highly predictive of i), which would happen if there is no other chunk in declarative memory that is associated with j, and finally,
- the association strength should decrease as more and more chunks in declarative memory are associated with j, since j becomes less predictive of any of these chunks.

This intuition is captured by the following formula (see Anderson and Schooler 1991; Anderson 2007):

(13) $S_{ji} \approx \log \frac{prob(i|j)}{prob(i)}$

In words, the strength of association between value j in a context buffer that spreads activation and chunk i in declarative memory that receives activation is approximately the log probability of needing chunk i from memory conditional on the fact that value j is present in the buffer, 'normalized' by the probability that chunk i is unconditionally needed.

Formally, this is the pointwise mutual information (**pmi**) between (i) the event that chunk i is needed/requested from declarative memory and (ii) the event that j is a value in the activation-spreading context buffer, i.e., a chunk in one of the slots of that context buffer:

(14) Pointwise mutual information between two events i, j:
$\mathbf{pmi}(i, j) = \log \frac{prob(i,j)}{prob(i)prob(j)} = \log \frac{prob(i|j)}{prob(i)} = \log \frac{prob(j|i)}{prob(j)}$

As the definition in (14) above shows, **pmi** is a symmetric measure of association between single events (not an expectation, like mutual information). It can have both negative and positive values, and it is 0 if the events are independent. Understanding association strengths S_{ji} in terms of the **pmi** between the declarative memory chunk i

and the value j in the cognitive context makes intuitive sense: strength of association is a measure of how predictive the contextual value j is of the need to retrieve chunk i from memory. See also Appendix A.3 in Reitter et al. (2011) for a short discussion.

ACT-R has developed a way to estimate the values in (13). First, for cases in which i and j are not associated in any way, i.e., they are independent, the joint probability $prob(i, j)$ is the product of the marginals $prob(i)prob(j)$, so:

(15) $S_{ji} = \mathbf{pmi}(i, j) = \log \frac{prob(i,j)}{prob(i)prob(j)} = \log \frac{prob(i)prob(j)}{prob(i)prob(j)} = \log 1 = 0$

To put it differently, if i and j are independent, the contextual value j is not predictive at all of the need to retrieve chunk i from declarative memory, so $prob(i\,|\,j) = \frac{prob(i,j)}{prob(j)} = \frac{prob(i)prob(j)}{prob(j)} = prob(i)$. Therefore, the contextual value j does not boost the activation of chunk i in any way, and to ensure that no activation spreads, we zero out the 'connection' strength $S_{ji} = \log \frac{prob(i|j)}{prob(i)} = \log \frac{prob(i)}{prob(i)} = \log 1 = 0$.

If i and j are not independent, i.e., there is an association between them, so j has some predictive value with regards to the need-probability of chunk i, the common estimate for $prob(i\,|\,j)$ is as follows:

(16) $prob(i|j) = \frac{1}{\text{fan}_j}$, where:

- fan_j is the number of chunks associated with j in declarative memory, i.e.,
- fan_j is the number of chunks that have j as a value in one of their slots.

The intuition behind this common estimate is that a value j in the cognitive context is *equally predictive* of any chunk in declarative memory that it is associated with. Basically, in the past, whenever we had value j in the cognitive context, we were equally likely to need to retrieve any of the declarative memory chunks associated with j. This kind of assumption is unrealistic in a naturalistic, 'ecologically valid' setting, but it is probably reasonable in the context of a counterbalanced experiment.

A common estimate for $prob(i)$ is:

(17) $prob(i) = \frac{1}{|dm|}$, where:

- $|dm|$ is the size of declarative memory (*dm*): the number of chunks present in *dm*.

Again, this is extremely unrealistic since it assumes that all the chunks in declarative memory have the same history of past usage (or no history of past usage), so they have the same probability of being needed/retrieved. This estimate makes sense as a flat uniform prior used for convenience, perhaps in an experimental setting where frequentist and Bayesian posterior estimates of need probabilities for experimental items are intended to be identical.

With these two assumptions in place, associative strength S_{ji} can be estimated as follows:

(18) $S_{ji} = \log \frac{|dm|}{\text{fan}_j} = \log |dm| - \log \text{fan}_j$.

Note how the dependency on a specific declarative memory chunk i disappears because of the (unrealistic) uniformity assumptions built into the $prob(i|j)$ and $prob(i)$ estimates.

It is hard to estimate the size of declarative memory, so the minuend $\log|dm|$ is often treated as a free (hyper)parameter S, with the requirement that S should be larger than $\log \text{fan}_j$, for any value j. If this was not so, association could be negative in some cases, i.e., in some cases, association strength would yield negative spreading activation, decreasing (inhibiting) base activation for some items rather than simply failing to boost it.

In sum, the final form for associative strength that is commonly used in ACT-R modeling is as follows:

(19) Associative strength equation between a value j and a chunk i:

$$S_{ji} = \begin{cases} S - \log \text{fan}_j & \text{if } i \text{ and } j \text{ are associated, i.e., } i = j \text{ or } j \text{ is a value of } i \\ 0 & \text{otherwise} \end{cases}$$

where:

- S is the maximum associative strength, a free (hyper)parameter.

Putting this together, we arrive at the activation equation in (20). This shows how spreading activation from just one buffer affects the total activation of elements in declarative memory. Extending to more than one buffer is easy: we just sum up the spreading activation from all the buffers, as shown in (21).

(20) Activation equation (simplified to one buffer): $\boxed{A_i = B_i + \sum_{j=1}^{m} W_j S_{ji}}$,

for any chunk i in declarative memory and all values j of the chunk in the buffer.
This equation has three major components:

a. Base-level learning equation: $B_i = \log\left(\sum_{k=1}^{n} t_k^{-d}\right) = \log\left(\sum_{k=1}^{n} \frac{1}{\sqrt{t_k}}\right)$ (since usually $d = 0.5$), where t_k is the time since the k-th practice / presentation of chunk i.

b. Attentional weight equation: see (12)

c. Associative strength equation: see (19)

(21) Activation equation (generalized to all buffers): $\boxed{A_i = B_i + \sum_{k=1}^{n} \sum_{j=1}^{m_k} W_{kj} S_{ji}}$,

for any chunk i in declarative memory, all buffers k and all values j of the chunk present in buffer k, where the chunk in buffer k has m_k slots.
This equation has the same three major components as the simpler one in (20) above. Differences:

a. We sum over all buffers k, from 1 to n buffers in the cognitive context.

b. The weights / source activations W_{kj} are indexed with both the value j that is their source, as well as the buffer k where value j is located.

To understand this a little better, think of the typical scenario in which spreading activations, i.e., source activations scaled by associative strengths, are used. The values j we typically consider are values stored in the slots of the imaginal or the goal chunk. These buffers drive the cognitive process, so they provide a crucial part of the cognitive context in which we might want to retrieve items from memory.

When we have a goal or an imaginal chunk, we associatively bring to salience, i.e., spread activation to, chunks in declarative memory that are associated with the current imaginal or goal chunk, since these declarative memory chunks might be needed. We operationalize this 'association' between a chunk in the cognitive context and a chunk i in declarative memory in terms of the chunks being content identical (consisting of the same same set of slot-value pairs) or sharing some value j in some of their slots.

This essentially results in increasing the activation of those declarative memory chunks that are related to the current cognitive context, i.e., ultimately, that are potentially relevant to the current stage of the cognitive process we are involved in. The associative strength S_{ji} is really the probability that chunk i is relevant given a cognitive context in which we attend to the value j.

One intuitive way to think about the activation of chunks in declarative memory and the additive relation between base activation and spreading activation is to imagine declarative memory was a sea of darkness with small rafts, i.e., chunks, floating everywhere on it. Each raft has a small light, and the brightness of that light indicates its total activation: the brighter that light is, the easier the raft is to find and grab—that is, we can retrieve it more accurately and more quickly.

The light on each raft is powered by two power sources. One of them is a rechargeable battery stored on the raft itself (well, it's more like a capacitor, but let's ignore this). This reflects base activation, i.e., the history of previous usages of a chunk. Every time we use a chunk (retrieve a raft), we plug its 'local battery' in for a quick charge. Immediately after that, the battery will have more power, so the light will be brighter.

The second source of power that can increase the brightness of the light on a raft is the current cognitive context, specifically the values held in the buffers. If these values are also stored on some of the rafts in declarative memory (that is, they are the values of some of the features of those chunks), they can act as wires delivering extra power to the lights on the rafts.

Let's focus on a specific chunk in some buffer in our cognitive context. Each value j in that chunk has a set amount of battery power (these are the source activations, i.e., the W_j values), and that power gets distributed to all the rafts in declarative memory that also store that value. This immediately predicts that the more rafts a value in the cognitive context is connected with—in ACT-R parlance, the higher the 'fan' of a value—, the less power it will transmit to each individual raft. This 'fan effect' is discussed in detail in Chap. 8.

The amount of power/activation that ‘spreads’ from the goal/imaginal buffer (or any buffer in the cognitive context that we decide to spread activation from) depends not only on the ‘battery power’ W_j of each value j, but also on the specific ‘wires’ connecting the buffer and the rafts/chunks in declarative memory that share that value. Different wires have different ‘resistance’ characteristics S_{ji}, and the extra power boost W_j is modulated by the ‘resistance’/strength of the connection.

Let us go through an example. Suppose we have the word *car* in the imaginal buffer and our declarative memory consists of two chunks x and y that are singular nouns (say, *book* and *pen*) and one chunk z that is a plural noun (say, *books*). The singular nouns x, y have two values in common with the *car* chunk in the imaginal buffer, namely the singular number and the noun category. The plural noun z has only one value in common with the *car* chunk, namely, the noun category.

The activation of the plural noun z is calculated in (22) below. Recall that j are the values of the *car* chunk in the imaginal buffer, and $j = 1$ for the form slot, $j = 2$ for the meaning slot, $j = 3$ for the syntactic category slot and $j = 4$ for the number morphology slot. Since there are 4 total slots in the imaginal buffer chunk, the source activations are all set to $\frac{1}{4}$ (see (12) above, with $W = 1, n = 4$).

Turning to association strengths, we note that only the value in the syntactic category slot (i.e., noun/N) spreads activation. This means that the association strengths for all the other values are zero, i.e., $S_{1z} = S_{2z} = S_{4z} = 0$. The fan of the value in the syntactic category slot, namely fan_3, is 4, since this value is N and there are four nouns total in declarative memory: x, y, z and, we assume, also the *car* chunk currently in the imaginal buffer.

The calculation, therefore, proceeds as follows:

(22)
$$\begin{aligned} A_z &= B_z + \sum_{j=1}^{m} W_j S_{jz} \\ &= B_z + W_1 S_{1z} + W_2 S_{2z} + W_3 S_{3z} + W_4 S_{4z} \\ &= B_z + \tfrac{1}{4}\cdot S_{1z} + \tfrac{1}{4}\cdot S_{2z} + \tfrac{1}{4}\cdot S_{3z} + \tfrac{1}{4}\cdot S_{4z} \\ &= B_z + \tfrac{1}{4}\cdot 0 + \tfrac{1}{4}\cdot 0 + \tfrac{1}{4}\cdot (S - \log \text{fan}_3) + \tfrac{1}{4}\cdot 0 \\ &= B_z + \tfrac{1}{4}\cdot (S - \log 4) \end{aligned}$$

In contrast, the activation of the singular noun x proceeds as shown in (23) below. This time, the singular receives spreading activation both from the syntactic category value (N) and from the number specification (sg).

The activation spreading from the number value is higher than the one spreading from the syntactic category value because there are 4 nouns total ($\text{fan}_3 = 4$), but only 3 of them are singular ($\text{fan}_4 = 3$). This makes intuitive sense: values that appear only in handful of chunks are more predictive of these chunks and should boost their activation more than values that are more frequent and, therefore, less discriminatory.

(23)
$$\begin{aligned} A_x &= B_x + \sum_{j=1}^{m} W_j S_{jx} \\ &= B_x + W_1 S_{1z} + W_2 S_{2z} + W_3 S_{3x} + W_4 S_{4x} \\ &= B_z + \tfrac{1}{4}\cdot S_{1z} + \tfrac{1}{4}\cdot S_{2z} + \tfrac{1}{4}\cdot S_{3x} + \tfrac{1}{4}\cdot S_{4x} \\ &= B_x + \tfrac{1}{4}\cdot 0 + \tfrac{1}{4}\cdot 0 + \tfrac{1}{4}\cdot (S - \log \text{fan}_3) + \tfrac{1}{4}\cdot (S - \log \text{fan}_4) \\ &= B_x + \tfrac{1}{4}\cdot (S - \log 4) + \tfrac{1}{4}\cdot (S - \log 3) \end{aligned}$$

6.4 Activation, Retrieval Probability and Retrieval Latency

Now that we have a formal model of activation with its two components, namely:

- base activation that encodes the effects of prior use on memory, and
- spreading activation that encodes contextual effects

we can turn to how we can predict human performance based on activation.[5]

Recall that the two behavioral measures we are trying to predict are (i) the probability of selecting a particular response, specifically of retrieving a chunk from memory, and (ii) the latency of selecting a response, specifically the time taken by the retrieval process. The relevant equations are provided in (24) and (25) below.

(24) Probability of retrieval equation: $P_i = \frac{1}{1+e^{-\frac{A_i-\tau}{s}}}$, where:

- s is the noise parameter and is typically set to about 0.4
- τ is the retrieval threshld, i.e., the activation at which we have a chance-level (0.5) retrieval probability[6]

(25) Latency of retrieval equation: $T_i = Fe^{-fA_i}$, where:

- F is the latency factor (basically, an intercept on log time scale)
- f is the latency exponent (a slope on log time scale)[7]

In addition to the activation A_i of chunk i, these equations have a few parameters. The threshold parameter τ in the probability of retrieval equation (24) and the latency factor F in the (25) vary from model to model, but there is a general relationship between them, provided in (26).

(26) $F \approx 0.35e^{\tau}$, i.e., the retrieval latency at threshold is approximately 0.35 s/350 ms[8]

To understand the two equations in (24) and (25) a bit better, recall our discussion of base activation, and the important remark that activation A_i is really the log of the need-odds of a particular chunk i. That is, A_i is the logit (log odds) transformation of the need-probability of chunk i.

[5]Note that these two components of activation have roles similar to the model $\mathcal{M}$ and the variable assignment g parameters of the interpretation function $[\![\cdot]\!]^{\mathcal{M},g}$ in formal semantics. The model $\mathcal{M}$ is parallel to base activation as it encodes more permanent, context-invariant information, while the variable assignment g is parallel to spreading activation as it encodes contextually-sensitive information of a more transient nature.

[6]When $A_i = \tau$, we have: $P_i = \frac{1}{1+e^{-\frac{\tau-\tau}{s}}} = \frac{1}{1+e^0} = \frac{1}{2}$.

[7]On log time scale, we have: $\log T_i = \log(Fe^{-fA_i}) = \log F - fA_i$.

[8]When $A_i = \tau$ and f is set to its default value of 1, we have: $T_i = Fe^{-A_i} = Fe^{-\tau} \approx 0.35e^{\tau}e^{-\tau} = 0.35$.

The need-probability of chunk i is just the probability that chunk i is the one needed to satisfy the current memory retrieval request. Talking about need-probability is interchangeable with talking about need-odds, or activation, which is just need-logits.

In (27) below, we show how we can compute need-odds and activation (need-logits) if we are given the need-probability p_i of chunk i. In (28), we show how we can compute need-odds and need-probability if we are given the activation A_i (i.e., we are given the need-logits A_i) for chunk i.

(27) Given the need-probability p_i for chunk i, we have:

- need-odds $o_i = \frac{p_i}{1-p_i}$
- need-logits / activation $A_i = \log o_i = \log \frac{p_i}{1-p_i}$

(28) Given the activation / need-logits A_i for chunk i, we have:

- need-odds $o_i = e^{A_i}$
- need-probability $p_i = \frac{o_i}{1+o_i} = \frac{e^{A_i}}{1+e^{A_i}}$;
  equivalently, $p_i = \frac{1}{1+\frac{1}{o_i}} = \frac{1}{1+\frac{1}{e^{A_i}}} = \frac{1}{1+e^{-A_i}}$

The very last equation in (28) shows how to compute need-probability p_i when we are given activation (need-logits) A_i. But this is exactly the equation we used to obtain probability of retrieval in (24) above. The only difference is that, in (24), we add two parameters, the threshold τ and the noise s, which enable us to make the model more realistic/flexible so that we can fit different kinds of data well.

Thus, the probability of retrieval equation immediately follows from the fact that we take activation to encode the log-odds (logit) transformation of need-probability.

The latency of retrieval equation in (25) is equally intuitive. Ignoring the parameters F and f, i.e., setting them to 1 (incidentally, 1 is the default value for f), we have that:

(29) $T_i = e^{-A_i} = \frac{1}{e^{A_i}} = \frac{1}{o_i}$ (if we set $F = f = 1$)

That is, the retrieval latency for chunk i is inversely proportional to the need-odds of i. The higher the need-odds for chunk i, the less time it will take to retrieve it. The lower the need-odds for chunk i, the more time it will take to retrieve it.

The need odds for a chunk i are high if the chunk has been used a lot and/or recently (this comes from base activation), and the chunk is highly relevant given the current cognitive context (this comes from spreading activation). It therefore makes sense that such a chunk would be easy/fast to retrieve: it was retrieved a lot and/or recently, and it is strongly associated with what the cognitive process is currently attending to.

Finally, modeling retrieval times/latencies as inversely proportional to need-odds makes mathematical sense. While probabilities take values in the interval [0, 1], odds take values in the interval $[0, \infty)$, i.e., in the set of positive real numbers (and 0), which is the same interval in which reaction times/latencies also take values.

Let's now work through an example. We will plot the probability and latency of retrieval for the same hypothetical case as the one in Fig. 6.5 above, assuming the activation of the `carLexeme` chunk under consideration is just its base-level activation. We set the parameters as follows:

- noise $s = 0.4$
- threshold $\tau = 0.3$
- latency factor $F = 0.47$
- latency exponent $f = 1$.

Note that according to the equation in (26), $F \approx 0.35e^{0.3} \approx 0.35 \times 1.35 \approx 0.47$ s, which is what we set our F value to.[9] Of course, we pulled the values for these parameters out of thin air for this particular example. In general, we want to use statistical inference (e.g., Bayesian methods with `pymc3`) to estimate these parameters from the data, and we will do exactly this when we model lexical decision tasks in the next chapter (Chap. 7). But we set the parameters to these (more or less default) values for the current example.

In **[py23]**, we use the previously computed vector `base_act` of base activations to compute the probabilities of retrieval and the latencies of retrieval for the 10 s we are interested in. We then plot these three curves (activation, retrieval probability, retrieval latency). The code for the plot is provided in **[py24]**, and the plots are provided in Fig. 6.6.

[py23]
```
>>> s = 0.4
>>> tau = 0.3
>>> F = 0.47
>>> f = 1
>>> prob_retrieval = 1 / (1 + np.exp(-(base_act - tau)/s))
>>> prob_retrieval
array([0.3208213 , 0.99962368, 0.99910542, ..., 0.69230396, 0.69226509,
       0.69222622])
>>> latency_retrieval = F * np.exp(-f*base_act)
>>> # latency of retrieval in ms
>>> (latency_retrieval * 1000).astype("int")
array([470,  14,  21, ..., 251, 251, 251])
>>> threshold_prob_scale = 1 / (1 + np.exp(-(tau-tau)/s)) # =1/(1+1)
>>> # it's 1/2, i.e., 50% probability when activation is at threshold
>>> threshold_prob_scale
0.5
>>> threshold_latency_scale = F * np.exp(-tau)
>>> # time to retrieve in ms when activation is at threshold
>>> (threshold_latency_scale * 1000).astype("int")
348
```

[py24]
```
>>> def generate_prob_latency_figure():
...     fig, (ax1, ax2, ax3) = plt.subplots(ncols=1, nrows=3, sharex=True)
...     fig.set_size_inches(5.8, 8.3)
...     # plot 1
...     ax1.plot(moments, base_act, linestyle='-')
...     ax1.plot(pres_times, np.ones(5) * -0.3, 'ro')
...     ax1.plot(moments, np.ones(len(moments)) * tau,\
...              linestyle='--', color='black')
...     ax1.annotate('Threshold', xy=(8, tau - 0.26), size=10)
...     ax1.set_title('Activation (blue),\
...                   threshold (black) and 5 presentations (red)')
```

[9]Note that this value is close to $F = 0.46$ s in Vasishth et al. (2008, 692), and it is pretty far from $F = 0.14$ s in Lewis and Vasishth (2005, 382).

```
...     ax1.set_ylabel('Activation (logits)')
...     # plot 2
...     ax2.plot(moments, prob_retrieval, linestyle='-')
...     ax2.plot(pres_times, np.zeros(5), 'ro')
...     ax2.plot(moments, np.ones(len(moments)) * threshold_prob_scale,\
...             linestyle='--', color='black')
...     ax2.annotate('Threshold', xy=(8, threshold_prob_scale - 0.067),\
...                  size=10)
...     ax2.set_title('Retrieval probability (blue), \
...                    threshold (black) and 5 presentations (red)')
...     ax2.set_ylabel('Probability of retrieval')
...     # plot 3
...     ax3.plot(moments, latency_retrieval, linestyle='-')
...     ax3.plot(pres_times, np.ones(5) * -0.03, 'ro')
...     ax3.plot(moments, np.ones(len(moments)) * threshold_latency_scale,\
...             linestyle='--', color='black')
...     ax3.annotate('Threshold', xy=(8, threshold_latency_scale - 0.037),\
...                  size=10)
...     ax3.set_title('Retrieval latency (blue), \
...                    threshold (black) and 5 presentations (red)')
...     ax3.set_xlabel('Time (s)')
...     ax3.set_ylabel('Latency of retrieval (s)')
...     # clean up and save
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/prob_latency_figure.eps')
...     plt.savefig('./figures/prob_latency_figure.png')
...     plt.savefig('./figures/prob_latency_figure.pdf')
...
>>> generate_prob_latency_figure()
```

In Fig. 6.6, we plot the threshold τ as an interrupted black line in every plot:

- in the top panel, we plot its raw value (0.3) on the activation (logit) scale;
- in the middle panel, the threshold is at 50% probability, as intended given that activation at threshold level should yield even odds of retrieval (1/1; chance level);
- in the bottom panel, the threshold is at about 350 ms, which is the time of retrieval when activation is at threshold level; this is actually determined by the constant 0.35 (350 ms) we used in (26).

Figure 6.6 shows that, as activation increases above threshold after the fourth and fifth presentation/rehearsal of the `carLexeme` chunk (see the top panel in the figure), retrieval accuracy increases above chance (see the plot in the middle panel) and the retrieval becomes faster and faster (retrieval time decreases below 340 ms; see plot in bottom panel).

Remarkably, the ACT-R account of declarative memory unifies two separate measures (retrieval accuracy and retrieval latency) under one quantity: activation. Furthermore, activation can be independently derived if we know, or can reasonable conjecture:

- the pattern of previous use for a chunk—this will give us the base activation component;
- the cognitive context—this will give us the spreading activation component.

In the following chapter, we will apply this model to lexical access and we will evaluate how well the ACT-R model of memory accounts for both accuracy and latency in lexical decision tasks, as well as latencies in self-paced reading experiments.

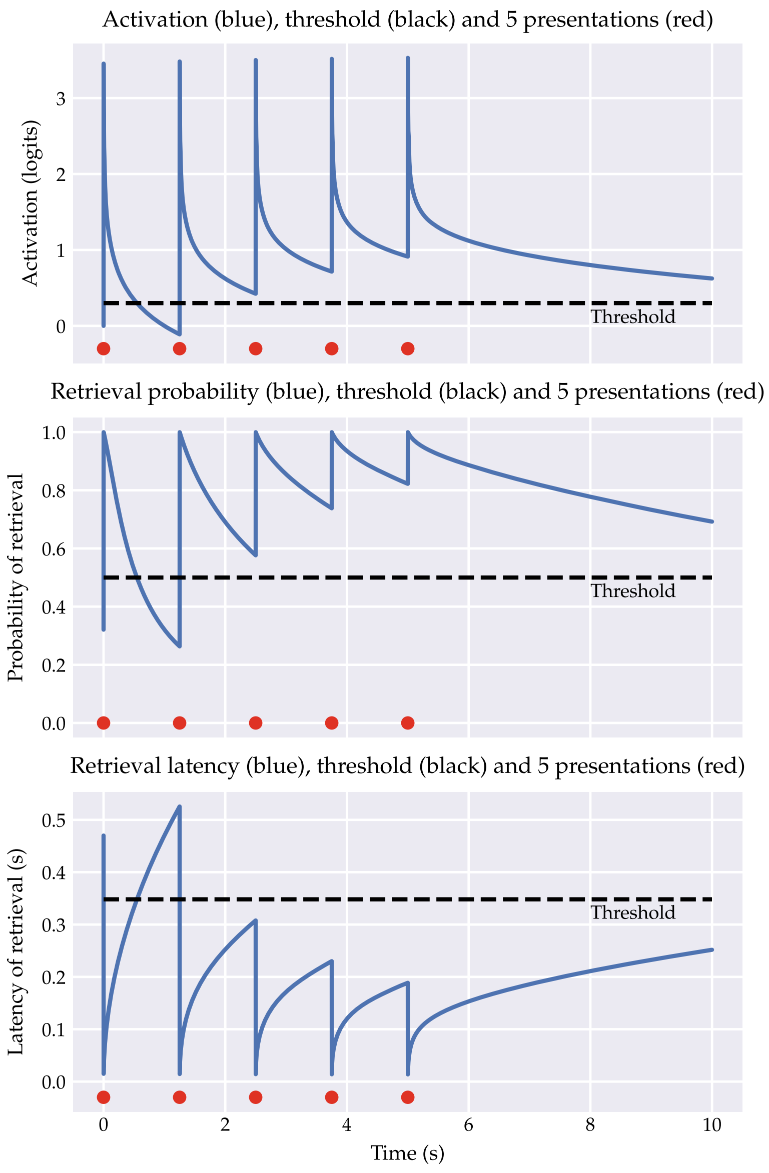


Fig. 6.6 Activation, retrieval probability and retrieval latency as a function of time

6.5 Appendix

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

File **ch6_code.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch6_code.py.

Chapter 7
Competence-Performance Models for Lexical Access and Syntactic Parsing

In Chap. 4, we introduced a simple lexical decision task and a simple left-corner parser. The models we introduced in that chapter might be sufficient with respect to the way they simulate interactions with the environment, but they are too simplistic in their assumptions about memory, since memory retrievals are not dependent on any parameters of the retrieved word. In this chapter, we will improve on both models by incorporating the ACT-R model of declarative memory we just introduced in the previous chapter.

We start with a discussion of word frequency and the way it modulates lexical decision (Sect. 7.1). We then build several ACT-R models for lexical-decision tasks that incorporate the subsymbolic declarative memory components introduced in the previous chapter and take into account word frequency in a theoretically motivated way (Sects. 7.2–7.4). In Sect. 7.5, we do the same for a left-corner parser.

7.1 The Log-Frequency Model of Lexical Decision

One very robust parameter affecting latencies and accuracies in lexical decision tasks is frequency (Whaley 1978). In fact, frequency effects have been found not just in lexical decision tasks, but in many if not all tasks that involve some kind of lexical processing (Forster 1990b; Monsell 1991). These frequency effects have a specific functional form: since Howes and Solomon (1951), it is accepted that lexical access can be well approximated as a log-function of frequency.

Modeling lexical access in terms of log-frequency provides a good, but not perfect, fit to the data. Murray and Forster (2004) studied the role of frequency in detail and identified various issues with the log-frequency model. The data consisted of

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_7

Table 7.1 Frequency bands of words used in Murray and Forster (2004) (Exp. 1); frequency reported in number of tokens per 1 million words

Group	Frequency range	Mean frequency	Latency (ms)	Accuracy (%)	Example word
1	315–197	242.0	542	97.22	Guy
2	100–85	92.8	555	95.56	Somebody
3	60–55	57.7	566	95.56	Extend
4	42–39	40.5	562	96.3	Dance
5	32–30	30.6	570	96.11	Shape
6	24–23	23.4	569	94.26	Besides
7	19	19.0	577	95	Fit
8	16	16.0	587	92.41	Dedicate
9	14–13	13.4	592	91.67	Robot
10	12–11	11.5	605	93.52	Tile
11	10	10.0	603	91.85	Between
12	9	9.0	575	93.52	Precedent
13	7	7.0	620	91.48	Wrestle
14	5	5.0	607	90.93	Resonate
15	3	3.0	622	84.44	Seated
16	1	1.0	674	74.63	Habitually

collected responses and response times in a lexical decision task using words from 16 frequency bands, summarized in Table 7.1.[1]

Using the RT latencies from Murray and Forster (2004), let us build a log-frequency model and evaluate the discrepancies between the predictions of the model and the data. We first store the data in two variables `freq` (mean frequency) and `rt` (reaction time/latency; measured in s) (Fig. 7.1).

[py25]
```
>>> import matplotlib as mpl
>>> mpl.use("pgf")
>>> pgf_with_pdflatex = {"text.usetex": True, "pgf.texsystem": "pdflatex",
...                      "pgf.preamble": [r"\usepackage{mathpazo}",
...                                       r"\usepackage[utf8x]{inputenc}",
...                                       r"\usepackage[T1]{fontenc}",
...                                       r"\usepackage{amsmath}"],
...                      "axes.labelsize": 8,
...                      "font.family": "serif",
...                      "font.serif":["Palatino"],
...                      "font.size": 8,
...                      "legend.fontsize": 8,
...                      "xtick.labelsize": 8,
...                      "ytick.labelsize": 8}
```

[1]Example words in Table 7.1 are based on the *Corpus of Contemporary American English* (COCA; http://corpus.byu.edu/coca/), specifically the list available at http://www.wordfrequency.info/files/entriesWithoutCollocates.txt, which lists frequencies of words of 450 million words total (as of March 7, 2017). The chosen example word was one of the closest one to the mean frequency listed in the same row. These were not the words used in the actual experiment—Murray and Forster (2004) controlled for other parameters, e.g., word length while manipulating word frequency.

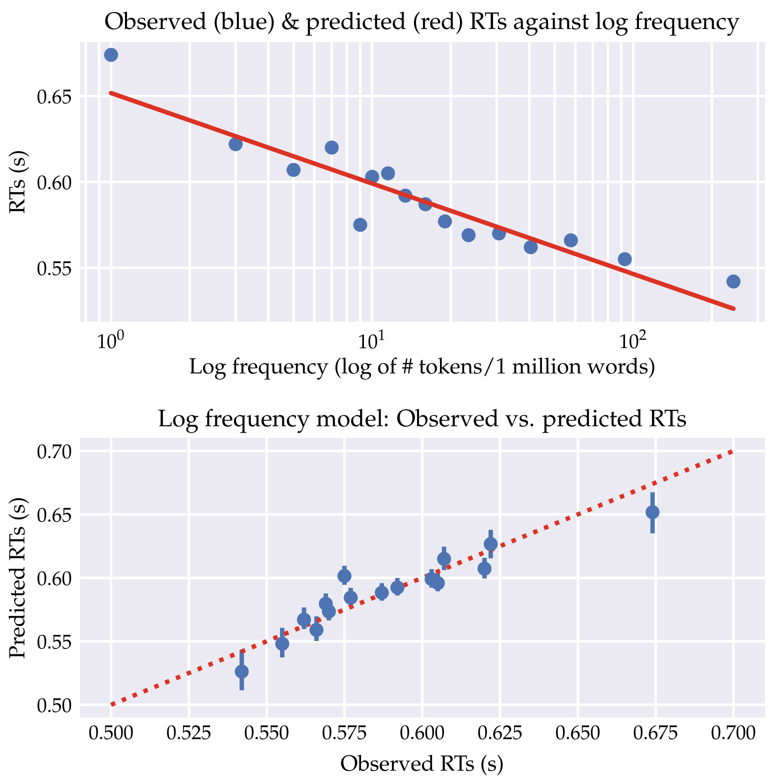


Fig. 7.1 Log-frequency model estimates and observed RTs

```
>>> mpl.rcParams.update(pgf_with_pdflatex)
>>> import matplotlib.pyplot as plt
>>> plt.style.use('seaborn')
>>> import seaborn as sns
>>> sns.set_style({"font.family":"serif", "font.serif":["Palatino"]})
>>> import numpy as np
>>> import pandas as pd
>>> import pymc3 as pm
>>> from pymc3.backends import Text
>>> from pymc3.backends.text import load

>>> freq = np.array([242, 92.8, 57.7, 40.5, 30.6, 23.4, 19,
...                   16, 13.4, 11.5, 10, 9, 7, 5, 3, 1])
>>> rt = np.array([542, 555, 566, 562, 570, 569, 577, 587,
...                 592, 605, 603, 575, 620, 607, 622, 674])/1000
>>> accuracy = np.array([97.22, 95.56, 95.56, 96.3, 96.11, 94.26,
...                       95, 92.41, 91.67, 93.52, 91.85, 93.52,
...                       91.48, 90.93, 84.44, 74.63])/100
```

We can now build a Bayesian model. We are thoroughly familiar with this kind of code, so we include it in [py26] below without any further comments:

[py26]
```
>>> log_freq_model = pm.Model()
>>> with log_freq_model:
...     # priors
...     intercept = pm.Normal('intercept', mu=0, sd=300)
...     slope = pm.Normal('slope', mu=0, sd=300)
...     sigma = pm.HalfNormal('sigma', sd=300)
...     # likelihood
...     mu = pm.Deterministic('mu', intercept + slope*np.log(freq))
...     observed_rt = pm.Normal('observed_rt', mu=mu, sd=sigma,
...                             observed=rt)
...
>>> #with log_freq_model:
>>>     #db = Text('./data/log_freq_model_trace')
>>>     #trace = pm.sample(draws=5000, trace=db, tune=15000,
>>>                       #n_init=200000, njobs=4)

>>> with log_freq_model:
...     trace = load('./data/log_freq_model_trace')
...
```

We can now plot the estimates of the log-frequency model:

[py27]

```
>>> mu = trace["mu"]
>>> def generate_log_freq_figure():
...     fig, (ax1, ax2) = plt.subplots(ncols=1, nrows=2)
...     fig.set_size_inches(5.5, 5.5)
...     # plot 1
...     ax1.plot(freq, rt, marker='o', linestyle='')
...     ax1.plot(freq, mu.mean(axis=0), color='red', linestyle='-')
...     ax1.set_title('Observed (blue) \& predicted (red) RTs\
...                    against log frequency')
...     ax1.set_xlabel('Log frequency (log of \# tokens/1 million words)')
...     ax1.set_xscale('log', basex=10)
...     ax1.set_ylabel('RTs (s)')
...     ax1.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # plot 2
...     yerr=[mu.mean(axis=0)-pm.hpd(mu)[:,0],
...           pm.hpd(mu)[:,1]-mu.mean(axis=0)]
...     ax2.errorbar(rt, mu.mean(axis=0), yerr=yerr,
...                  marker='o', linestyle='')
...     ax2.plot(np.linspace(0.5, 0.7, 10), np.linspace(0.5, 0.7, 10),
...              color='red', linestyle=':')
...     ax2.set_title('Log frequency model: Observed vs. predicted RTs')
...     ax2.set_xlabel('Observed RTs (s)')
...     ax2.set_ylabel('Predicted RTs (s)')
...     ax2.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # clean up and save
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=1.9)
...     plt.savefig('./figures/log_freq_model_figure.eps')
...     plt.savefig('./figures/log_freq_model_figure.png')
...     plt.savefig('./figures/log_freq_model_figure.pdf')
...
>>> generate_log_freq_figure()
```

The plots show that the log-frequency model gets the middle values right, but it tends to underestimate the amount of time needed to access words in the extreme frequency bands—both low frequency (associated with high RTs) and high frequency (associated with low RTs). Murray and Forster (2004) take this as an argument for a specific information retrieval mechanism, the Rank Hypothesis (see Forster 1976, 1992), but as they note, other models of retrieval could similarly improve data fit. One such model treats frequency effects as practiced memory retrieval, which is commonly assumed to be a power function of time in the same way that memory performance is (Newell and Rosenbloom 1981; Anderson 1982; Logan 1990).

7.2 The Simplest ACT-R Model of Lexical Decision

Practiced memory retrieval in ACT-R crucially relies on the power-function model of declarative memory. The power function is used to compute (base) activation based on the number of practice trials/'rehearsals' of a word (see (5) in Chap. 6), which in turn is used to compute latency and accuracy for retrieval processes (see (25) and (24) in Chap. 6).

For any word, the number of rehearsals that contribute to its base activation are crucially determined by its frequency. There are other factors that determine the number and timing of the rehearsals, but we will assume a simple model here: the number of rehearsals is exclusively determined by frequency. We will also assume, for simplicity, that presentations of a word are linearly spaced in time.

To be specific, let's consider a 15-year old speaker. How can we estimate the time points at which a word was used in language interactions that the speaker participated in? Once we know these time points, we can compute the base activation for that word, which in turn will make predictions about retrieval latency and retrieval accuracy that we can check against the Murray and Forster (2004) data in Table 7.1.

We know the lifetime of the speaker (15 years), so if we know the total number of words an average 15-year old speaker has been exposed to, we can easily calculate how many times a particular word was used on average, based on its frequency. Once we find out how many times a word with a specific frequency was presented to our speaker during their lifetime, we can then present the word at linearly spaced intervals during the life span of the speaker (we use linearly spaced intervals for simplicity).

A good approximation of the number of words a speaker is exposed to per year can be found in Hart and Risley (1995). Based on recordings of 42 families, Hart and Risley estimate that children comprehend between 10 million and 35 million words a year, depending to a large extent on the social class of the family. This amount increases linearly with age.

According to the Hart and Risley (1995) study, a 15-year old has been exposed to anywhere between 50 and 175 million words total. For simplicity, let's use the mean of 112.5 million words as the total amount of words a 15-year old speaker has been exposed to. This is a very conservative estimate because we ignore production, as well as the linguistic exposure associated with mass media.

The roughness of our estimate is not an issue for our purposes since we are interested in the *relative* effect of frequency, not its *absolute* effect. We do not want to predict how much time the retrieval of a word from one frequency band requires, but how much time a word requires compared to a word from another frequency band.

In **[py28]** below, we first compute the number of seconds in a year, and then the total number of seconds in the life span of the 15-year old speaker we're modeling (lines 1–2). The function `time_freq` defined on lines 3–9 takes the mean frequency vector `freq` in **[py25]** above and generates a schedule of linearly spaced word rehearsals/presentations for words from the 16 frequency bands studied by Murray and Forster (2004). The schedule of rehearsals covers the entire life span of our 15-year old speaker.

[py28]
```
>>> SEC_IN_YEAR = 365*24*3600                                           1
>>> SEC_IN_TIME = 15*SEC_IN_YEAR                                        2
>>> def time_freq(freq):                                                3
...     max_idx = np.int(np.max(freq) * 112.5)                          4
...     rehearsals = np.zeros((max_idx, len(freq)))                     5
...     for i in np.arange(len(freq)):                                  6
...         temp = np.arange(np.int((freq[i]*112.5)))                   7
...         temp = temp * np.int(SEC_IN_TIME/(freq[i]*112.5))           8
...         rehearsals[:len(temp),i] = temp                             9
...     return(rehearsals.T)                                            10
...                                                                     11
```

On line 4 in [py28], we initialize our rehearsal schedule in the matrix `rehearsals`. This matrix has as many rows as the number of rehearsals for the most frequent word band: `np.max(freq)` gives us the maximum frequency in words per million, which we multiply by 112.5 million words (the total number of words our 15-year old speaker has been exposed to). The `rehearsals` matrix has 16 columns: as many columns as the frequency bands we are interested in.

The `for` loop on lines 6–9 in [py28] iterates over the 16 frequency bands and, for each frequency band, it does the following. On line 7, we identify the total number of rehearsals for frequency band *i* throughout the life span of the speaker (`freq[i]*112.5`) and generate a vector with as many positions as there are rehearsals. On line 8, at each position in that vector, we store the time of a rehearsal in seconds. The result is a vector `temp` of linearly spaced rehearsal times that we store in our full `rehearsals` matrix (line 9).

These rehearsal times can also be viewed as time periods since rehearsals if we reverse the vector (recall that we need time periods since rehearsals when we compute base activation). But we don't need to actually reverse the vector since we will have to sum the time periods to compute activation, and summation is commutative.

Finally, we return the full `rehearsals` matrix on line 10, in transposed form because of the way we will use it to compute base activation (see below).

With this function in hand, we compute a rehearsal schedule for all 16 frequency bands on line 3 of [py29] below. We store the matrix in a `theano` variable called `time`. The `theano` library, which we import on lines 1–2, enables us to do computations with multi-dimensional arrays efficiently, and provides the computational backbone for the Bayesian modeling library `pymc3`. We need to access it directly to be able to compute activations from the rehearsal schedule stored in the `time` variable.

[py29]
```
>>> import theano                                                       1
>>> import theano.tensor as tt                                          2
>>> time = theano.shared(time_freq(freq), 'time')                       3
```

Now that we have the rehearsal schedule, we can implement the lexical decision model:

[py30]
```
>>> lex_dec_model = pm.Model()                                          1
>>> with lex_dec_model:                                                 2
...     # prior for base activation                                     3
...     decay = pm.Uniform('decay', lower=0, upper=1)                   4
...     # priors for latency                                               5
...     intercept = pm.Uniform('intercept', lower=0, upper=2)              6
...     latency_factor = pm.Uniform('latency_factor', lower=0, upper=5)    7
...     # priors for accuracy                                              8
...     noise = pm.Uniform('noise', lower=0, upper=5)                      9
...     threshold = pm.Normal('threshold', mu=0, sd=10)                   10
...     # compute activation                                              11
...     scaled_time = time ** (-decay)                                    12
...     def compute_activation(scaled_time_vector):                       13
...         compare = tt.isinf(scaled_time_vector)                        14
...         subvector = scaled_time_vector[(1-compare).nonzero()]         15
...         activation_from_time = tt.log(subvector.sum())                16
...         return activation_from_time                                   17
...     activation_from_time, _ = theano.scan(fn=compute_activation,      18
...                                           sequences=scaled_time)      19
...     # latency likelihood                                              20
...     mu_rt = pm.Deterministic('mu_rt', intercept +\                    21
...                         latency_factor*tt.exp(-activation_from_time)) 22
...     rt_observed = pm.Normal('rt_observed', mu=mu_rt, sd=0.01,         23
...                           observed=rt)                                24
...     # accuracy likelihood                                             25
...     odds_reciprocal = tt.exp(-(activation_from_time - threshold)/noise) 26
...     mu_prob = pm.Deterministic('mu_prob', 1/(1 + odds_reciprocal))    27
...     prob_observed = pm.Normal('prob_observed', mu=mu_prob, sd=0.01,   28
...                             observed=accuracy)                        29
...                                                                       30
>>> #with lex_dec_model:                                                  31
>>>     #db = Text('./data/lex_dec_model_trace')                          32
>>>     #trace = pm.sample(draws=10000, trace=db,                         33
>>>                      #n_init=200000, njobs=6)                         34
                                                                          35
>>> with lex_dec_model:                                                   36
...     trace = load('./data/lex_dec_model_trace')                        37
...                                                                       38
```

Computing activations from the rehearsal schedule requires us to define a separate function `compute_activation`—see lines 13–17 in [py30]. This function assumes that the matrix `scaled_time` has been computed (line 12): to compute this matrix, we take our rehearsal schedule stored in the `time` matrix (time periods since rehearsals for all frequency bands) and raise these time periods to the `-decay` power. The result is a matrix that stores scaled times, i.e., the base activation boost contributed by each individual word rehearsal for all frequency bands.

Some of the values in the `time` matrix were 0. In the `scaled_time` matrix, they become infinity. When we compute final activations, we want to discard all these infinity values, which is what the `compute_activation` function does. It takes the 16 vectors of scaled times (for the 16 frequency bands) as inputs one at a time. Then, it identifies all the infinity values (line 14). Then, it extracts the subvector of the input vector that contains only non-infinity values (line 15). With this subvector in hand, we can sum the scaled times and take the log of the result to obtain our final activation value (line 16), which the function returns to us.

With the function `compute_activation` in hand, we need to iterate over the 16 vectors of scaled times for our 16 frequency bands and compute the 16 resulting activations by applying the `compute_activation` function to each of these vectors of scaled times. However, `theano`-based programming is purely functional, which means there are no `for` loops. We therefore use the `theano.scan` method on lines 18–19 of **[py30]** to iteratively apply the `compute_activation` function to each of the 16 vectors in the `scaled_time` matrix.[2]

The likelihoods of the lexical decision model in **[py30]** (lines 21–24 and 26–29 in **[py30]**) are direct implementations of the retrieval latency and retrieval probability equations in (25) and (24). We omit the latency exponent in the latency likelihood (see `mu_rt` on lines 21–22) because we assume it is set to its default value of 1. We will see that this value is not appropriate, so we will have to move to a model in which the latency exponent is also fully modeled.

Note that the dispersions around the mean RTs and mean probabilities are very minimal—we set the standard deviations on lines 23 and 28 to 0.01. The reason is that our observed values for both RTs and accuracy are not raw values—they are already means, namely, the empirical means for the 16 frequency bands reported in Murray and Forster (2004). As such, we assume these means are very precise reflections of the underlying parameter values.

We could model these standard deviations explicitly, but we decided not to since we have only 32 observations here (16 for RTs, 16 for accuracies), and we are trying to estimate a fairly large number of parameters already: `decay`, `intercept`, `latency_factor`, `noise` and `threshold`. Low information priors for these parameters are specified on lines 4, 6–7 and 9–10 in **[py30]**.[3]

The only new parameter in this model relative to the ACT-R probability and latency equations in (24) and (25) is the `intercept` parameter we use in the latency likelihood (line 21 in **[py30]**). The intercept is supposed to absorb the time in the lexical decision task associated with operations other than memory retrieval: focusing visual attention, motor planning etc.

With the model fully specified, we sample from the posterior distributions of the parameters. Once we obtain the samples, we are ready to plot them to evaluate how well the model fits the data (we take the first 2000 samples to be the burn-in and drop them; see, for example, Kruschke (2011) for more discussion of burn-in, thinning etc.). The code for the plots is provided in **[py31]** and the resulting plots are shown in Fig. 7.2.

[py31]
```
>>> trace = trace[2000:]
>>> mu_rt = pd.DataFrame(trace['mu_rt'])
>>> yerr_rt = [(mu_rt.mean()-mu_rt.quantile(0.025))*1000,
...            (mu_rt.quantile(0.975)-mu_rt.mean())*1000]

>>> mu_prob = pd.DataFrame(trace['mu_prob'])
>>> yerr_prob = [(mu_prob.mean()-mu_prob.quantile(0.025)),
...              (mu_prob.quantile(0.975)-mu_prob.mean())]

>>> def generate_lex_dec_model_figure():
```

[2]For more information about functional programming, see, for example, https://en.wikipedia.org/wiki/Functional_programming and Abelson et al. (1996). Purely functional programming languages (Haskell is probably the most well-known example nowadays) should be easy to understand for formal semanticists, given their familiarity with λ-calculus.

[3]By low information priors, we mean priors that do not assign high probability to particular narrow regions in the parameter space. If some regions are less probable than others, that is because their lower probability can be determined from considerations that are independent of experimental data we are trying to model.

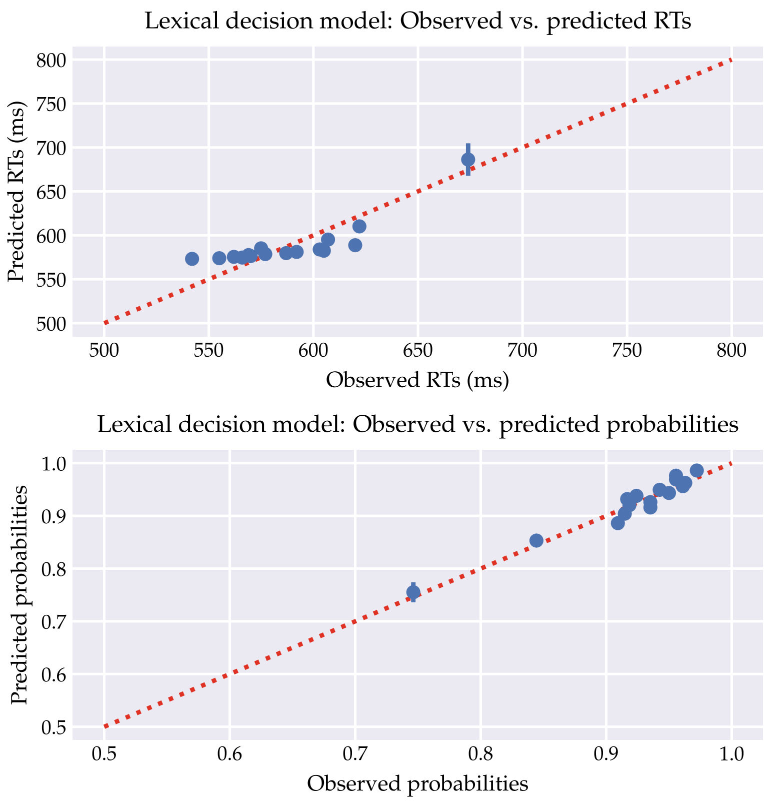


Fig. 7.2 Lexical decision model: estimated and observed RTs and probabilities

```
...     fig, (ax1, ax2) = plt.subplots(ncols=1, nrows=2)
...     fig.set_size_inches(5.5, 5.5)
...     # plot 1: RTs
...     ax1.errorbar(rt*1000, mu_rt.mean()*1000, yerr=yerr_rt, marker='o',
...                  linestyle='')
...     ax1.plot(np.linspace(500, 800, 10), np.linspace(500, 800, 10),
...              color='red', linestyle=':')
...     ax1.set_title('Lexical decision model:\
...                   Observed vs. predicted RTs')
...     ax1.set_xlabel('Observed RTs (ms)')
...     ax1.set_ylabel('Predicted RTs (ms)')
...     ax1.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # plot 2: probabilities
...     ax2.errorbar(accuracy, mu_prob.mean(), yerr=yerr_prob, marker='o',
...                  linestyle='')
...     ax2.plot(np.linspace(50, 100, 10)/100,\
...                         np.linspace(50, 100, 10)/100,
...                         color='red', linestyle=':')
...     ax2.set_title('Lexical decision model:\
...                   Observed vs. predicted probabilities')
...     ax2.set_xlabel('Observed probabilities')
...     ax2.set_ylabel('Predicted probabilities')
...     ax2.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # clean up and save
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/lex_dec_model_figure.eps')
...     plt.savefig('./figures/lex_dec_model_figure.png')
...     plt.savefig('./figures/lex_dec_model_figure.pdf')
...
>>> generate_lex_dec_model_figure()
```

An important thing to note about the ACT-R lexical decision model is that predictions about latencies and probabilities are theoretically connected: base activation is an essential ingredient in predicting both of them. Thus, we are not proceeding purely in an inductive fashion here by looking at the RT data on one hand, the accuracy data on the other hand, and then drawing theoretical conclusions from the data in an informal way, i.e., in a way that is suggestive and possibly productive, but ultimately vague and incapable of making precise predictions.

Instead, our mathematical model takes a core theoretical concept (base activation as a function of word frequency) and connects it in a mathematically explicit way to latency and accuracy. Furthermore, our computational model directly implements the mathematical model, and enables us to fit it to the experimentally obtained latency and accuracy data.

In addition to connecting distinct kinds of observable behavior via the same unobservable theoretical construct(s), a hallmark of a good scientific theory is that it is falsifiable. And the plots in Fig. 7.2 show that an ACT-R model of lexical decision that sets the latency exponent to its default value of 1 (in effect omitting it) is empirically inadequate.

The bottom plot in Fig. 7.2 shows that our lexical decision model does a good job of modeling retrieval probabilities. The predicted probabilities are very close to the observed ones, and they are precisely estimated (there are very few visible error bars protruding out of the plotted points).

In contrast, latencies are poorly modeled, as the top plot in Fig. 7.2 shows. The predicted RTs are not very close to the observed RTs, and our model is very confident in its incorrect predictions (error bars are barely visible for most predicted RTs).

7.3 The Second ACT-R Model of Lexical Decision: Adding the Latency Exponent

Our ACT-R lexical decision model without a latency exponent does not provide a satisfactory fit to the Murray and Forster (2004) latency data. In fact, the log-frequency model is both simpler (although less theoretically motivated) and empirically more adequate.

We therefore move to a model that is minimally enriched by explicitly modeling the latency exponent. The usefulness of the latency exponent in modeling reaction time data has been independently noted in the recent literature—see, for example, West et al. (2010).

The code for the model is provided in **[py32]** below. The only additions are (i) the half-normal prior for the latency exponent on line 10 and (ii) its corresponding addition to the latency likelihood on line 25. Note that we use a different method to sample the posterior for this model (Sequential Monte Carlo/SMC, line 37).[4]

[py32]
```
 1  >>> from pymc3.backends.text import dump
 2
 3  >>> lex_dec_model_lat_exp = pm.Model()
 4  >>> with lex_dec_model_lat_exp:
 5  ...     # prior for base activation
 6  ...     decay = pm.Uniform('decay', lower=0, upper=1)
 7  ...     # priors for latency
 8  ...     intercept = pm.Uniform('intercept', lower=0, upper=2)
 9  ...     latency_factor = pm.Uniform('latency_factor', lower=0, upper=5)
10  ...     latency_exponent = pm.HalfNormal('latency_exponent', sd=3)
11  ...     # priors for accuracy
12  ...     noise = pm.Uniform('noise', lower=0, upper=5)
13  ...     threshold = pm.Normal('threshold', mu=0, sd=10)
14  ...     # compute activation
15  ...     scaled_time = time ** (-decay)
16  ...     def compute_activation(scaled_time_vector):
17  ...         compare = tt.isinf(scaled_time_vector)
18  ...         subvector = scaled_time_vector[(1-compare).nonzero()]
19  ...         activation_from_time = tt.log(subvector.sum())
20  ...         return activation_from_time
21  ...     activation_from_time, _ = theano.scan(fn=compute_activation,
22  ...                                           sequences=scaled_time)
23  ...     # latency likelihood
24  ...     mu_rt = pm.Deterministic('mu_rt', intercept +\
25  ...                              latency_factor*tt.exp(-latency_exponent*\
26  ...                              activation_from_time))
27  ...     rt_observed = pm.Normal('rt_observed', mu=mu_rt, sd=0.01,
28  ...                             observed=rt)
29  ...     # accuracy likelihood
30  ...     odds_reciprocal = tt.exp(-(activation_from_time - threshold)/noise)
31  ...     mu_prob = pm.Deterministic('mu_prob', 1/(1 + odds_reciprocal))
32  ...     prob_observed = pm.Normal('prob_observed', mu=mu_prob, sd=0.01,
33  ...                               observed=accuracy)
34  ...
35  >>> # run 4 times to obtain 4 chains
36  >>> #with lex_dec_model_lat_exp:
37  >>>     #step = pm.SMC()
38  >>>     #trace = pm.sample(draws=20000, step=step, njobs=1)
39
40  >>> #dump = Text('./data/lex_dec_model_lat_exp_trace', trace)
41
42  >>> with lex_dec_model_lat_exp:
43  ...     trace = load('./data/lex_dec_model_lat_exp_trace')
44  ...
```

We plot the results of this enriched lexical decision model: the code for the plots is provided in [py33] and the resulting plots are shown in Fig. 7.3.

[py33]
```
>>> mu_rt = pd.DataFrame(trace['mu_rt'])
>>> yerr_rt = [(mu_rt.mean()-mu_rt.quantile(0.025))*1000,
...            (mu_rt.quantile(0.975)-mu_rt.mean())*1000]

>>> mu_prob = pd.DataFrame(trace['mu_prob'])
>>> yerr_prob = [(mu_prob.mean()-mu_prob.quantile(0.025)),
...              (mu_prob.quantile(0.975)-mu_prob.mean())]

>>> def generate_lex_dec_model_lat_exp_figure():
```

[4]Discussing different kinds of MCMC sampling schemes, chain convergence diagnostics etc. is beyond the scope of this book. For a fairly recent survey of SMC methods, see, for example, Creal (2012). For a brief comparison of BUGS versus NUTS, see, for example, this blog post: http://elevanth.org/blog/2017/11/28/build-a-better-markov-chain/. For more discussion of MCMC diagnostics etc., see Gelman and Hill (2007), Kruschke (2011) and references therein, among others.

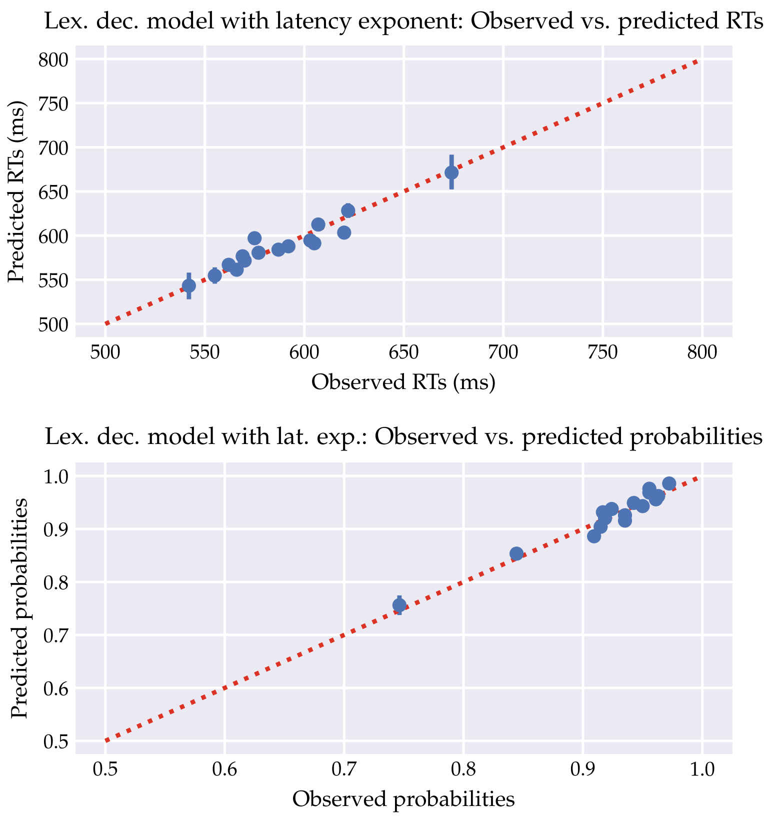


Fig. 7.3 Lex. dec. model with latency exp.: estimated and observed RTs and probabilities

```
...     fig, (ax1, ax2) = plt.subplots(ncols=1, nrows=2)
...     fig.set_size_inches(5.5, 5.5)
...     # plot 1: RTs
...     ax1.errorbar(rt*1000, mu_rt.mean()*1000, yerr=yerr_rt, marker='o',
                        linestyle='')
...     ax1.plot(np.linspace(500, 800, 10), np.linspace(500, 800, 10),
                 color='red', linestyle=':')
...     ax1.set_title('Lex. dec. model with latency exponent:\
...                    Observed vs. predicted RTs')
...     ax1.set_xlabel('Observed RTs (ms)')
...     ax1.set_ylabel('Predicted RTs (ms)')
...     ax1.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # plot 2: probabilities
...     ax2.errorbar(accuracy, mu_prob.mean(), yerr=yerr_prob, marker='o',
...                  linestyle='')
...     ax2.plot(np.linspace(50, 100, 10)/100,
...              np.linspace(50, 100, 10)/100,
...              color='red', linestyle=':')
...     ax2.set_title('Lex. dec. model with lat. exp.:\
...                    Observed vs. predicted probabilities')
...     ax2.set_xlabel('Observed probabilities')
...     ax2.set_ylabel('Predicted probabilities')
...     ax2.grid(b=True, which='minor', color='w', linewidth=1.0)
...     # clean up and save
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/lex_dec_model_lat_exp_figure.eps')
...     plt.savefig('./figures/lex_dec_model_lat_exp_figure.png')
...     plt.savefig('./figures/lex_dec_model_lat_exp_figure.pdf')
...
>>> generate_lex_dec_model_lat_exp_figure()
```

We see that the lexical decision model that explicitly models the latency exponent fits both latencies and probabilities very well. The latencies, in particular, are modeled better than both the lexical decision model without a latency exponent, and the log-frequency model, which did not have a very good fit to the data at the extreme frequency bands (low or high frequencies).

We list below the estimated posterior mean and 95% credible interval (CRI) for the latency exponent: the posterior mean value and the CRI are pretty far away from the default value of 1 we assumed in the previous model.

[py34]
```
>>> latency_exponent_posterior = trace["latency_exponent"]
>>> latency_exponent_posterior.mean()
0.26874855202916675
>>> pm.hpd(latency_exponent_posterior)
array([0.05284098, 0.46545318])
```

The posterior estimates for the other parameters (means and 95% CRIs) are provided below for reference:

[py35]
```
>>> decay_posterior = trace["decay"]
>>> decay_posterior.mean()
0.20922545157863634
>>> pm.hpd(decay_posterior)
array([6.48482310e-04, 7.12426351e-01])

>>> intercept_posterior = trace["intercept"]
>>> intercept_posterior.mean()
0.48309734409006416
>>> pm.hpd(intercept_posterior)
array([0.37181586, 0.55819152])

>>> latency_factor_posterior = trace["latency_factor"]
>>> latency_factor_posterior.mean()
0.35641063490136776
>>> pm.hpd(latency_factor_posterior)
array([0.00476266, 0.77476657])

>>> threshold_posterior = trace["threshold"]
>>> threshold_posterior.mean()
-1.2335980386761345
>>> pm.hpd(threshold_posterior)
array([-10.80783392,   2.99617913])

>>> noise_posterior = trace["noise"]
>>> noise_posterior.mean()
1.780720441127118
>>> pm.hpd(noise_posterior)
array([1.5927438 , 1.96868986])
```

7.4 Bayes+ACT-R: Quantitative Comparison for Qualitative Theories

In this subsection, we show how we can embed ACT-R models implemented in `pyactr` into Bayesian models implemented in `pymc3`. This embedding opens the way towards doing quantitative comparison based on experimental data for both subsymbolic and symbolic theories.

That is, this Bayes+ACT-R combination enables us to do *quantitative* theory comparison. We will be able to take our symbolic theories that make claims about competence, for example, Discourse Representation Theory (DRT; Kamp 1981; Kamp and Reyle 1993), as we will see in Chaps. 8 and 9, embed them in larger performance theories that have a detailed processing component, and then compare different theories quantitatively based on behavioral data of the kind commonly collected in psycholinguistics.

In this section, we introduce the basics of our Bayes+ACT-R framework by considering and comparing three models for lexical decision tasks:

i. the first model builds on the simple ACT-R/`pyactr` lexical decision model introduced in Chap. 4; we will show how that model can be used as part of the likelihood function of a larger Bayesian model;
ii. the second model is cognitively more realistic than the first one: it makes use of the imaginal buffer as an intermediary between the visual module and the declarative memory module; we set the delay for imaginal-buffer encoding to its default value of 200 ms; once again, this ACT-R/`pyactr` model will provide part of the likelihood component of a larger Bayesian model;
iii. the third and final model is cognitively more realistic since it makes use of the imaginal buffer, just as the second model, but we set the encoding delay of the imaginal buffer to the non-default value of 0 ms[5]; this is the imaginal-buffer delay we needed when we implemented our left-corner parser in Chap. 4; just as before, the ACT-R/`pyactr` model is embedded in a larger Bayesian model, for which it provides part of the likelihood function.

The first model (i) without the imaginal buffer and the other two models (ii) and (iii) with imaginal buffers differ with respect to a symbolic (qualitative) component. In this particular case, the symbolic component (imaginal-buffer usage or lack thereof) belongs to the processing part of the symbolic (non-quantitative) theory, but theoretical differences at the 'core' competence level can (and will) be similarly compared.

In contrast, the last two models (ii) and (iii) differ with respect to specific conjectures about a subsymbolic (quantitative) component, namely the average time to encode chunks in the imaginal buffer.

[5]In computational models of psychological experiments, the encoding delay in the imaginal buffer is usually set to 200 ms. See Chap. 3, and also Taatgen et al. (2009).

Our Bayes+ACT-R framework enables us to compare all these models. This comparison is not only qualitative. The models can be quantitatively evaluated and compared relative to specific experimental data. Since `pyactr` enables us to embed ACT-R models as the likelihood component of larger Bayesian models built with `pymc3`, we can do statistical inference over the subsymbolic parameters of our ACT-R model in the standard Bayesian way, rather than trying different values one at a time and manually identifying the best fitting ones.

We'll therefore be able to identify the standard measures of central tendency (posterior means, but also medians or modes if needed), as well as compute credible intervals for every parameter of interest. The Bayesian framework will furthermore enable us to conduct unrestricted model comparison (using Bayes factors or WAIC, for example), unlike maximum likelihood methods—see the discussion at the end of this section and in Sect. 7.5.

Throughout this book, whenever we embed an ACT-R model in a Bayesian model, we turn off all the non-deterministic (stochastic) components of the ACT-R model other than the ones for which we are estimating parameters. This effectively turns the ACT-R model into a complex, but deterministic, function of the parameters, which we can straightforwardly incorporate as a component of the likelihood function of the Bayesian model.

For more realistic simulations, we would have to turn on various non-deterministic components of the ACT-R model (e.g., noise associated with visual module), in which case we would have to resort to Approximate Bayesian Computation (ABC; see Sisson et al. (2019) and references therein) to incorporate an approximation of the ACT-R induced likelihood into our Bayesian model. ABC is beyond the scope of this book, but it is a very promising direction for future research, and a central issue to be addressed as more linguistically sophisticated cognitive models are developed.

7.4.1 The Bayes+ACT-R Lexical Decision Model Without the Imaginal Buffer

The link to the full code for this model is provided in the appendix to this chapter—see Sect. 7.7.1. We will only discuss here the most important and novel aspects of this Bayes+ACT-R model combination. We first initialize the model under the variable `lex_decision` and declare its goal buffer to be `g`.

[py36]
```
>>> import pyactr as actr
>>> environment = actr.Environment(focus_position=(320, 180))
>>> lex_decision = actr.ACTRModel(environment=environment,
...                               subsymbolic=True,
...                               automatic_visual_search=True,
...                               activation_trace=False,
...                               retrieval_threshold=-80,
...                               motor_prepared=True,
...                               eye_mvt_scaling_parameter=0.18,
...                               emma_noise=False)
>>> lex_decision.goals = {}
>>> lex_decision.set_goal("g")
set()
```

We set up the data: see the FREQ, RT and ACCURACY variables in [py37] below (recall that pyactr measures time in s, not ms, so we divide the RTs by 1000). We then generate the presentations times for the 16 word-frequency bands considered in Murray and Forster (2004): see FREQ_DICT and the theano variable time.

[py37]
```
>>> FREQ = np.array([242, 92.8, 57.7, 40.5, 30.6, 23.4, 19,
...                  16, 13.4, 11.5, 10, 9, 7, 5, 3, 1])
>>> RT = np.array([542, 555, 566, 562, 570, 569, 577, 587,
...                592, 605, 603, 575, 620, 607, 622, 674])/1000
>>> ACCURACY = np.array([97.22, 95.56, 95.56, 96.3, 96.11, 94.26,
...                      95, 92.41, 91.67, 93.52, 91.85, 93.52,
...                      91.48, 90.93, 84.44, 74.63])/100

>>> # on average, 15 years of exposure is 112.5 million words
>>> FREQ_DICT = {}
>>> FREQ_DICT['guy'] = 242*112.5
>>> FREQ_DICT['somebody'] = 92*112.5
>>> FREQ_DICT['extend'] = 58*112.5
>>> FREQ_DICT['dance'] = 40.5*112.5
>>> FREQ_DICT['shape'] = 30.6*112.5
>>> FREQ_DICT['besides'] = 23.4*112.5
>>> FREQ_DICT['fit'] = 19*112.5
>>> FREQ_DICT['dedicate'] = 16*112.5
>>> FREQ_DICT['robot'] = 13.4*112.5
>>> FREQ_DICT['tile'] = 11.5*112.5
>>> FREQ_DICT['between'] = 10*112.5
>>> FREQ_DICT['precedent'] = 9*112.5
>>> FREQ_DICT['wrestle'] = 7*112.5
>>> FREQ_DICT['resonate'] = 5*112.5
>>> FREQ_DICT['seated'] = 3*112.5
>>> FREQ_DICT['habitually'] = 1*112.5

>>> ORDERED_FREQ = sorted(list(FREQ_DICT), key=lambda x:FREQ_DICT[x],\
...                       reverse=True)

>>> def time_freq(freq):
...     rehearsals = np.zeros((np.int(np.max(freq) * 113), len(freq)))
...     for i in np.arange(len(freq)):
...         temp = np.arange(np.int((freq[i]*112.5)))
...         temp = temp * np.int(SEC_IN_TIME/(freq[i]*112.5))
...         rehearsals[:len(temp),i] = temp
...     return(rehearsals.T)
...
>>> time = theano.shared(time_freq(FREQ), 'time')
>>> LEMMA_CHUNKS = [(actr.makechunk("", typename="word", form=word))
...                 for word in ORDERED_FREQ]
```

We are now ready to build the procedural core of our model. The production rules are the same as the ones we introduced and discussed in Chap. 4, listed for ease of reference in [py38] below:

- the "attend word" rule takes a visual location encoded in the visual *where* buffer and issues a command to the visual *what* buffer to move attention to that visual location;
- the "retrieving" rule takes the visual value/content discovered at that visual location, which is a potential word form, and places a declarative memory request to retrieve a word with that form;
- finally, the "lexeme retrieved" and "no lexeme found" rules take care of the two possible outcomes of the memory retrieval request: if a word with that form is retrieved from memory ("lexeme retrieved"), a command is issued to the motor module to press the 'J' key; if no word is retrieved ("no

`lexeme found"`), a command is issued to the motor module to press the `'F'` key.

[py38]
```
>>> lex_decision.productionstring(name="attend word", string="""
...     =g>
...     isa     goal
...     state   attend
...     =visual_location>
...     isa    _visuallocation
...     ?visual>
...     state   free
...     ==>
...     =g>
...     isa     goal
...     state   retrieving
...     +visual>
...     isa     _visual
...     cmd     move_attention
...     screen_pos =visual_location
...     ~visual_location>
... """)
{'=g': goal(state= attend), '=visual_location': _visuallocation(color= ,
screen_x= , screen_y= , value= ), '?visual': {'state': 'free'}}
==>
{'=g': goal(state= retrieving), '+visual': _visual(cmd= move_attention,
color= , screen_pos= =visual_location, value= ), '~visual_location': None}

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

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

>>> lex_decision.productionstring(name="no lexeme found", string="""
...     =g>
...     isa     goal
...     state   retrieval_done
...     ?retrieval>
...     buffer  empty
...     state   error
...     ==>
...     =g>
...     isa     goal
...     state   done
...     +manual>
...     isa     _manual
...     cmd     press_key
...     key     'F'
... """)
{'=g': goal(state= retrieval_done),
'?retrieval': {'buffer': 'empty', 'state': 'error'}}
==>
{'=g': goal(state= done), '+manual': _manual(cmd= press_key, key= F)}
```

With the production rules in place, we can start preparing the way towards embedding the ACT-R model into a Bayesian model. The main idea is that we will use the ACT-R model as the likelihood component of the Bayesian model for lexical-decision latencies/RTs. Specifically, we will feed parameter values for the latency factor `lf`, latency exponent `le` and `decay` into the ACT-R model, run the model with these parameters for words from all 16 frequency bands, and collect the resulting RTs.

The Bayesian model will then use these RTs to sample new values for the `lf`, `le` and `decay` parameters in proportion to how well the RTs generated by the ACT-R model agree with the experimentally collected RTs (and the diffuse priors over these parameters).

The first function we need is `run_stimulus(word)` in **[py39]** below. This function takes a `word` from one of the 16 frequency bands as its argument and runs one instance of the ACT-R lexical decision model for that word. To do that, we first reset the model to its initial state: we flush buffers without moving their contents to declarative memory (lines 2–9 in **[py39]**), we set the `word` argument as the new stimulus (line 10), we initialize the goal buffer `g` with the `"start"` chunk (lines 11–13), and we initialize the lexical decision simulation (lines 14–19).

At this point, we run a `while` loop that steps through the simulation until a lexical decision is made by pressing the `'J'` or `'F'` key, at which point we record the time of the decision in the variable `estimated_time` (set to -1 if the word was not retrieved),[6] exit the `while` loop and return the estimated RT (lines 20–28).

The second function `run_lex_decision_task()` in **[py39]** runs a full lexical decision task by calling the `run_stimulus(word)` function for words from all 16 frequency bands (lines 31–33). The function returns the vector of estimated lexical decision RTs for all these words (line 34).

[6]In this Bayes+ACTR-R estimation, we only model latencies for successful retrievals. In other words, we want the model to always retrieve successfully, and if it does not, we penalize the result by choosing a latency that is markedly off. To ensure that the model performs a successful retrieval, the retrieval threshold in the ACT-R model is set to a very low value ($\tau = -80$, see line 7 in **[py36]**).

[py39]
```
>>> def run_stimulus(word):
...     try:
...         lex_decision.retrieval.pop()
...     except KeyError:
...         pass
...     try:
...         lex_decision.goals["g"].pop()
...     except KeyError:
...         pass
...     stim = {1: {’text’: word, ’position’: (320, 180)}}
...     lex_decision.goals["g"].add(actr.makechunk(nameofchunk=’start’,
...                                            typename="goal",
...                                            state=’attend’))
...     environment.current_focus = [320,180]
...     lex_decision.model_parameters[’motor_prepared’] = True
...     lex_dec_sim = lex_decision.simulation(realtime=False,
...               gui=False, trace=False,
...               environment_process=environment.environment_process,
...               stimuli=stim, triggers=’’, times=10)
...     while True:
...         lex_dec_sim.step()
...         if lex_dec_sim.current_event.action == "KEY PRESSED: J":
...             estimated_time = lex_dec_sim.show_time()
...             break
...         if lex_dec_sim.current_event.action == "KEY PRESSED: F":
...             estimated_time = -1
...             break
...     return estimated_time
...
>>> def run_lex_decision_task():
...     sample = []
...     for word in ORDERED_FREQ:
...         sample.append(run_stimulus(word))
...     return sample
...
```

With the `run_lex_decision_task()` function in hand, we only need to be able to interface the ACT-R model implemented in `pyactr` with a Bayesian model implemented in `pymc3` (and `theano`). This is what the function `actrmodel_latency` in [py40] below does. This function runs the entire lexical decision task for specific values of the latency factor `lf`, latency exponent `le` and `decay` parameters provided by the Bayesian model (which will be discussed below). The activation computed by `theano` with the same value of the `decay` argument is also passed as a separate argument `activation_from_time` to save (a significant amount of) computation time.

The `actrmodel_latency` function takes these four parameter values as arguments (line 3 in [py40]), initializes the lexical decision model with them (lines 4–9), runs the lexical decision task with these model parameters (line 10) and returns the resulting vector of RTs (line 11). The entire function is wrapped inside the `theano`-provided decorator `@as_op` (lines 1–2),[7] which enables `theano` and `pymc3` to use the `actrmodel_latency` function as if it was a native `theano`/`pymc3` function. The only thing the `@as_op` decorator needs is data-type declarations for the arguments of the `actrmodel_latency` function (`lf`, `le` and `decay` are scalars,

[7] `Python3` decorators provide a specific way to define and call higher-order functions, that is, functions that take other functions as arguments and/or return functions as values. Such functions are, of course, very familiar to formal semanticists, who follow Montague in extensively using such functions (usually represented in a simply-typed λ-calculus) to model the semantics of natural languages.

while `activation_from_time` is a vector—line 1) and for its value (which is a vector—line 2).

[py40]
```
>>> @theano.as_op(itypes=[tt.dscalar, tt.dscalar, tt.dscalar, tt.dvector],   1
...               otypes=[tt.dvector])                                         2
... def actrmodel_latency(lf, le, decay, activation_from_time):               3
...     lex_decision.model_parameters["latency_factor"] = lf                  4
...     lex_decision.model_parameters["latency_exponent"] = le                5
...     lex_decision.model_parameters["decay"] = decay                        6
...     activation_dict = {x[0]: x[1] for x in \                              7
...                        zip(LEMMA_CHUNKS, activation_from_time)}           8
...     lex_decision.decmem.activations.update(activation_dict)               9
...     sample = run_lex_decision_task()                                      10
...     return np.array(sample)                                               11
...                                                                           12
```

We are now ready to use the `actrmodel_latency` function as the likelihood function for latencies in a Bayesian model very similar to the ones we already discussed in this chapter. The model is specified in **[py41]** below. The prior for the `decay` parameter is uniform (line 3), the priors for the lexical-decision accuracy parameters `noise` and `threshold` are uniform and normal (lines 4–5), and the priors for the lexical-decision latency parameters `lf` and `le` are both half-normal (lines 6–7).

We then compute activation based on word frequency in the same way we did before (lines 8–15), after which we specify the likelihood function for lexical-decision latency (lines 16–19), which crucially uses the ACT-R model via the `actrmodel_latency` function (line 16), and the likelihood function for lexical-decision accuracy (lines 20–23). Note that the accuracy is computed independently of the latency, which simplifies the workings of the `pyactr` model (as we already indicated, we can assume that the `pyactr` model recalls all words successfully).

[py41]
```
>>> lex_decision_with_bayes = pm.Model()                                      1
>>> with lex_decision_with_bayes:                                             2
...     decay = pm.Uniform('decay', lower=0, upper=1)                         3
...     threshold = pm.Normal('threshold', mu=0, sd=10)                       4
...     noise = pm.Uniform('noise', lower=0, upper=5)                         5
...     lf = pm.HalfNormal('lf', sd=1)                                        6
...     le = pm.HalfNormal('le', sd=1)                                        7
...     scaled_time = time ** (-decay)                                        8
...     def compute_activation(scaled_time_vector):                           9
...         compare = tt.isinf(scaled_time_vector)                            10
...         subvector = scaled_time_vector[(1-compare).nonzero()]             11
...         activation_from_time = tt.log(subvector.sum())                    12
...         return activation_from_time                                       13
...     activation_from_time, _ = theano.scan(fn=compute_activation,          14
...                                           sequences=scaled_time)          15
...     pyactr_rt = actrmodel_latency(lf, le, decay, activation_from_time)    16
...     mu_rt = pm.Deterministic('mu_rt', pyactr_rt)                          17
...     rt_observed = pm.Normal('rt_observed', mu=mu_rt, sd=0.01,             18
...                             observed=RT)                                  19
...     odds_reciprocal = tt.exp(-(activation_from_time - threshold)/noise)   20
...     mu_prob = pm.Deterministic('mu_prob', 1/(1 + odds_reciprocal))        21
...     prob_observed = pm.Normal('prob_observed', mu=mu_prob, sd=0.01,       22
...                               observed=ACCURACY)                          23
...                                                                           24
```

The Bayesian model is schematically represented in Fig. 7.4 (following the type of figures introduced in Kruschke 2011).

The plots in Fig. 7.5 show that the Bayes+ACT-R model without any imaginal-buffer involvement has a very good fit to both the latency and the accuracy data. The

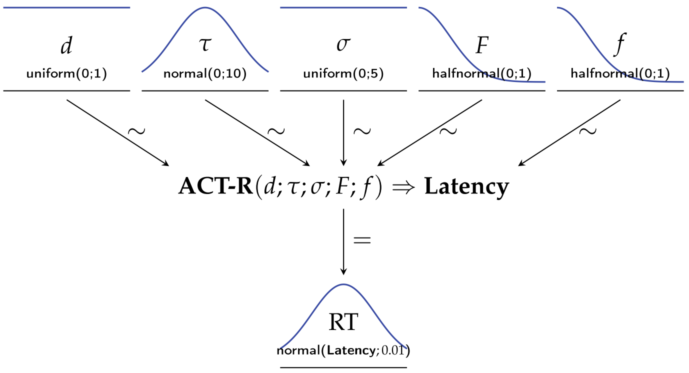


Fig. 7.4 The structure of the Bayesian model in [py41]

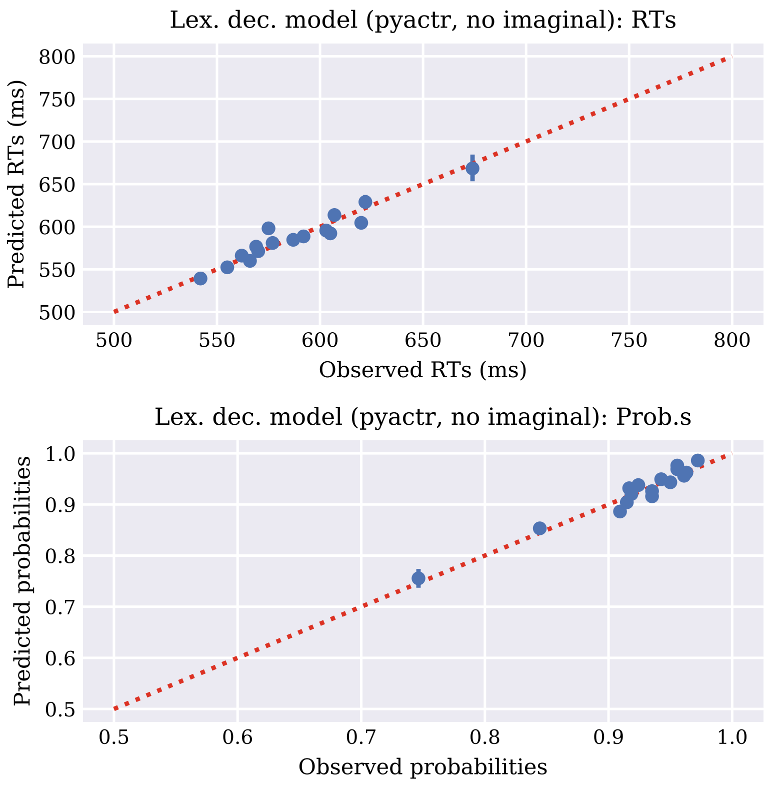


Fig. 7.5 Lex. dec. model, Bayes+ACT-R, no imaginal buffer: estimated and observed RTs and probabilities

top panel plots observed RTs against predicted RTs, while the bottom panel plots observed probabilities against predicted probabilities. The closer the points are to the red diagonal line, the better the model predictions—and we see that the model can fit the observed latency and accuracy data very well.

For reference, we provide the Gelman-Rubin diagnostic (a.k.a. `Rhat`/$\hat{R}$) for this model in **[py42]** below. As Gelman and Hill (2007, 352) put it, "`Rhat` gives information about convergence of the algorithm. At convergence, the numbers [...] should equal 1 [...]. If `Rhat` is less than 1.1 for all parameters, then we judge the algorithm to have approximately converged, in the sense that parallel chains have mixed well." We see that the `Rhat` values for the model are below 1.1, which is reassuring.[8]

[py42]
```
>>> with lex_decision_with_bayes:
...     trace = load('./data/lex_dec_pyactr_no_imaginal')
...
>>> pm.diagnostics.gelman_rubin(trace)
{'threshold': 1.0054899750513782,
'decay': 1.0062475153014665,
'noise': 1.009517446448299,
'lf': 1.0083217341786315,
'le': 1.01606803162613,
'mu_rt': array([1.01458128, 1.01275183, 1.01129197, 1.00973846,
               1.00822621, 1.00639902, 1.00478262, 1.00338619,
               1.0019927 , 1.00094688, 1.00027242, 0.99997675,
               1.00019667, 1.00228782, 1.00666908, 1.01181443]),
'mu_prob': array([1.0111905 , 1.01148533, 1.01152191, 1.01144346,
               1.01127359, 1.01096758, 1.01058891, 1.01014176,
               1.00953423, 1.00884847, 1.00804849, 1.00736052,
               1.00533042, 1.00221617, 0.999931  , 1.00598736])}
```

However, this model oversimplifies the process of encoding visually retrieved data. We assume that the visual value found at a particular visual location is immediately shuttled to the retrieval buffer to place a declarative memory request – see the productions `"attend word"` and `"retrieving"` in **[py38]** above.

This disregards the cognitively-motivated ACT-R assumption that transfers between the visual *what* buffer and the retrieval buffer are mediated by the `goal` or `imaginal` buffers. Cognition in ACT-R is goal-driven, so any important step in a cognitive process should be crucially driven by the chunks stored in the `goal` and/or `imaginal` buffers.

7.4.2 *Bayes+ACT-R Lexical Decision with Imaginal-Buffer Involvement and Default Encoding Delay for the Imaginal Buffer*

We now turn to the examination of the first of two alternative Bayes+ACT-R models, both of which crucially involve the `imaginal` buffer as an intermediary between

[8] For more information about $\hat{R}$, see Gelman et al. (2013, 284–286), and also `pymc3`'s implementation of and help file for the Gelman-Rubin diagnostic. Plots of the chains for the models in this book are available in the `pyactr-book` github repository (see the 'figures' folder, specifically file names that end with '..._trace'), and so are the chains themselves (see the 'data' folder).

the visual *what* and retrieval buffers. The full code for the model discussed in this subsection is available at the link provided in the appendix to this chapter—see Sect. 7.7.1.

The Bayesian model remains the same, the only part we change is the ACT-R likelihood for latencies. Specifically, we modify the procedural core of the ACT-R model as shown in **[py43]** below. We first add the imaginal buffer to the model (line 1 in **[py43]**), and then replace the `"attend word"` and `"retrieving"` rules with three rules `"attend word"` (lines 4–21), `"encoding word"` (lines 28–42) and `"retrieving"` (48–62).

The new rule `"encoding word"` mediates between `"attend word"` and `"retrieving"`. The visual value retrieved by the `"attend word"` rule is shifted into the imaginal buffer by the `"encoding word"` rule. Then, the `"retrieving"` rule takes that value, i.e., word form, from the imaginal buffer and places it into the retrieval buffer.

[py43]
```
>>> lex_decision.set_goal("imaginal")                                          1
set()                                                                          2
                                                                               3
>>> lex_decision.productionstring(name="attend word", string="""               4
...     =g>                                                                    5
...     isa     goal                                                           6
...     state   attend                                                         7
...     =visual_location>                                                      8
...     isa    _visuallocation                                                 9
...     ?visual>                                                               10
...     state   free                                                           11
...     ==>                                                                    12
...     =g>                                                                    13
...     isa     goal                                                           14
...     state   encoding                                                       15
...     +visual>                                                               16
...     isa     _visual                                                        17
...     cmd     move_attention                                                 18
...     screen_pos =visual_location                                            19
...     ~visual_location>                                                      20
... """)                                                                       21
{'=g': goal(state= attend), '=visual_location': _visuallocation(color= ,       22
screen_x= , screen_y= , value= ), '?visual': {'state': 'free'}}                23
==>                                                                            24
{'=g': goal(state= encoding), '+visual': _visual(cmd= move_attention,          25
color= , screen_pos= =visual_location, value= ), '~visual_location': None}     26
                                                                               27
>>> lex_decision.productionstring(name="encoding word", string="""             28
...     =g>                                                                    29
...     isa     goal                                                           30
...     state   encoding                                                       31
...     =visual>                                                               32
...     isa     _visual                                                        33
...     value   =val                                                           34
...     ==>                                                                    35
...     =g>                                                                    36
...     isa     goal                                                           37
...     state   retrieving                                                     38
...     +imaginal>                                                             39
...     isa     word                                                           40
...     form    =val                                                           41
... """)                                                                       42
{'=g': goal(state= encoding), '=visual': _visual(cmd= , color= ,               43
screen_pos= , value= =val)}                                                    44
==>                                                                            45
{'=g': goal(state= retrieving), '+imaginal': word(form= =val)}                 46
                                                                               47
>>> lex_decision.productionstring(name="retrieving", string="""                48
...     =g>
...     isa     goal
...     state   retrieving
...     =imaginal>
...     isa     word
...     form    =val
...     ==>
...     =g>
...     isa     goal
...     state   retrieval_done
...     +retrieval>
...     isa     word
...     form    =val
... """)
{'=g': goal(state= retrieving), '=imaginal': word(form= =val)}
==>
{'=g': goal(state= retrieval_done), '+retrieval': word(form= =val)}
```

These modifications are all symbolic (discrete, non-quantitative) modifications. We will however be able to fit the new model to the same data and quantitatively compare it with the no-imaginal-buffer model discussed in the previous subsection.

The top plot in Fig. 7.6 shows that the model has a very poor fit to the latency data. Adding the imaginal-buffer mediated encoding step adds 200 ms to every lexical decision simulation, since 200 ms is the default ACT-R delay for chunk-encoding into the imaginal buffer.

We therefore see that the predicted latencies for all 16 word-frequency bands are greatly overestimated (they are far above the red diagonal line). The model with the imaginal buffer cannot run faster than about 725 ms, at least not when the imaginal-buffer encoding delay is left at its default 200 ms value.

We can think of the 200 ms imaginal delay as part of the baseline intercept for our ACT-R model. The intercept is simply too high to fit high-frequency words, for which the lexical decision task should take 100 to 200 ms less than this intercept.

The `Rhat` values for this model are once again below 1.1 (in fact, they are very close to 1):

(1)
```
{'threshold': 1.006603925999719,
'decay': 1.0064848072431016,
'noise': 0.9999915488454998,
'lf': 1.0020696744920665,
'le': 1.003927035222827,
'mu_rt': array([1.        , 1.        , 0.99995   , 1.00003341,
                0.99998133, 1.00011225, 1.0004147 , 1.0005439 ,
                1.00137661, 1.00200847, 1.00259929, 1.00214862,
                1.00356952, 1.00571574, 1.00682557, 1.00644145]),
'mu_prob': array([1.00002043, 1.00003799, 1.00005779, 1.00008192,
                1.00011011, 1.0001483 , 1.00018856, 1.0002309 ,
                1.00028581, 1.00034384, 1.00040581, 1.00045968,
                1.00060608, 1.00080852, 1.00091303, 1.00046312])}
```

We see here one of the main benefits of our Bayes+ACT-R framework. We are able to fit any model to experimental data, and we are able to compute quantitative predictions (means and credible intervals) for any model. We are therefore able to quantitatively compare our qualitative theories.

In this particular case, we see that a model that is cognitively more plausible fares more poorly than a simpler, less realistic model.

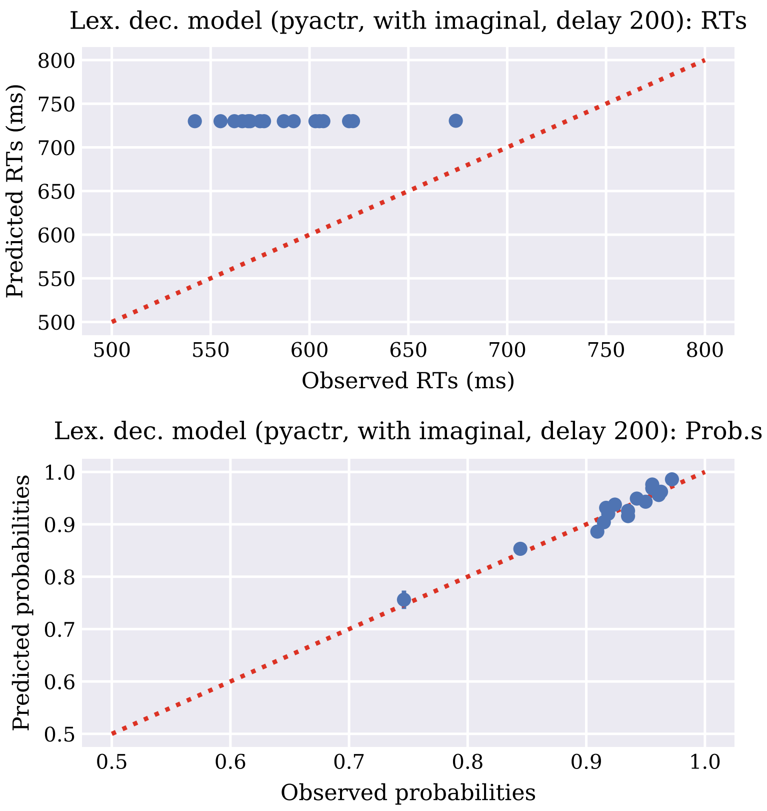


Fig. 7.6 Lex. dec. model, Bayes+ACT-R, with imaginal buffer and default delay (200 ms): estimated and observed RTs and probabilities

7.4.3 Bayes+ACT-R Lexical Decision with Imaginal Buffer and 0 Delay

We will now improve the imaginal-buffer model introduced in the previous subsection by setting the imaginal delay to 0 ms, instead of its default 200 ms value. When we built our left-corner parser in Chap. 4, we already saw that the imaginal delay might need to be decreased if we want to have empirically-adequate models of linguistic phenomena. This is because natural language interpretation involves incremental construction of rich hierarchical representations that seriously exceed in complexity the representations needed to model other high-level cognitive processes in ACT-R.

The full code for the model discussed in this subsection is once again available at the link provided in the appendix to this chapter—see Sect. 7.7.1. The only change relative to the model in the previous subsection is setting the delay for the imaginal buffer to 0 when the model is reset to its initial state in the `run_stimulus(word)` function.

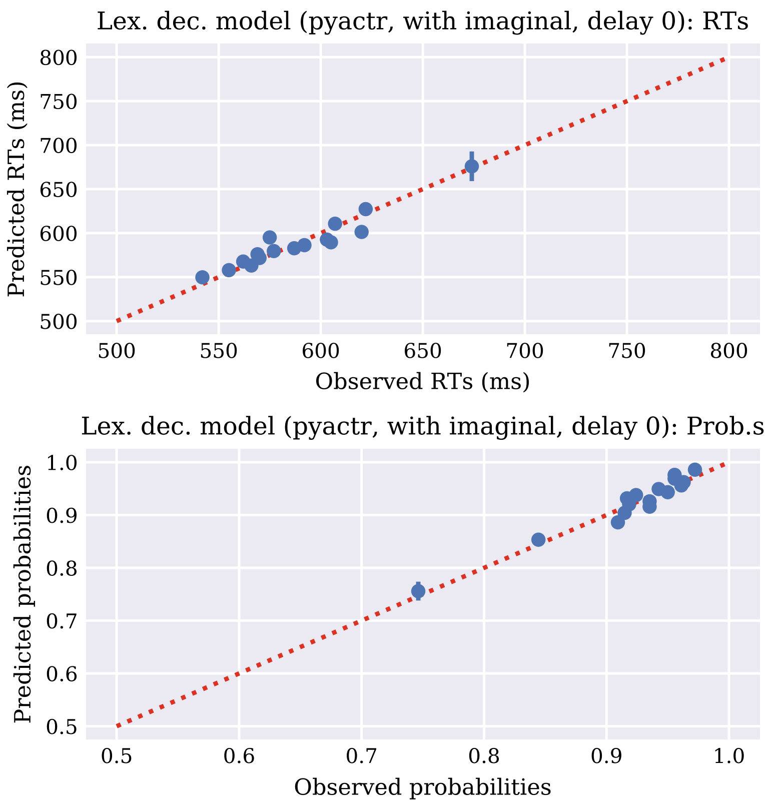


Fig. 7.7 Lex. dec. model, Bayes+ACT-R, with imaginal buffer and 0 ms delay: estimated and observed RTs and probabilities

The resulting predictions are plotted against the observed data in Fig. 7.7. We see here that, once the latency 'intercept' of the ACT-R model is suitably lowered by removing the imaginal-encoding delay, a cognitively plausible model that makes crucial use of the imaginal buffer can fit the data very well.

The `Rhat` values for this model are also below 1.1:

(2)

```
{'threshold': 1.0239260960818166,
'decay': 1.0240525803780482,
'noise': 1.0002178227957363,
'lf': 1.0144459091108735,
'le': 1.0015428790086978,
'mu_rt': array([1.00355696, 1.00402281, 1.00437915, 1.00471154,
                1.00506452, 1.00545713, 1.00575474, 1.00605512,
                1.00633888, 1.00655243, 1.00678295, 1.00685419,
                1.00690249, 1.00638288, 1.00460097, 1.00171009]),
'mu_prob': array([1.00049237, 1.00062568, 1.00073162, 1.00083857,
                  1.00094761, 1.00107675, 1.00119749, 1.00130761,
                  1.00144432, 1.00157235, 1.00167635, 1.00177533,
                  1.00196299, 1.00205086, 1.00156458, 1.00025179])}
```

We now have a formally explicit way to connect competence-level theories to experimental data via explicit processing models. That is, we can formally, explicitly connect qualitative (symbolic, competence-level) theory construction—the main business of the generative grammarian—and quantitative (subsymbolic, performance-level) statistical inference and model comparison based on experimentally collected behavioral data—the main business of the experimental linguist.

Traditionally, these are separate activities that are only connected informally. The fundamental vagueness of this informal connection is intrinsically unsatisfactory. But, in addition to that, this vagueness encourages the generative grammarian and the experimental linguist to work in separate spheres, with the generative grammarian developing sophisticated theories with a relatively weak empirical basis, and the experimental linguist often using an informal, overly simplified theory that can fit in the Procrustean bed of a multi-way ANOVA (or similar linear models).

There are several reasons for embedding ACT-R models in Bayesian models for statistical inference, rather than just using maximum likelihood estimation. These reasons are not specific to ACT-R models, but are brought into sharper relief by the complexity of these models relative to the generalized linear models standardly used in (psycho)linguistics.

The first reason is that we can put information from previous ACT-R work into the priors. Most importantly, however, the Bayesian framework enable us to perform generalized model comparison (via Bayes factors, or using other criteria). In contrast, maximum likelihood model comparison fails for models for which we cannot estimate the number of parameters in the model. Estimating the number of parameters is already difficult for models with random effects. For hybrid symbolic-subsymbolic models like the ACT-R ones we have been constructing, the question of identifying the “number” of parameters is not even well-formed.

For a distinct line of argumentation that the integration of ACT-R and Bayesian models is a worthwhile endeavor, see Weaver (2008).

7.5 Modeling Self-paced Reading with a Left-Corner Parser

Apart from the lexical decision model, Chap. 4 also showed how to implement a left-corner parser model in ACT-R/`pyactr`. We noted in that chapter that the parsing model was not realistic due to, among other things, its simplifying assumption that memory retrievals of lexical information always take a fixed amount of time, irrespective of the specific state and properties of the components of the recall process (the specific recall cue, the state of declarative memory, the contents of the other buffers etc.). Since we now have a more realistic model of lexical access at our disposal, we might want to investigate whether this model could also be used to improve our parsing model.

We can go even further than that: one interesting property of ACT-R is that it assumes one model of retrieval irrespective of the cognitive sub-domain under consideration. We can therefore ask how this model of retrieval fares with respect to language: can we model both syntactic and lexical retrieval using the same mechanisms and the same parameter values within one ACT-R model? ACT-R/`pyactr` left-corner parsing models addressing these questions were introduced and discussed in Brasoveanu and Dotlačil (2018). In this section, we will summarize the main points of that work.

Brasoveanu and Dotlačil (2018) studied the fit of the parser to human data by simulating Experiment 1 in Grodner and Gibson (2005).[9] This is a self-paced reading experiment (non-cumulative moving-window; Just et al. 1982): participants read sentences that do not appear as a whole on the screen. Rather, they have to press a key to reveal the first word, and with every key press, the previous word disappears and the next word appears. What is measured (and modeled) is how much time people spend on each word.

The modeled self-paced reading experiment has two conditions. In one condition, the subject noun phrase is modified by a subject-gap relative clause—see (3) below. In the second condition, the subject noun phrase is modified by an object-gap relative clause—see (4) below.

Using relative clauses is crucial, since this allows us to study the properties of syntactic retrieval. At the gap site, indicated as t_i in (3/4) below, the parser has to retrieve the wh-word from declarative memory to correctly interpret the relative clause. Studying the reading-time profiles of these sentences can therefore help us understand the latencies of both lexical and syntactic recall.

(3) The **reporter who$_i$ t_i sent the photographer to the editor hoped** for a story.

(4) The **reporter who$_i$ the photographer sent t_i to the editor hoped** for a story.

Brasoveanu and Dotlačil (2018) modeled 9 regions of interest (ROIs), boldfaced in (3/4) above. These are word 2 (the matrix noun in subject position) through word 10 (the matrix verb).

Just as when we modeled lexical decision, Brasoveanu and Dotlačil (2018) built more than one model and quantitatively compared them. This comparison is a necessary part of developing good ACT-R models, and cognitive models in general.[10] But more importantly, it enables us to gain insight into underlying (unobservable) cognitive mechanisms by identifying the better fitting model(s) in specific ROIs.

In total, three models were created to simulate self-paced reading and parsing. All three models were extensions of the eager left-corner parser described in Sect. 4.4. The two main modifications were: (i) the parser was extended with a more realistic model of lexical access, the same as the one used in the second ACT-R model for lexical decision in this chapter (see Sect. 7.3), and (ii) the parser had to recall the

[9] A small subset of this data was studied in a different ACT-R model of parsing, the left-corner parser discussed in Lewis and Vasishth (2005).

[10] As well as statistical models, machine learning models etc.

wh-word in the relative clause to correctly parse it. The parser incorporated visual and motor modules, just like the one in Chap. 4.

The three models differ from each other in two respects. First, Models 1 and 2 assume a slightly different order of information processing than Model 3. Models 1 and 2 are designed in a strongly serial fashion:

- first, a word w is attended visually;
- after that, its lexical information is retrieved, and syntactic retrieval also takes place (if applicable, e.g., when we need to retrieve the relativizer who_i);
- the parse tree is then created and, finally,
- visual attention is moved to the next word $w + 1$ at the same time as the motor module is instructed to press a key to reveal that word;
- then the whole process is repeated for word $w + 1$.

The processes in Model 3 were staged in a more parallel fashion: after lexical retrieval, syntactic retrieval (if applicable) and syntactic parsing happened at the same time as visual-attention and motor commands were prepared and executed. This difference is schematically shown in Figs. 7.8 and 7.9.

The second way the models differ is with respect to the analysis of subject gaps. Models 1 and 3 assume that the parser predictively postulates the subject gap immediately after reading the wh-word (word 3 in (3) and (4)). This strategy should slow

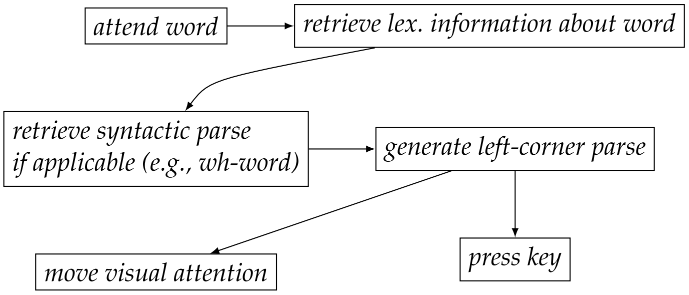

attend word
retrieve lex. information about word
retrieve syntactic parse if applicable (e.g., wh-word)
generate left-corner parse
move visual attention
press key

Fig. 7.8 Flowchart of parsing process per word for Models 1 and 2

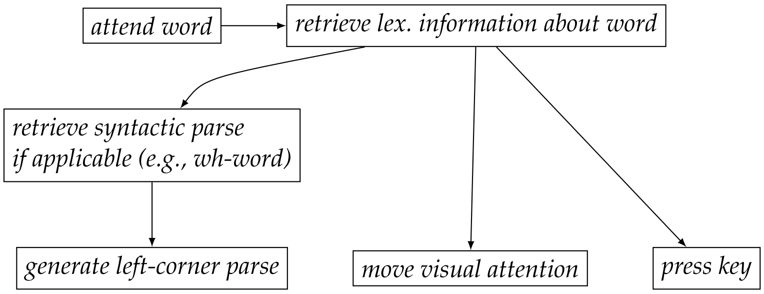

attend word
retrieve lex. information about word
retrieve syntactic parse if applicable (e.g., wh-word)
generate left-corner parse
move visual attention
press key

Fig. 7.9 Flowchart of parsing process per word for Model 3

down the parser on the wh-word itself, since it has to postulate the upcoming gap when reading it. But the strategy predicts that the parser will speed up when reading the following word in the subject-relative clause sentence (3), since the parser has already postulated the gap and nothing further needs to be done to parse the gap.

In contrast, Model 2 assumes that no subject gap is predictively postulated when reading the wh-word: the gap is parsed bottom-up. This strategy predicts faster reading times on the wh-word compared to Models 1 and 3. But it also predicts a slowdown on the next word in the subject-relative clause sentence (3), since it is at this point that the subject gap is parsed/postulated.

Why would we compare these three models? The main reason is to test two distinct hypotheses (qualitative/symbolic theories) about the human processor. These hypotheses are commonly entertained in psycholinguistics, but are not usually fully formalized and computationally implemented.

And it is important to realize that we can never really establish at an *informal* level if hypotheses embedded in complex competence-performance theories like the ones we're entertaining here make correct predictions. To really test hypotheses and theories at this level of complexity, we need to fully formalize and computationally implement them, and then attempt to fit them to experimental data. We don't really know what a complex model does until we run it.

The two hypotheses we test are the following. First, we implement and test the standard assumption that the parser is predictive and fills in gap positions before they appear (cf. Stowe 1986; Traxler and Pickering 1996). Given that hypothesis, we expect that Models 1 and 3 fit the reading data better than Model 2.

Second, it is commonly assumed that processing is to some degree parallel. In particular, a standard assumption of one of the leading models of eye movement (E-Z Reader, Warren et al. 2009) is that moving visual attention to word $n + 1$ happens alongside with the syntactic integration of word n. Under this hypothesis, we expect Model 3 to have a better fit than Models 1 and 2.

Both predictions turn out to be correct, supporting previous claims and showing that these claims hold under the careful scrutiny of fully formalized and computationally implemented end-to-end models that are quantitatively fit to experimental data.

Equally importantly, this shows that our Bayes+ACT-R framework can be used to quantitatively test and compare qualitative (symbolic) hypotheses about cognitive processes underlying syntactic processing.

The code for Model 3 is linked to in the appendix to this chapter (see Sect. 7.7.2). The three models were fit to experimental data from Grodner and Gibson (2005) (their Experiment 1) using the Bayesian methods described previously in this chapter.

Four parameters were estimated: the k parameter, which scales the effect of visual distance, the rule firing parameter r, the latency factor lf and the latency exponent le. Of these, only the first two have not been discussed in this chapter.

The rule firing parameter specifies how much time any rule should take to fire and has been (tacitly) used throughout the whole book. The default value of this parameter, which we always used up to this point, is 50 ms.

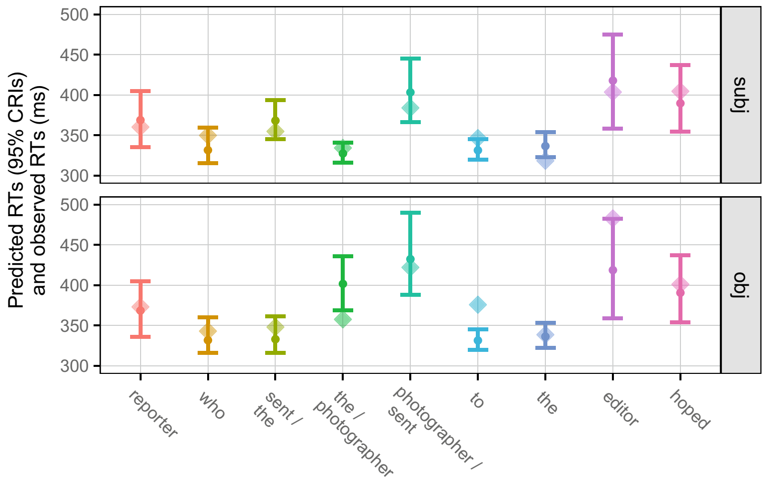


Fig. 7.10 Model 1: postulated subject gaps

The k parameter is used to modulate the amount of time that visual encoding T_{enc} takes, and it has been discussed in Chap. 4, Sect. 4.3.1.[11] We fit the k parameter mainly to show that parameters associated with peripherals (e.g., the visual and motor modules) can be jointly estimated with procedural and declarative memory parameters when fitting full models to data.

After we fit (the parameters of) the three models to data, we collect the posterior predictions for the 9 ROIs in the two (subject-gap and object-gap) conditions, plotted in Figs. 7.10, 7.11 and 7.12. The diamonds in these graphs indicate the actual, observed mean RTs for each word from Grodner and Gibson (2005). The bars provide the 95% CRIs for the posterior mean RTs, which are plotted as filled dots.

It is important to note that what we estimate here are parameters for the full process of reading the 9 ROIs. We do not estimate means and CRIs region by region (which is the current standard in the field), falsely assuming independence and leaving the underlying dependency structure, i.e., the parsing process, largely implicit and unexamined.

Figure 7.10 shows that Model 1 captures wh-gap retrieval well: the observed mean reading times on the 3rd word (*sent*) in the top panel (**subj**-gap) and the 5th word (also *sent*) in the bottom panel (**obj**-gap) fall within the CRI. However, the spillover effect on the word after the object gap—the 6th word (*to*) in the bottom panel—is not captured: the model underestimates it pretty severely.

[11]Recall that T_{enc} is specified by the function $K \cdot (-\log f) \cdot e^{kd}$, where k is the parameter that scales the effect of visual distance d measured in degrees of visual angle, and f is the (normalized) frequency of the object (word) being encoded. However, since word frequency affects lexical retrieval, we do not need to use it in visual encoding, so we substitute word length (a straightforward visual property of a word) for $-\log f$. K is another parameter, set to its default value of 0.01.

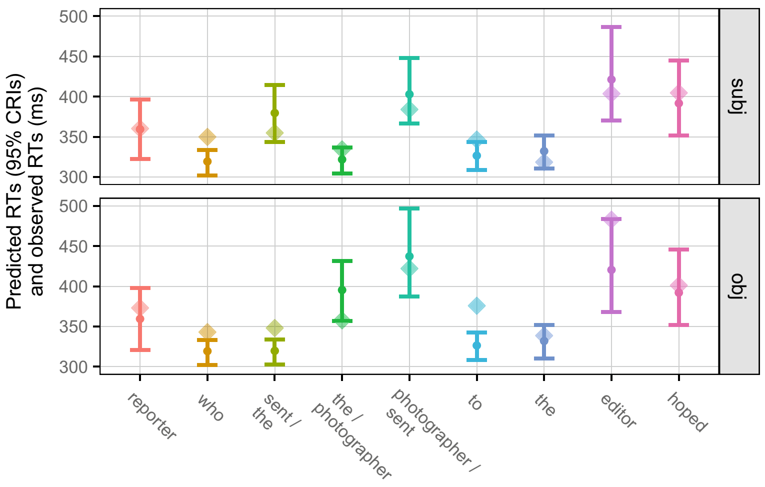


Fig. 7.11 Model 2: no postulated subject gaps

The posterior predictions of Model 2, provided in Fig. 7.11, are clearly worse: the 95% CRIs are completely below the observed mean RTs for the wh-word in both conditions, and also for the word immediately following the wh-word in the object-gap condition. This indicates that the model underestimates the parsing work triggered by the wh-word, and it also underestimates the reanalysis work that needs to be done on the word immediately following the wh-word in the object-gap condition.

Finally, Model 3 is the best among these three models. It captures the spillover effect for object gaps and increases the precision of the estimates (note the smaller CRIs). At the same time, Model 3 maintains the good fit exhibited by Model 1 (but not Model 2) for the wh-word and the following word in both conditions. This is shown in Fig. 7.12. As we already mentioned, the code for this final and most successful model is linked to in the appendix to this chapter (Sect. 7.7.2).

This relatively informal quantitative comparison between models can be made more precise by using WAIC measures for model comparison. For example, if we use WAIC$_2$,[12] which is variance based, we can clearly see that Model 3 has the most precise posterior estimates for the Grodner and Gibson (2005) data; see Brasoveanu and Dotlačil (2018) for more details.

Thus, we see that the left-corner parser, first introduced in Chap. 4, can be extended with a detailed, independently motivated theory of lexical and syntactic retrieval. The resulting model can successfully simulate reading time data from a self-paced reading experiment.

The fact that the three models we considered help us distinguish between several theoretical assumptions, and that the model with the best fit implements hypotheses

[12]See Gelman et al. (2013) for more discussion of WAIC$_1$ and WAIC$_2$.

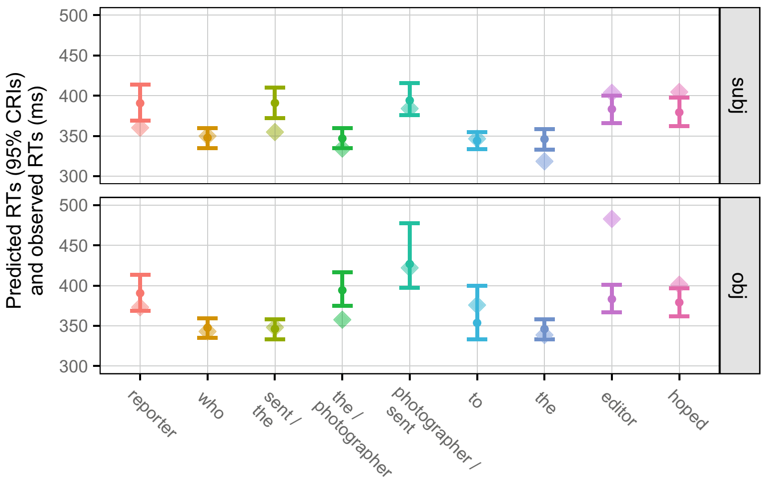


Fig. 7.12 Model 3: ‘parallel’ reader

that we expect to be correct for independent reasons, is encouraging for the whole enterprise of computational cognitive modeling pursued in this book.

Finally, we see that Model 3 does not show any clear deficiencies in simulating the mean RTs of self-paced reading tasks, even though it presupposes one and the same framework for both lexical and syntactic retrieval. This supports the ACT-R position of general recall mechanisms across various cognitive sub-domains, including linguistic sub-domains such as lexical and syntactic memory.

Before concluding, we have to point out that, even though the investigation presented in Brasoveanu and Dotlačil (2018) and summarized in this section is very promising, it is rather preliminary, particularly when compared to the models in the rest of this chapter and the rest of this book. Furthermore, the estimates of the three models we just discussed were obtained using different sampling methods than the ones we use for the Bayes+ACT-R models throughout this book.

Improving on these preliminary results and models, and investigating if the sampling methods used in this book would substantially benefit the ACT-R models in Brasoveanu and Dotlačil (2018) is left for a future occasion.

7.6 Conclusion

The models discussed in this chapter show that the present computational implementation of ACT-R can be used to successfully fit data from various linguistic experiments, as well as compare and evaluate assumptions about underlying linguistic representations and parsing processes. While one could investigate many experiments using the presented methodology, we opted for a different approach here, focusing only on a handful of studies and dissecting modeling assumptions and the way computational modeling can be done in our Bayes+ACT-R/`pymc3`+`pyactr` framework.

We take the results to be encouraging. We believe they provide clear evidence that developing precise and quantitatively accurate computational models of lexical access and syntactic processing is possible in the proposed Bayes+ACT-R framework, and a fruitful way to pursue linguistic theory development.

Unfortunately, the models are clearly incomplete with respect to many aspects of natural language use in daily conversation. An important aspect that is completely missing in these models stands out: natural language meaning and interpretation.

Our goal in conversation is to understand what others tell us, not (just) recall lexical information and meticulously parse others' messages into syntactic trees. Ultimately, we construct meaning representations, and computational cognitive models should have something to say about that. This is precisely what the next two chapters of this book will address.

7.7 Appendix: The Bayes and Bayes+ACT-R Models

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

7.7.1 Lexical Decision Models

File **ch7_first_three_models.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_first_three_models.py.

File **ch7_lexical_decision_pyactr_no_imaginal.py**.:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_lexical_decision_pyactr_no_imaginal.py.

File **ch7_lexical_decision_pyactr_with_imaginal.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_lexical_decision_pyactr_with_imaginal.py.

File **ch7_lexical_decision_pyactr_with_imaginal_delay_0.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_lexical_decision_pyactr_with_imaginal_delay_0.py.

7.7.2 *Left-Corner Parser Models*

File **readme.txt** explains how to run the model:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/readme.txt.

File **parser_rules.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/parser_rules.py.

File **run_parser.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/run_parser.py.

File **estimation_subj_obj_extraction.py**:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/estimation_subj_obj_extraction.py.

Chapter 8
Semantics as a Cognitive Process I: Discourse Representation Structures in Declarative Memory

In this chapter, we introduce our assumptions about semantic representations and build a semantic processor, that is, a basic parser able to incrementally construct such semantic representations. Our choice for a processing-friendly semantics framework is Discourse Representation Theory (DRT, Kamp 1981; Kamp and Reyle 1993).

We build an ACT-R parser for Discourse Representation Structures (DRSs), and motivate the assumptions we make when building it by accounting for the fan experiment reported in Anderson (1974), as summarized and discussed in the more recent Anderson and Reder (1999).

The fan experiment investigates how basic propositional information of the kind encoded by atomic DRSs[1] is (stored and) retrieved from declarative memory. This is an essential component of real-time semantic interpretation in at least two respects:

i. incrementally processing semantic representations involves composing/integrating semantic representations introduced by new sentences or new parts of a sentence with semantic representations of the previous discourse;
ii. incremental interpretation also involves evaluating new semantic representations relative to our mental model of the world, and integrating their content into our world knowledge database stored in declarative memory.

The main reason for selecting DRT as our semantic framework is that atomic DRSs and the compositional construction principles used to build them provide meaning representations and elementary compositional operations that are well understood mathematically, widely used in formal semantics, and can simultaneously function as both meaning representations (logical forms, in linguistic parlance) and their content/world knowledge (models, in linguistic parlance).

[1] Atomic DRSs are equivalent to atomic first-order logic formulas, conjunctions thereof, and atomic formulas or conjunctions thereof with a prefix of existential quantifiers. Multiple atomic DRSs can be merged into a single atomic DRS—with caveats for certain cases, usually requiring bound-variable renaming. For more discussion, see Kamp and Reyle (1993), Groenendijk and Stokhof (1991) and Muskens (1996) among others.

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_8

Because of this double function, DRT and atomic DRSs can be thought of as mental models in the sense of Johnson-Laird (1983, 2004); Johnson-Laird et al. (1989), with several advantages. First, they provide a richer and better understood array of representations and operations. Second, they come with a comprehensive mathematical theory of their structure and interpretation (dynamic logic/dynamic semantics). Finally, they are well-known and used by linguists in the investigation of a wide variety of natural language semantic phenomena.[2]

It is no accident that DRT is the most obvious choice for psycholinguistic models of natural language semantics. DRT has always had an explicit representational commitment, motivated by the goal of interfacing semantics and cognitive science more closely (Kamp 1981 already mentions this).

Importantly, DRT is a dynamic semantic framework, so it has a notion of *DRS merge*, that is, a merge of two (conjoined) representations into a larger representation of the same type. This DRS merge operation makes the construction, maintenance and incremental update of semantic representations more straightforward and, also, similar to the construction, maintenance and incremental update of syntactic representations.[3]

DRT, however, is not the only possible choice as a way to add meaning representation to cognitive models of language comprehension. Less "representational" systems—whether dynamic, e.g., Dynamic Predicate Logic (DPL, Groenendijk and Stokhof 1991) or Compositional DRT (CDRT, Muskens 1996), or static, e.g.,

[2]The ability of DRSs to do this double duty (both logical forms and partial model structures) is restricted to atomic DRSs—see the discussion of persistent DRSs and model extension in Kamp and Reyle (1993, pp. 96–97). We include a brief quote from that discussion here for ease of reference:

> […] [I]t is not all that easy to see any very clear difference between models and [atomic/persistent] DRSs at this point: both provide sets of "individuals", to which they assign names, properties and relations. Presently we will encounter other DRSs with a more complicated structure [non-atomic DRSs, needed for negation, conditionals, quantifiers etc.], and which no longer have the persistence property. These DRSs will look increasingly different from the models in which they are evaluated, and the illusion that DRSs are just small models will quickly evaporate.
>
> [But] persistent DRSs [can be thought of as partial models] in that they will typically assert the existence of only a small portion of the totality of individuals that are supposed to exist in the worlds of which they intend to speak, [and] in that they will specify only some of the properties and relations of those individuals they mention. Thus a DRS may, for given discourse referents x and y belonging to its universe, simply leave it open whether or not they stand in a certain relation. Models, in contrast, leave no relevant information out. Thus, if $\mathtt{a}$ and $\mathtt{b}$ are individuals in the universe of model $\mathfrak{M}$ and the pair $\langle \mathtt{a}, \mathtt{b} \rangle$ does not belong to the extension in $\mathfrak{M}$ of, say, the predicate `owns`, then this means that `a` does not own `b`, not that the question whether `a` owns `b` is, as far as $\mathfrak{M}$ is concerned, undecided. (Kamp and Reyle 1993, pp. 96–97).

[3]Merging DRSs—when possible—is a consequence of various facts about dynamic conjunction and the update semantics associated with variable assignments and atomic lexical relations—see Groenendijk and Stokhof (1991), Muskens (1996), Brasoveanu and Dotlačil (2007, Chap. 2) and Brasoveanu and Dotlačil (2020) among others.

Montague's Intensional Logic (IL, Montague 1973), variants of Gallin's Ty3/Ty4/etc. (Gallin 1975) or partial/multivalued logics (Muskens 1995a)—are also possible, although less straightforwardly so, at least at a first glance.

The extent to which these alternative semantic frameworks are performance-friendly, that is, the extent to which they are fairly straightforwardly embeddable in a cognitive-architecture-based framework for mechanistic processing models, might provide a useful way to distinguish between them, and a plausible new metric for semantic theory evaluation. What we have in mind here is nothing new: Sag (1992) and Sag and Wasow (2011) argue persuasively that linguistic frameworks should be evaluated with respect to their performance-friendliness. The discussion there understandably focuses on syntactic frameworks, but it is easily transferable to semantics.

Our hope is that, once a set of 'performance-friendly' semantic frameworks are identified, and the ways of embedding them into explicit processing models for semantics are explored and motivated, the predictions of the resulting competence-performance models will enable us to empirically distinguish between these semantic frameworks.

The structure of this chapter is as follows. In the following Sect. 8.1, we introduce the fan-effect phenomenon and the original experiment reported in Anderson (1974) that established its existence. The fan effect provides a window into the way propositional facts—basically, atomic DRSs—are organized in declarative memory.

Section 8.2 discusses how DRSs can be encoded in ACT-R chunks, and makes explicit how the fan effect can be interpreted as reflecting DRS organization in declarative memory.

Sections 8.3 and 8.4 model the fan experiment by parsing the experimental items (simple English sentences with two indefinites) into DRSs, storing them in memory, and fitting the resulting model to the retrieval latencies observed in the experiment.

Parsing natural language sentences into DRSs is fully modeled: Sect. 8.3 explicitly models how experimental participants comprehend these sentences by means of a fully specified incremental parser/interpreter for both syntactic and semantic representations. This eager left-corner parser incrementally constructs the syntactic representations for the experimental sentences (phrase-structure grammar trees of the kind we already discussed in Chap. 4), and in parallel, it incrementally constructs the corresponding semantic representations, that is, DRSs.

Modeling the fan experiment in Anderson (1974) by means of this incremental syntax/semantics parser will enable us to achieve two goals. First, we shed light on essential declarative memory structures that subserve the process of natural language interpretation. Second, we introduce and discuss basic modeling decisions we need to make when integrating the ACT-R cognitive architecture and the DRT semantic framework.

Section 8.4 shows that the incremental syntax/semantics parser is able to fit the fan-effect data well, and Sect. 8.5 summarizes the main points of this chapter.

8.1 The Fan Effect and the Retrieval of DRSs from Declarative Memory

The fan effect

> refers to the phenomenon that, as participants study more facts about a particular concept, their time to retrieve a particular fact about that concept increases. Fan effects have been found in the retrieval of real-world knowledge [...], face recognition [...] [etc.] The fan effect is generally conceived of as having strong implications for how retrieval processes interact with memory representations. It has been used to study the representation of semantic information [...] and of prior knowledge [...]. (Anderson and Reder 1999,186).

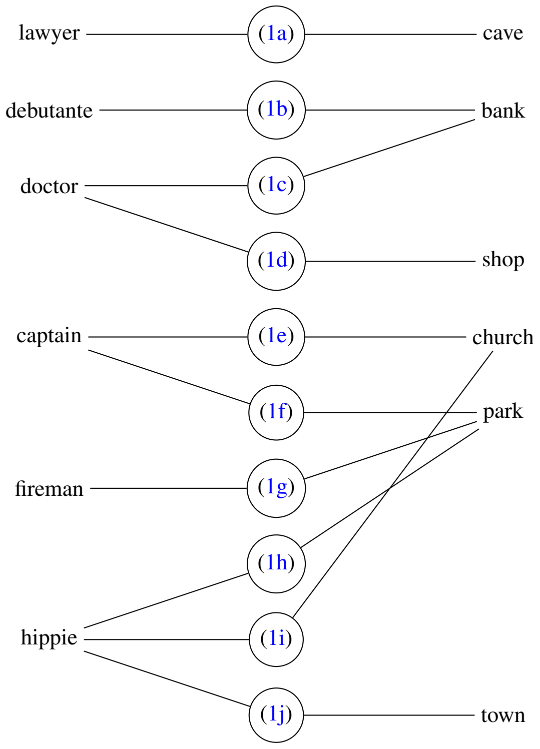
The original experiment in Anderson (1974) demonstrated the fan effect in recognition memory. Participants studied 26 facts about people being in various locations, 10 of which are exemplified in (1) below.

(1) a. A lawyer is in a cave.
    b. A debutante is in a bank.
    c. A doctor is in a bank.
    d. A doctor is in a shop.
    e. A captain is in a church.
    f. A captain is in a park.
    g. A fireman is in a park.
    h. A hippie is in a park.
    i. A hippie is in a church.
    j. A hippie is in a town.

In the training part of the experiment, participants committed 26 items of this kind to memory. In the test part of the experiment, participants were presented with a series of sentences, some of which they had studied in the training part and some of which were novel. They had to recognize the sentences they had studied, i.e., the targets, and had to reject the foils, which were novel combinations of the same people and locations.

We selected the 10 items in (1) because, together, they form a minimal network of facts that instantiates the 9 experimental conditions in Anderson (1974). These conditions have to do with how many studied facts are connected to each type of person and location. To see this explicitly, we can represent the set of 10 facts in (1) as a network in which each fact is connected by an edge to the type/concept of person and location it is about, as shown in (2) below. Person concepts are listed on the left, and location concepts are listed on the right.

(2)



The network representation in (2) shows how different person and location concepts fan into 1, 2 or 3 sentences/facts. The table in (3) below shows how 9 of the 10 items exemplify the 9 conditions of the Anderson (1974) fan experiment (the 10th item is needed in our fact database to be able to instantiate all 9 conditions).

(3)

		location fan		
		1	**2**	**3**
person fan	**1**	lawyer-cave (1a)	debutante-bank (1b)	fireman-park (1g)
	2	doctor-shop (1d)	captain-church (1e)	captain-park (1f)
	3	hippie-town (1j)	hippie-church (1i)	hippie-park (1h)

The table in (3) shows that the term 'fan' refers to the number of facts, that is, *sentences*, associated with each common concept, that is, *noun*. The mean reaction times (RTs, measured in s) for target recognition and foil rejection in the Anderson (1974) experiment are provided in the tables below (reproduced from Anderson and Reder 1999, p. 187, Table 2).

(4)

Target		location fan		
	RTs	**1**	**2**	**3**
person fan	**1**	1.11	1.17	1.15
	2	1.17	1.20	1.23
	3	1.22	1.22	1.36

(5)

Foil RTs		location fan **1**	**2**	**3**
person fan	**1**	1.20	1.25	1.26
	2	1.22	1.36	1.47
	3	1.26	1.29	1.47

Several generalizations become apparent upon examining this data.

(6) Fan-effect generalizations (as summarized in Anderson and Reder 1999, p. 187):

a. averaging over targets and foils, the effect of 1-fan (both person and location) was about 1.2 s and increased by about 50 ms for each additional fan:
   - the effect was about 1.25 s for 2-fan and about 1.3 s for 3-fan;
   - other experiments have shown that the fan effects can be of different sizes for different kinds of concepts, e.g., animates versus inanimate objects versus small locations versus large locations;

b. the min effect: retrieval latency is a function of the minimum fan associated with a probe;
   - for example, participants tend to respond more slowly to the 2–2 fan items than to the 1–3 or 3–1 items;
   - this effect has been repeatedly replicated;
   - it is taken as evidence for parallel access to memory from the two cues/concepts, with search being more determined by the lower fan concept;

c. approximately equal fan effects for targets and foils:
   - the target means were 1.16 s for 1-fan, 1.20 s for 2-fan, and 1.26 s for 3-fan;
   - the foil means were 1.23 s for 1-fan, 1.33 s for 2-fan, and 1.37 s for 3-fan;
   - we see a somewhat larger effect for foils, but this effect is reversed in other data sets;
   - this suggests that foil rejection is not done by a serial (possibly exhaustive) search of the facts one knows about a cue/concept.

The ACT-R account of these generalizations, together with the quantitative fit to the data in (4) and (5), is provided in Andreson and Reder (1999, pp. 188–189). This account crucially relies on the spreading activation component of the ACT-R equation for activation of chunks in memory that we introduced in Chap. 6. Recall that the activation A_i of a fact/sentence i in memory is:

(7) $$A_i = B_i + \sum_j W_j S_{ji}$$

Base activation B_i is determined by the history of usage of fact i (how many times i was retrieved and how long ago), while the spreading activation component

$\sum_j W_j S_{ji}$ boosts the activation of fact i based on the concepts j that fact i is associated with. Specifically, for each concept j, W_j is the extra activation flowing from concept j, and S_{ji} is the strength of the connection between concept j and fact i that modulates how much of the extra activation W_j actually ends up flowing to fact i.

The account of the fan effect observed in Anderson (1974) crucially relies on the strengths of association S_{ji} between concepts j and facts i. These strengths are modeled as shown below:

(8) $S_{ji} = S + \log P(i|j)$

S is a constant (baseline strength) to be estimated for specific experiments, while $P(i|j)$ is an estimate of the probability of needing fact i from declarative memory when concept j is present in the cognitive context, e.g., in the goal or imaginal buffers. That is, $P(i|j)$ is an estimate of how predictive concept j is of fact i.

In an experimental setup in which all facts are studied and tested with equal frequency, it is reasonable to assume that the predictive probability $P(i|j)$ is simply $\frac{1}{\text{fan}_j}$, where fan_j is the fan of concept j. If a concept has a fan of 1, e.g., *lawyer* in (2) above, then the probability of the associated fact (1a) is 1. However, if a concept has a fan of 3, like *hippie*, all three facts associated with this concept, namely (1h), (1i) and (1j), are equiprobable, with a probability of $\frac{1}{3}$.

Thus, for the experiment in Anderson (1974), the strength-of-association equation can be simplified to:

(9) $S_{ji} = S + \log\left(\frac{1}{\text{fan}_j}\right) = S - \log(\text{fan}_j)$

The equation in (9) implies that strength of association, and thereby fact activation, will decrease as a logarithmic function of concept fan. Strength/activation of a fact decreases with concept fan because the probability of a fact given a concept decreases with the fan of that concept. For more discussion of strength of association, source activation etc., see Sect. 6.3.

The last piece of the ACT-R model is the function that outputs retrieval latencies given fact activations. This is provided in (10) below, which should be familiar from Chaps. 6 and 7 (the latency exponent is omitted, i.e., set to its default value 1).

(10) $T = I + Fe^{-A_i}$

The only addition is the intercept I, which captures all cognitive activities other than fact retrieval, e.g., encoding the test sentence, generating the response etc. Recall that we used a similar intercept in Chap. 7 when we proposed the first two models of lexical decision—see Sects. 7.2 and 7.3

All of these model components will be made explicit in Sects. 8.3 and 8.4 below, where we provide the first end-to-end ACT-R model of the fan effect in the literature. No such models have been available because incremental semantic interpretation was never explicitly modeled in ACT-R before.

We can further simplify our mathematical model specification by putting together Eqs. (7), (9) and (10) as follows:

(11) $$T = I + Fe^{-A_i} = I + Fe^{-B_i - \sum_j W_j S_{ji}} = I + Fe^{-B_i - \sum_j W_j (S - \log(\text{fan}_j))}$$
$$= I + Fe^{-B_i - S\sum_j W_j} e^{\sum_j W_j \log(\text{fan}_j)} = I + Fe^{-B_i - S\sum_j W_j} \prod_j e^{W_j \log(\text{fan}_j)}$$
$$= I + Fe^{-B_i - S\sum_j W_j} \prod_j \text{fan}_j^{W_j}$$
$$= I + F' \prod_j \text{fan}_j^{W_j}, \qquad \text{where } F' = Fe^{-B_i - S\sum_j W_j}$$

We can further assume that $\sum_j W_j$ is fixed and set to 1. This is motivated by the capacity limitations in retrieval discussed in Anderson et al. (1996). Finally, if we assume that the source activations W_j of all concepts j are equal, we can set them all to a constant W. The final form of the ACT-R model for fan-dependent retrieval latencies is therefore:

(12) $T = I + F' \prod_j \text{fan}_j^{W}$, where:

- $F' = Fe^{-B_i - S}$
- $W = \frac{1}{\#\text{ of concepts } j \text{ associated with fact } i}$

The latency equation in (12) shows how the ACT-R model captures the fan effect, i.e., the generalization in (6a) that recognition latency increases with fan. To see this more clearly, let's apply this equation to the facts in the Anderson (1974) experiment. These facts are connected to 3 concepts (a person, a location, and the predicate *in*), so the source activation W is $\frac{1}{3}$. The resulting latency equation is provided in (13).

(13) $$T = I + F'(\text{fan}_{person} \cdot \text{fan}_{location} \cdot \text{fan}_{in})^{\frac{1}{3}} = I + F' \sqrt[3]{\text{fan}_{person} \cdot \text{fan}_{location} \cdot \text{fan}_{in}}$$

The predicate *in* is connected to all the sentences/facts, so it will have a constant fan and it will contribute a constant amount of activation (hence latency) across all conditions in the experiment. But the person and location fan values fan_{person} and $\text{fan}_{location}$ are manipulated in the experiment, and the equation in (13) correctly predicts that, as they increase, the corresponding latency T increases.

The equation in (13) also provides an account of the min effect (generalization (6b) above). The reason is that the product of a set of numbers with a constant sum, specifically the product $\text{fan}_{person} \cdot \text{fan}_{location}$, is maximal when the numbers are equal, e.g., $2 \times 2 > 3 \times 1$.

To understand how the ACT-R model captures the generalization in (6c), that is, the fact that foils are retrieved almost as quickly as target sentences, we need to specify the process of foil recognition. The main idea is that foil recognition does not involve an exhaustive search. Instead:

(14) Foil recognition

- foils are recognized by retrieving a fact that involves either the person or the location in the foil;

- if the retrieved fact does not match the test sentence, participants will respond ‘false.’

For simplicity, we can assume that participants retrieve the mismatching fact half the time with a person cue, and half the time with a location cue. Either way, *mismatching facts have one less source of spreading activation* coming from the foil sentence (which is encoded in the goal or imaginal buffer) than matching facts. Their total activation will therefore be lower, so the time to retrieve foils will be slightly higher than the time to retrieve targets.

By the same token, *the correct fact will almost always be retrieved for target sentences because this fact will have more sources of spreading activation* coming from the target sentence—hence a higher total activation—than any other incorrect fact in memory.

To be more specific, the activation of the correct fact given a target sentence is shown in (15) below. Recall that all source activations W_{person}, $W_{location}$ and W_{in} are assumed to be $\frac{1}{3}$.

(15) $A_{target} = B_{target} + \frac{S-\log(\text{fan}_{person})}{3} + \frac{S-\log(\text{fan}_{location})}{3} + \frac{S-\log(\text{fan}_{in})}{3}$

All three terms $\frac{S-\log(\text{fan}_{person})}{3}$, $\frac{S-\log(\text{fan}_{location})}{3}$ and $\frac{S-\log(\text{fan}_{in})}{3}$ are positive, so they each add an extra activation boost, hence an extra decrease in latency of retrieval.

In contrast, activation of foils has only 2 spreading activation terms. Either the location term is missing, as in (16a) below, where fact retrieval for the foil sentence is person-based, or the person term is missing, as in (16b), where fact retrieval for the foil sentence is location-based. Less spreading activation means lower total activation, which leads to slightly increased latency of retrieval.

(16) a. $A_{foil} = B_{foil} + \frac{S-\log(\text{fan}_{person})}{3} + \frac{S-\log(\text{fan}_{in})}{3}$

b. $A_{foil} = B_{foil} + \frac{S-\log(\text{fan}_{location})}{3} + \frac{S-\log(\text{fan}_{in})}{3}$

This ACT-R model is sufficiently flexible to account for a range of effects beyond the original fan experiment in Anderson (1974). Different values for sources of activation W_j can account for differential fan effects associated with different types of concepts (inanimate objects versus persons, for example).

Similarly, different values for strengths of activation S_{ji} that depend on the frequency of presentation of fact-concept associations, which affect probabilities $P(i\,|\,j)$, can account for retrieval interference effects that go beyond simple fan effects. For more discussion, see Anderson and Reder (1999).

However, for the remainder of this chapter, we will focus exclusively on the original fan experiment in Anderson (1974), and specifically on modeling retrieval latencies for the target sentences in (4). The next section reformulates the fan effect in terms of the way meaning representations (DRSs) that are associated with target sentences are organized in declarative memory.

8.2 The Fan Effect Reflects the Way Meaning Representations (DRSs) Are Organized in Declarative Memory

We can reformulate the notion of fan in Anderson's experiment, as well as the network of facts and concepts in (2) above, as a relation between the main DRS contributed by a sentence and the sub-DRSs contributed by its three parts: the person indefinite, the location indefinite, and the relational predicate *in*.

Consider the 1–1 fan (that is, 1 person–1 location fan) example in (1a), repeated in (17) below. The DRSs (meaning representations) of the three major components of the sentence, that is, the indefinites *a lawyer* and *a cave*, and the binary predicate *it*, are composed/combined together to form the DRS/meaning representation for the full sentence.

The exact nature of the three meaning components and the composition method vary from framework to framework. For example, the method of composition in Kamp and Reyle (1993) is a set of construction principles operating over hybrid representations combining DRSs and syntactic trees. The method of composition in Brasoveanu (2007, Chap. 3), building on much previous work (Groenendijk and Stokhof 1990; Chierchia 1995; Muskens 1995b, 1996 among others) is classical Montagovian function application/β-reduction operating over DRS-like representations, which are just abbreviations of terms in a many-sorted version of classical simply-typed lambda calculus (Gallin 1975). Finally, the method of composition in Brasoveanu and Dotlačil (2015b) (building on Vermeulen 1994 and Visser 2002) is dynamic conjunction over DRSs interpreted as updates of richly structured interpretation contexts that record information state histories.

However, we do not need to fully specify a semantic framework for natural language meaning representation and composition to reformulate the fan experiment in formal semantics/DRT terms. It is sufficient to acknowledge that the main DRS contributed by a sentence like (17) is formed out of three sub-DRSs, contributed by the two indefinites *a lawyer* and *a cave*, and the preposition *in*.

This partitioning into three sub-DRSs matches the rough compositional skeleton generally assumed in the formal semantics literature for this type of sentences, as well as the real-time incremental comprehension process the ACT-R architecture imposes on us.

Recall that, because of the seriality imposed in ACT-R by one production firing at a time, and by the imaginal buffer being able to hold only one chunk at a time, we never have a full view of the syntactic tree representation.

Similarly, we will never have a full view of the DRS semantic representation. This representation will be assembled one sub-DRS at a time, resulting in the main DRS in (18). But this 'deep', i.e., hierarchical, representation of the main DRS with the sub-DRSs actually encoded as values of its slots is only implicitly available in declarative memory, just like the full syntactic tree of the sentence in (17) is only implicitly available in declarative memory.

(17) A lawyer is in a cave.

(18)
$${}_{\text{MAIN-DRS}}\left[\begin{array}{ll} \text{SUB-DRS}_1: & \begin{array}{|c|}\hline x \\ \hline \text{lawyer}(x) \\ \hline\end{array} \\ \\ \text{SUB-DRS}_2: & \begin{array}{|c|}\hline y \\ \hline \text{cave}(y) \\ \hline\end{array} \\ \\ \text{SUB-DRS}_3: & \begin{array}{|c|}\hline \\ \hline \text{in}(x, y) \\ \hline\end{array} \end{array}\right]$$

If we merge (i.e., dynamically conjoin and 'reduce') the three sub-DRSs into one DRS, we obtain the DRS in (19) below, which is precisely the semantic representation assigned to sentence (17) in DRT. It can be shown that this merged DRS is truth-conditionally equivalent to the classical first-order logic formula in (20),[4] which is the basic semantic representation that pretty much all static (neo-)Montagovian semantic frameworks derive for sentence (17).

(19) Merging the sub-DRSs into one DRS:
$$\begin{array}{|c|}\hline x, y \\ \hline \text{lawyer}(x) \\ \text{cave}(y) \\ \text{in}(x, y) \\ \hline\end{array}$$

(20) $\exists x \exists y(\text{lawyer}(x) \wedge \text{cave}(y) \wedge \text{in}(x, y))$

Chunks like the one exemplified in (18) lend themselves fairly straightforwardly to modeling the fan experiment in Anderson (1974). But before we show how to do that, let's take a further step and encode the sub-DRSs as chunks, i.e., as attribute-value matrices. One way to do it is as shown in (21) below:

- first, we replace the variables/discourse referents x and y with subscripted variables, e.g., v_1 and v_2; actually, while we're at it, we can drop v altogether and simply retain the indices $1, 2, \ldots$ as variables/discourse referents[5];
  - for example, lawyer(x), cave(y) and in(x, y) become lawyer(1), cave(2) and in(1, 2);

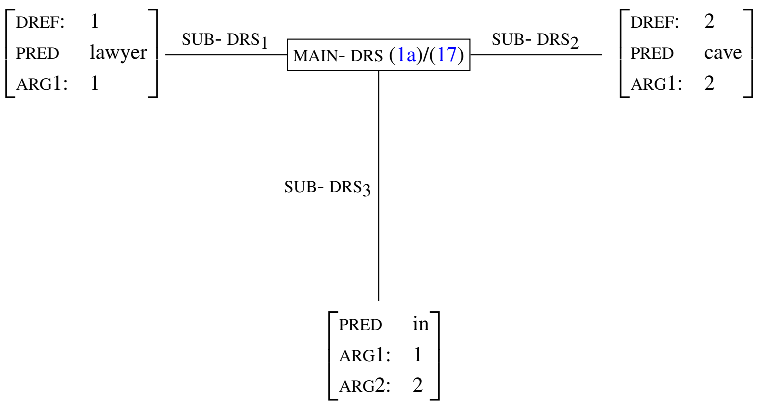
[4]See Kamp and Reyle (1993) and Groenendijk and Stokhof (1991) among others for pertinent discussion.

[5]Some formal semantics textbooks also simplify variable names to natural number indices, e.g., Heim and Kratzer (1998). Another way in which this is useful for us is that we can now take variables to be simply names for positions in a stack, that is, we implicitly move from the total variable assignments of classical first-order logic or the partial variable assignments (a.k.a. embedding functions) of DRT to (finite) stacks as the preferred way of representing interpretation contexts. See Dekker (1994), Vermeulen (1995), Nouwen (2003, 2007) among others for more discussion of stacks or stack-like structures in (dynamic) semantic frameworks for natural language interpretation.

- second, we separate DRS conditions into distinct features for the predicate and for its arguments;
  - for example, the condition in(1, 2) is replaced with the chunk/attribute value matrix $\begin{bmatrix} \textsc{pred} & \text{in} \\ \textsc{arg1:} & 1 \\ \textsc{arg2:} & 2 \end{bmatrix}$;
- finally, sub-DRSs that introduce new discourse referents—that is, they contribute something like an implicit existential quantifier in addition to conditions—will have a new feature DREF to indicate what new discourse referent they introduce;
  - for example, SUB-DRS$_1$ in (21) introduces the new discourse referent 1, which is a lawyer, so 1 is both the value of the DREF feature and the value of the ARG1 feature (since it is the first, and only, argument of the 'lawyer' predicate).

(21)
$$\textsc{main-drs}\begin{bmatrix} \textsc{sub-drs}_1\text{:} & \begin{bmatrix} \textsc{dref:} & 1 \\ \textsc{pred} & \text{lawyer} \\ \textsc{arg1:} & 1 \end{bmatrix} \\ \textsc{sub-drs}_2\text{:} & \begin{bmatrix} \textsc{dref:} & 2 \\ \textsc{pred} & \text{cave} \\ \textsc{arg1:} & 2 \end{bmatrix} \\ \textsc{sub-drs}_3\text{:} & \begin{bmatrix} \textsc{pred} & \text{in} \\ \textsc{arg1:} & 1 \\ \textsc{arg2:} & 2 \end{bmatrix} \end{bmatrix}$$

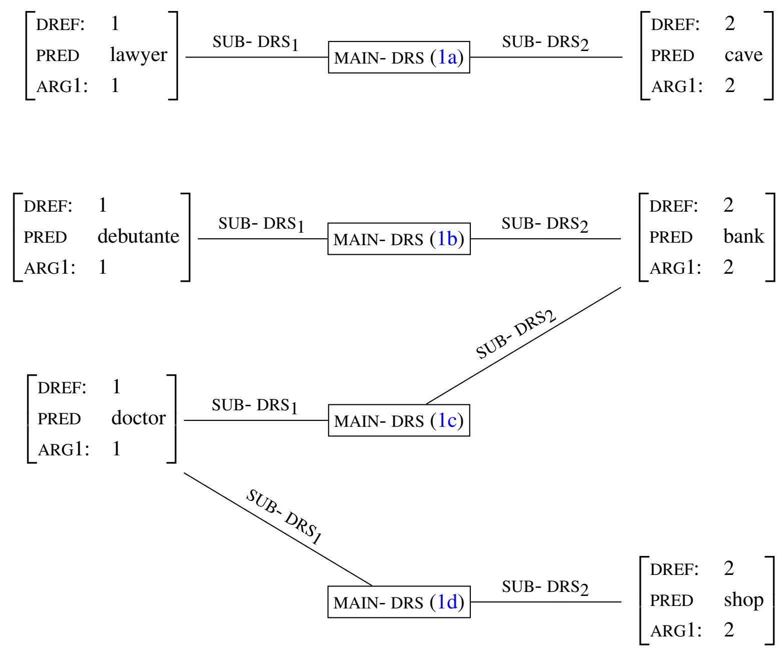
The graph like representation of the chunk in (21) is provided in (22) below.

(22)


Since all the main DRSs for the items in (1) are going to be connected to the SUB-DRS$_3$ contributed by the preposition *in*, we can omit that connection—and we will do that from now on.

With that omission in place, we can reproduce the network of facts and concepts in (2) more properly as a semantic network of DRSs in declarative memory. For space reasons, (23) below only provides the DRS network corresponding to the first four experimental items (1a) through (1d).

(23)



Structuring DRSs in memory as a network in which a main DRS contains, i.e., is connected to, sub-DRSs contributed by various sub-sentential expressions is reminiscent of the structured meanings approach to natural language semantics in Cresswell (1985).

We leave the exploration of further connections between the structure of meaning representations in declarative memory, semantic composition, incremental processing and various approaches to the formal semantics of propositional attitudes (structured meanings among others) for a future occasion.

8.3 Integrating ACT-R and DRT: An Eager Left-Corner Syntax/Semantics Parser

With these semantic assumptions in place, we are ready to specify our model for the Anderson (1974) fan experiment. The model we introduce in this section is the first model of a fan experiment to explicitly incorporate syntactic and semantic parsing.

In fact, this is the first end-to-end model of the testing phase of Anderson (1974) experiment, explicitly incorporating (i) a visual component (eye-movement while reading), (ii) a motor component (accept or reject whether the test sentence was studied in the training phase), and (iii) a syntax/semantics incremental interpreter for the test sentence as it is being read.

For simplicity, we focus exclusively on modeling target sentences, and the observed mean target latencies in (4) above. But the model will be designed with foils in mind too, and could be equally applied to model the mean foil latencies in (5) above.

The core component of our cognitive model for a participant in the Anderson (1974) fan experiment is an eager left-corner parser that parses syntactic trees and DRSs simultaneously and in parallel. As a new word is incrementally read in the usual left-to-right order for English:

- the model eagerly and predictively builds as much of the syntactic and semantic representation as it can (before deciding to move the eyes to the next word);
- this process of syntactic and semantic representation building is the process of comprehending the new word and integrating it into the currently available partial syntactic and semantic structure;
- these cognitive actions of comprehension and integration result in a new partial syntactic and semantic structure;
- in turn, this syntax/semantics structure provides the context relative to which the next word is interpreted, and which will be updated as this next word is comprehended and integrated.

The overall dynamics of parsing is thus very similar to dynamic semantics: a sequence of parsing actions or a sequence of dynamic semantics updates charts a path through the space of information states. Each info state provides both (i) the context relative to which a semantic update is interpreted and, at the same time, (ii) the context that is changed as a result of executing that update.

Information states are simpler for dynamic semantics: they are basically variable assignments or similar structures. For our ACT-R incremental parser, information states are complex entities consisting of the partial syntactic and DRS structure built up to that point, as well as the state of the buffers and modules of the ACT-R mind at that point in the comprehension process.

Our syntax/semantics parser builds semantic representations and syntactic structures simultaneously, but the two types of representations are independently encoded and relatively loosely connected. In particular, we will postulate a separate imaginal-like buffer, which we will label `discourse_context`, where DRSs are incrementally built. We will continue to store partial syntactic representations in the imaginal buffer.

The decision to build syntax/semantics representations simultaneously, but in separate buffers is in the spirit of the ACT-R architecture. We avoid integrating two complex representations directly in a single 'super-chunk', keeping chunk size relatively small and also keeping the chunks themselves relatively flat. By 'flat' chunks, we mean chunks without many levels of embedding, that is, without many features that have other chunks as values, whose features in their turn have other chunks as values etc.

These 'deep' hierarchical representations are ubiquitous in generative syntax and semantics, but they are not compatible with the view of (high-level) cognitive processes embodied by the ACT-R cognitive architecture, at least not immediately. Thus, it is important to note that the independently motivated ACT-R cognitive architecture places non-trivial constraints on the form of our linguistic performance models, and indirectly on our competence-level models.

We had to make many implementation decisions when we divided the syntax and semantics labor across buffers and productions, and some of these decisions could have been made differently. However, the resulting model is (we hope) pedagogically accessible, and is in keeping with most of the received wisdom about the properties of the human processor in the psycholinguistic literature (Marslen-Wilson 1973, 1975; Frazier and Fodor 1978; Gibson 1991, 1998; Tanenhaus et al. 1995; Steedman 2001; Hale 2011 among many others). While this received wisdom focuses mainly on the syntactic aspects of real-time language comprehension, the simplest way to extend it to semantics is to assume that it applies in basically the same form and to the same extent (if possible).

Specifically, the human processor, and our model of it, is incremental since syntactic parsing and semantic interpretation do not lag significantly behind the perception of individual words. It is also predictive since the processor forms explicit syntactic and semantic representations of words and phrases that have not yet been heard. Finally, it satisfies the competence hypothesis because the incremental interpretation process requires the recovery of a grammar-based structural description on the syntax side, and of a meaning representation (DRS) on the semantic side.

Furthermore, the syntax and semantics parsing process we implement formalizes fairly directly the general view of sentence comprehension summarized in Gibson (1998, p. 11):

> Sentence comprehension involves integrating new input words into the currently existing syntactic and discourse structure(s). Each integration has a syntactic component, responsible for attaching structures together, such as matching a syntactic category prediction or linking together elements in a dependency chain. Each integration also has a semantic and discourse component which assigns thematic roles and adds material to the discourse structure. (Gibson 1998, p. 11)

Let us now turn to an example and show how our model incrementally interprets the sentence in (17/1a) above, namely:

A lawyer is in a cave.

Assume that the first word, i.e., the indefinite article *a*, has already been read and recognized as a determiner. At that point, the `"project: NP ==> Det N"` rule in (24) below is selected and fired.

(24)
```
parser.productionstring(name="project: NP ==> Det N", string="""        1
    =g>                                                                 2
    isa                parsing_goal                                     3
    task               parsing                                          4
    stack1             S                                                5
    stack2             =s2                                              6
    right_frontier     =rf                                              7
    parsed_word        =w                                               8
    dref_peg           =peg                                             9
    =retrieval>                                                         10
    isa                word                                             11
    cat                Det                                              12
    ==>                                                                 13
    =g>                                                                 14
    isa                parsing_goal                                     15
    task               move_peg                                         16
    stack1             N                                                17
    stack2             NP                                               18
    stack3             S                                                19
    stack4             =s2                                              20
    +imaginal>                                                          21
    isa                parse_state                                      22
    node_cat           NP                                               23
    daughter1          Det                                              24
    mother             =rf                                              25
    lex_head           =w                                               26
    +discourse_context>                                                 27
    isa                drs                                              28
    dref               =peg                                             29
    arg1               =peg                                             30
    ~retrieval>                                                         31
""")                                                                    32
```

First, note that:

- the lexical entry for the determiner *a* is assumed to be available in the retrieval buffer (lines 10–12 in (24));
- our current top goal is to parse an S (line 5);
- a fresh discourse referent index, a.k.a. `dref_peg`, is available in the goal buffer (line 9).

The value of the `dref_peg` feature is assigned to an ACT-R variable `=peg` that will be used in the cognitive actions triggered by this rule. The index `=peg` is fresh in the sense that no discourse referent with that index has been introduced up to this point. Thus, that index has never been part of the semantic representation up until now: it was never the argument of a predicate, it couldn't have served as the antecedent for a pronoun etc. The term 'peg' and its specific usage here originates in Vermeulen (1995).

Given these preconditions, the rule in (24) simultaneously builds:

- a syntactic representation in the `imaginal` buffer (lines 21–26), and
- a semantic representation in the `discourse_context` buffer (lines 27–30).

In the `imaginal` buffer, we build the unary branching node NP dominating the Det node, which in turn dominates the terminal *a*.

In the `discourse_context` buffer, we start a new DRS that will be further specified by the upcoming N (*lawyer*). This DRS introduces a new discourse referent with the index `=peg`, and also requires this discourse referent to be the first argument (`arg1`) of the still-unspecified predicate contributed by the upcoming N—and also the first argument of the subsequent VP, as we will soon see.

In the more familiar format, the DRS contributed by rule (24) to the `discourse_context` buffer is provided in (25) below. Note that the value of the `=peg` index is specified as 1 since this is the first discourse referent introduced in the sentence.

(25) DRS contributed by the indefinite determiner *a*:

AVM format:

$$\begin{bmatrix} \text{ISA:} & \text{drs} \\ \text{DREF:} & 1 \\ \text{ARG1:} & 1 \end{bmatrix}$$

DRS format:

$$\begin{array}{|c|}\hline \textit{1} \\ \hline \text{STILL- UNSPECIFIED- PREDICATE}(1) \\ \hline \end{array}$$

The semantic part of the production rule in (24)—and the way it sets up the interpretation context for the upcoming N and VP—is very similar to the meaning assigned to the singular indefinite determiner *a* in dynamic semantic frameworks. For example, the indefinite *a* is associated with the following kind of meaning representation in Compositional DRT[6]:

(26) $a^1 \rightsquigarrow \lambda P'_{\mathbf{et}}.\lambda P_{\mathbf{et}}.\ [1];\ P'(1);\ P(1)$

The subscripted type **et** in (26) ensures that P' and P are (dynamic) properties. These properties will be contributed by the upcoming N *lawyer* and the subsequent VP *is in a cave*. The new discourse referent 1 introduced by the indefinite is superscripted on the indefinite *a* itself, and it is marked as newly introduced in the semantic representation by square brackets [1]. The semicolon '$;$' is dynamic conjunction (familiar from imperative programming languages like C).

Technically, the meaning representation in (26) is a function-denoting term in (classical, static) many-sorted simply-typed higher-order logic. However, it can be paraphrased as a sequence of update instructions. Specifically, the indefinite determiner a^1 'says' that:

- once two properties P' and P are given to me in this specific order ($\lambda P'_{\mathbf{et}}.\lambda P_{\mathbf{et}}.$),
- I will introduce a new discourse referent/'variable' ([1]),
- then ($;$),
- I will check that the discourse referent satisfies the first property ($P'(1)$),
- and then ($;$),
- I will check that the discourse referent satisfies the second property ($P(1)$).

We will see that our processing model follows this semantic 'recipe' fairly closely, but it distributes it across:

[6]See Muskens (1996), and also the detailed discussion and extensive examples in Brasoveanu (2007, Chap. 3).

i. several distinct productions in procedural memory,
ii. several distinct chunks sequentially updated and stored in the `discourse_context` buffer, and
iii. specific patterns of information flow, that is, feature-value flow, between the `g` (goal), `imaginal` and `discourse_context` buffers.

In addition to adding chunks to the `imaginal` and `discourse_context` buffers, the production rule in (24) above also specifies a new task in the goal buffer, which is to `move_peg` (line 16 in (24)). This will trigger a rule that advances the 'fresh discourse referent' index to the next number. In our case, it will advance it to 2. This way, we ensure that we have a fresh discourse referent available for any future expression, e.g., another indefinite, that might introduce one.

We also have a new stack of expected syntactic categories in the goal buffer (lines 17–19 in (24)). We first expect an N, at which point we will be able to complete the subject NP. Once that is completed, we can return to our overarching goal of parsing an S.

We will not discuss the family of `move_peg` rules. The full code is linked to in the appendix to this chapter—see Sect. 8.6.2. Instead, we will simply assume that the discourse reference peg has been advanced and discuss the `"project and complete: N"` rule in (27) below. This is the production rule that is triggered after the word *lawyer* is read and its lexical entry is retrieved from declarative memory.

(27)
```
parser.productionstring(name="project and complete: N", string="""
    =g>
    isa             parsing_goal
    task            parsing
    stack1          N
    stack2          =s2
    stack3          =s3
    stack4          =s4
    right_frontier  =rf
    parsed_word     =w
    =retrieval>
    isa             word
    cat             N
    pred           =p
    ?discourse_context>
    buffer          full
    ==>
    =g>
    isa             parsing_goal
    stack1          =s2
    stack2          =s3
    stack3          =s4
    stack4          None
    +imaginal>
    isa             parse_state
    node_cat        NP
    daughter1       Det
    daughter2       N
    lex_head        =w
    mother          =rf
    =discourse_context>
    isa             drs
    pred           =p
    ~retrieval>
""")
```

The `"project and complete: N"` rule requires the lexical entry for an N (*lawyer*, in our case) to be in the retrieval buffer (lines 11–14 in (27)). It also requires the top of the goal stack, that is, the most immediate syntactic expectation, to be N (line 5). Finally, the rule checks that there is a DRS in the `discourse_context` buffer (lines 15–16). This is the DRS contributed by the preceding determiner *a*, and which will be further updated by the noun *lawyer*.

Once all these preconditions are satisfied, the `"project and complete: N"` rule triggers a several actions that update the current parse state, i.e., the current syntactic and semantic representations.

First, a new part of the tree is added to the `imaginal` buffer (lines 24–30): this is the binary branching node NP with two daughters, a Det on the left branch and an N on the right.

Second, the DRS in the `discourse_context` buffer that was introduced by the determiner *a* is updated with a specification of the `pred` feature (lines 31–33). Specifically, the predicate `=p` on line 33 is the one contributed by the lexical entry of the word *lawyer*, which is currently available in the `retrieval` buffer (see line 14).

After the `"project and complete: N"` rule in (27) above fires, the DRS in the `discourse_context` buffer becomes:

(28) DRS in `discourse_context` buffer after the N (*lawyer*) update:
AVM format: DRS format:

$$\begin{bmatrix} \text{ISA:} & \text{drs} \\ \text{DREF:} & 1 \\ \text{PRED} & \text{LAWYER} \\ \text{ARG1:} & 1 \end{bmatrix} \qquad \begin{array}{|c|}\hline \mathit{1} \\ \hline \text{LAWYER}(1) \\ \hline \end{array}$$

Note that at this point, the `discourse_context` buffer holds SUB-DRS$_1$ of the MAIN-DRS in (18/21).

With the subject NP *a lawyer* now completely parsed, both syntactically and semantically, we have the full left corner of the 'S → NP VP' grammar rule, so we can trigger it. The corresponding production rule `"project and complete: S ==> NP VP"` is provided in (29) below.

(29)
```
parser.productionstring(name="project and complete: S ==> NP VP",
string="""
    =g>
    isa             parsing_goal
    task            parsing
    stack1          NP
    stack2          S
    stack3          =s3
    stack4          =s4
    =discourse_context>
    isa             drs
    dref            =d
    ==>
    =g>
    isa             parsing_goal
    stack1          VP
    stack2          =s3
    stack3          =s4                          18
    right_frontier  VP                           19
    arg_stack       =d                           20
    +imaginal>                                   21
    isa             parse_state                  22
    node_cat        S                            23
    daughter1       NP                           24
    daughter2       VP                           25
""")                                             26
```

This production rule eagerly discharges the expectations for an NP and an S from the goal stack (lines 6–7 in (29)) and replaces them both with a VP expectation (line 16). At the same time, a new part of the syntactic tree is built in the `imaginal` buffer, namely the top node S and its two daughters NP and VP (lines 21–25).

Finally, one important semantic operation happens in this rule, namely the transfer of the discourse referent `=d` from the `discourse_context` buffer (line 12) to the top of the argument stack `arg_stack` in the goal buffer (line 20). This operation effectively takes the discourse referent 1 introduced by the subject NP *a lawyer* and makes it the first argument of the VP we are about to parse. That is, this is how the cognitive process implements the final $P(1)$ semantic operation/update in formula (26) above.

We can now proceed to parsing the copula *is*, which we take to be semantically vacuous for simplicity. As the rule in (30) shows, the copula (once read and lexically retrieved) simply introduces a new part of the syntactic tree: the VP node with a Vcop (copular verb) left daughter and a PP right daughter.

(30)

```
parser.productionstring(name="project and complete: VP ==> Vcop PP",
string="""
    =g>
    isa             parsing_goal
    task            parsing
    stack1          VP
    parsed_word     =w
    right_frontier  =rf
    =retrieval>
    isa             word
    cat             Vcop
    ==>
    =g>
    isa             parsing_goal
    stack1          PP
    right_frontier  PP
    +imaginal>
    isa             parse_state
    mother          =rf
    node_cat        VP
    daughter1       Vcop
    daughter2       PP
    lex_head        =w
    ~retrieval>
""")
```

We can now move on to the preposition *in*. Once this preposition is read and lexically retrieved, the rule `"project and complete: PP ==> P NP"` in (31) below is triggered.

(31)

```
parser.productionstring(name="project and complete: PP ==> P NP",    1
string="""                                                            2
    =g>                                                               3
    isa             parsing_goal                                      4
    task            parsing                                           5
    stack1          PP                                                6
    parsed_word     =w                                                7
    arg_stack       =a                                                8
    right_frontier  =rf                                               9
    =retrieval>                                                       10
    isa             word                                              11
    cat             P                                                 12
    pred            =p                                                13
    ==>                                                               14
    =g>                                                               15
    isa             parsing_goal                                      16
    stack1          NP                                                17
    +imaginal>                                                        18
    isa             parse_state                                       19
    mother          =rf                                               20
    node_cat        PP                                                21
    daughter1       P                                                 22
    daughter2       NP                                                23
    lex_head        =w                                                24
    +discourse_context>                                               25
    isa             drs                                               26
    arg1            =a                                                27
    pred            =p                                                28
    ~retrieval>                                                       29
""")                                                                  30
```

The preconditions of the rule in (31) are the following: (i) the top of the goal stack is a PP (line 6), (ii) the argument stack stores the discourse referent `=a` contributed by the subject NP (line 8), and (iii) the retrieval buffer contains the lexical entry of the preposition, specifically the predicate `=p` contributed by it (line 13). If these preconditions are satisfied, the following cognitive actions are triggered:

- replace the PP at top of the goal stack with an NP (line 17); this is the NP that we expect the preposition to have as its complement;
- build a new part of the syntactic tree in the `imaginal` buffer, with PP as the mother node and daughters P and NP (lines 18–24);
- add a new DRS to the `discourse_context` buffer, the predicate of which is the binary relation `=p` contributed by the preposition, and the first argument of which is the subject discourse referent `=a` (lines 25–28);
- finally, flush the retrieval buffer (line 29).

While the syntactic components of the PP rule in (31) are specific to prepositions, the semantic part is more general: this is how binary relations (transitive verbs like *devour*, relational nous like *friend* or *aunt* etc.) are all supposed to work. They contribute their predicate `=p` to a new DRS and they specify the first argument of their binary relation to be the discourse referent `=a` contributed by the subject NP, as shown in (32) below.

Note that the DRS has an empty universe because no new discourse referents are introduced. Also, the second argument of the IN binary relation is still unspecified (symbolized by '_') because the location NP *a cave*, which will provide that argument, has not yet been read and interpreted.

(32) DRS in `discourse_context` buffer after the P (*in*) update:
AVM format: DRS format:

$$\begin{bmatrix} \text{ISA:} & \text{drs} \\ \text{PRED} & \text{IN} \\ \text{ARG1:} & 1 \end{bmatrix} \qquad \begin{array}{|c|} \hline \phantom{\text{IN}(1,\_)} \\ \hline \text{IN}(1, \_) \\ \hline \end{array}$$

We can now move on to the location NP *a cave*. Once the indefinite determiner *a* is read and lexically retrieved, the `"project and complete: NP ==> Det N"` rule in (33) below is triggered. Note that this rule is different from the `"project: NP ==> Det N"` in (24) above. The earlier `"project NP"` rule is triggered for subjects, that is, for positions where an NP is not already expected, i.e., it is not already present at the top of the goal stack.

The `"project and complete NP"` rule is triggered for objects of prepositions, transitive verbs etc. since these are *expected* NPs. By expected NPs, we mean that an NP is already present at the top of the goal stack when the determiner is read, as shown on line 6 in (33) below.

(33)
```
parser.productionstring(name="project and complete: NP ==> Det N",     1
string="""                                                             2
    =g>                                                                3
    isa                parsing_goal                                    4
    task               parsing                                         5
    stack1             NP                                              6
    right_frontier     =rf                                             7
    parsed_word        =w                                              8
    dref_peg           =peg                                            9
    =retrieval>                                                        10
    isa                word                                            11
    cat                Det                                             12
    ?discourse_context>                                                13
    buffer             full                                            14
    ==>                                                                15
    =g>                                                                16
    isa                parsing_goal                                    17
    task               move_peg                                        18
    stack1             N                                               19
    +imaginal>                                                         20
    isa                parse_state                                     21
    node_cat           NP                                              22
    daughter1          Det                                             23
    mother             =rf                                             24
    lex_head           =w                                              25
    =discourse_context>                                                26
    isa                drs                                             27
    dref               =peg                                            28
    arg2               =peg                                            29
    +discourse_context>                                                30
    isa                drs                                             31
    arg1               =peg                                            32
    ~retrieval>                                                        33
""")                                                                   34
```

The syntactic contribution of the rule in (33) is simply adding an NP node with a Det daughter to the `imaginal` buffer (lines 20–25). The semantic contribution is two-fold.

First, the DRS introduced by the preposition *in* is further specified by introducing a new discourse referent and setting the second argument slot of the IN predicate to this newly introduced discourse referent (lines 26–29). The resulting DRS, provided in (34) below, is the same as SUB-DRS$_3$ in (18/21) above, with the modification that the new discourse referent 2 is introduced in this sub-DRS rather than in SUB-DRS$_2$. We return to this issue below.

At this point, we have completely assembled (a version of) SUB-DRS$_3$ in the `discourse_context` buffer.

(34) DRS in `discourse_context` buffer after the first part of the update with the indefinite determiner (*a*), which further specifies the DRS introduced by the preposition *in*:

AVM format:

$$\begin{bmatrix} \text{ISA:} & \text{drs} \\ \text{DREF:} & 2 \\ \text{PRED} & \text{IN} \\ \text{ARG1:} & 1 \\ \text{ARG2:} & 2 \end{bmatrix}$$

DRS format:

$$\begin{array}{|c|} \hline \mathit{2} \\ \hline \text{IN}(1, 2) \\ \hline \end{array}$$

The second semantic contribution made by rule (33) above is the addition of a new DRS to the `discourse_context` buffer (lines 30–32) that will be our location DRS, i.e., SUB-DRS$_2$ in (18/21). We simply specify that the newly introduced discourse referent 2 is the first argument of a yet unspecified predicate, to be specified by the upcoming noun *cave*. This DRS is provided in (35) below.

(35) DRS in `discourse_context` buffer after the second part of the update with the indefinite determiner (*a*):

AVM format:

$$\begin{bmatrix} \text{ISA:} & \text{drs} \\ \text{ARG1:} & 2 \end{bmatrix}$$

DRS format:

$$\begin{array}{|c|} \hline \\ \hline \text{STILL- UNSPECIFIED- PREDICATE}(2) \\ \hline \end{array}$$

Unlike SUB-DRS$_2$ in (18/21), the DRS in (35) has an empty universe, since we introduced discourse referent 2 in the DRS previously stored in the `discourse_context` buffer. For the simple example at hand (*A lawyer is in a cave*), the decision to introduce discourse referent 2 in SUB-DRS$_3$ (34) rather than in SUB-DRS$_2$ (35) is inconsequential: the ultimate semantic representation, i.e., merged DRS, will have the same form as in (19) above.

We decided to go with this way of encoding the introduction of object NP discourse referents rather than with the more standard semantic representation in (18/21) just to show that it is possible. Also, from a processing perspective, it is simpler and more natural to introduce a discourse referent at the earliest point in the left-to-right incremental interpretation process where the referent appears as the argument of a predicate.

This decision, however, unlike the standard formal semantics decision in (18/21), makes incorrect predictions with respect to the quantifier scope potential of the location NP. This is truth-conditionally inconsequential in our example because the subject NP is also an existential. But if the subject NP had been a universal quantifier, e.g., *every lawyer*, we would incorrectly predict that the location NP *a cave* can have only narrow scope relative to that quantifier. This is because its scope is tied to the point of discourse referent introduction, namely the predicate *in*. While this narrow-scope prediction is correct for so-called 'semantically incorporated' NPs, e.g., bare plurals (see Carlson 1977, 1980; Farkas and de Swart 2003 among others for more discussion), it is not generally correct for *bona fide* indefinite NPs, which are scopally unrestricted (see, for example, Brasoveanu and Farkas 2011 for a relatively recent discussion).

The semantic representation that makes correct scopal predictions is easy to obtain in this case: the discourse referent introduction on line 28 of (33) simply needs to be moved further down, e.g., immediately after line 31. In general, however, such a change might be non-trivial.

What we have here is an instance of a more general phenomenon: incremental processing order and semantic evaluation order do not always coincide (see Vermeulen 1994; Milward and Cooper 1994; Chater et al. 1995 for insightful discussions of this issue). We believe that the Bayes+ACT-R+DRT framework introduced in this book provides the right kind of formal infrastructure to systematically investigate this type of discrepancies between processing and semantic evaluation order. A detailed systematic investigation of such discrepancies will likely constrain in non-trivial ways both competence-level semantic theories and semantically-informed processing theories and models.

Once the final noun *cave* is read and lexically retrieved, a second application of the `"project and complete: N"` rule in (27) above is triggered. The semantic contribution of this rule is to further specify the DRS in (35) by adding the predicate. The resulting DRS is provided in (36) below, and is the final version of SUB-DRS$_2$ produced by our syntax/semantics parser.

(36) DRS in `discourse_context` buffer after the second part of the update with the indefinite determiner (*a*):
AVM format: DRS format:

$$\begin{bmatrix} \text{ISA:} & \text{drs} \\ \text{PRED:} & \text{CAVE} \\ \text{ARG1:} & 2 \end{bmatrix} \qquad \begin{array}{|c|}\hline \\ \hline \text{CAVE}(2) \\ \hline \end{array}$$

8.4 Semantic (Truth-Value) Evaluation as Memory Retrieval, and Fitting the Model to Data

At this point, our model has completely parsed the test sentence *A lawyer is in a cave*. With the DRS for this sentence in hand, we can move to establishing whether the sentence was studied in the training phase or not.

To put it differently, the training phase presented a set of facts, and now we have to evaluate whether the test sentence is true or not relative to those facts. That is, we view *semantic (truth-value) evaluation as memory retrieval.*[7]

Thus, the bigger picture behind our DRT-based model of the fan effect is that the process of semantic interpretation proceeds in two stages, similar to the way interpretation proceeds in DRT.

In the initial stage, we construct the semantic representation/DRS/mental discourse model for the current sentence. In DRT, specifically in Kamp and Reyle (1993), this stage involves a step-by-step transformation of a complete syntactic representation of the sentence into a DRS by means of a series of construction-rule applications. In our processing model, this stage consists of applying an eager, left-corner, syntax-and-semantics parser to the current test sentence.

In the second stage, we evaluate the truth of this DRS/mental model by connecting it to the actual, 'real-world' model, which is our background database of facts stored in declarative memory. In DRT, the second stage involves constructing an embedding function (a partial variable assignment) that verifies the DRS relative to the model. In our processing model, truth/falsity evaluation involves retrieving—or failing to retrieve—a fact from declarative memory that has the same structure as the DRS we have just constructed.

Let us turn now to how exactly we model semantic evaluation as memory retrieval. First, we assume that all the facts studied in the training phase of the fan experiment are stored in declarative memory before we even start parsing the test sentence. All the relevant code is linked to in the appendix of this chapter (see Sect. 8.6.1). We will list here only the lawyer-cave fact, together with the type declarations for main DRSs and (sub)DRSs:

(37)
```
actr.chunktype("drs", "dref pred arg1 arg2")                                  1
actr.chunktype("main_drs", "subdrs1 subdrs2 subdrs3")                         2
                                                                              3
lawyer = actr.makechunk(typename="drs", arg1=1, dref=1, pred='LAWYER')        4
cave = actr.makechunk(typename="drs", arg1=2, pred='CAVE')                    5
in_relation = actr.makechunk(typename="drs", arg1=1, arg2=2, dref=2,          6
                             pred='IN')                                       7
dm.add(actr.makechunk(typename="main_drs", subdrs1=lawyer,                    8
                      subdrs2=cave, subdrs3=in_relation))                     9
```

The three sub-DRSs are declared on lines 4–7 in (37) with the `makechunk` method, and are assigned to three Python3 variables (note: not ACT-R variables). We can then use these three Python3 variables as subparts/values inside the main DRS encoding the lawyer-cave fact (lines 8–9). Finally, we add the entire fact to declarative memory with the `dm.add` method (also lines 8–9).

The declarative memory module of our model is loaded with all the facts listed in (1) before the parsing process for a test sentence even starts. When the parse of a test sentence is completed, we have the location DRS, that is, SUBDRS$_2$ in the `discourse_context` buffer. For the sentence we parsed in the previous section, that location DRS is provided in (36) above.

[7]See Budiu and Anderson (2004) for a variety of potential applications of such a proposal.

To set up the proper spreading-activation configuration for the fan-effect experiment, we have to have all three sub-DRSs that are produced during parsing in the goal buffer. Since only the location DRS is available, we have to recall the person DRS and the *in* DRS listed in (31) and (34) above so that we can add them to the goal buffer.

We start by recalling the *in*-relation DRS with the rule in (38) below. This rule is triggered as soon as the test sentence has been completely parsed: the `task` is still `parsing` (line 5), but the goal stack is empty (the top of the stack is `None`—line 6). For good measure, we also check that the last DRS that was constructed (the location DRS) is still in the `discourse_context` buffer (lines 7–8). Assuming these pre-conditions are satisfied, we place a retrieval request for a DRS—any DRS (lines 13–15). Because the *in* DRS is the one that was most recently added/harvested to declarative memory, it has the highest activation, and this is the DRS we will end up retrieving.

(38)
```
parser.productionstring(name="recall in_relation sub-DRS",     1
string="""                                                      2
    =g>                                                         3
    isa              parsing_goal                               4
    task             parsing                                    5
    stack1           None                                       6
    ?discourse_context>                                         7
    buffer           full                                       8
    ==>                                                         9
    =g>                                                        10
    isa              parsing_goal                              11
    task             recall_person_subdrs                      12
    +retrieval>                                                13
    isa              drs                                       14
    arg1             ~None                                     15
""")                                                           16
```

The retrieval request, however, doesn't simply state that we should retrieve a chunk of type `drs` (line 14 in (38) above), but goes ahead and requires a non-empty `arg1` feature (line 15). This extra-specification seems redundant, but it is in fact necessary: recall that the identity of a chunk (and of a chunk type) is given by its list of features, not by the name we give to the type. The type name listed as the value of the feature `isa` is simply a convenient abbreviation for us as modelers. Therefore, placing a retrieval request with a cue consisting only of `isa drs` is tantamount to placing a retrieval request for *any* chunk in declarative memory, whether it is of type `drs`, or `main_drs`, or indeed any of the other types we use (`parsing_goal`, `parse_state` and `word`—see the beginning of Sect. 8.6.1 for all chunk-type declarations). To make sure we actually place a retrieval request for a DRS, we need to list one of its distinguishing features, and we choose the `arg1` feature here.

In addition to placing a retrieval request for the DRS with the highest activation, which is the *in* DRS, the rule in (38) also updates the task in the goal buffer to `recall_person_subdrs`. This sets the cognitive context up for the next recall rule, provided in (39) below. The preconditions of this rule include a full `discourse_context` buffer, where the location DRS is still stored (lines 9–10 in (39)), and a full `retrieval` buffer, which stores the successfully retrieved *in* DRS (lines 7–8).

(39)
```
parser.productionstring(name="recall person sub-DRS",     1
string="""                                                2
    =g>                                                   3
    isa             parsing_goal                          4
    task            recall_person_subdrs                  5
    stack1          None                                  6
    =retrieval>                                           7
    isa             drs                                   8
    ?discourse_context>                                   9
    buffer          full                                  10
    ==>                                                   11
    =g>                                                   12
    isa             parsing_goal                          13
    task            match_subdrs                          14
    expected3       =retrieval                            15
    ?retrieval>                                           16
    recently_retrieved False                              17
    +retrieval>                                           18
    isa             drs                                   19
    arg1            ~None                                 20
""")                                                      21
```

Once these preconditions are satisfied, the rule will take the *in* DRS in the retrieval buffer and store it in the goal buffer under an `expected3` feature (line 15). Encoding the *in* DRS as the value of the `expected3` feature indicates that this DRS is expected to be SUBDRS$_3$ of the verifying fact for the test sentence we just finished parsing.[8]

Importantly, setting the *in* DRS as the value of a feature in the goal buffer ensures that there is spreading activation from this sub-DRS to all the main DRSs/facts in declarative memory that contain it. This is essential for appropriately modeling the fan effect.

More generally, at this point in the cognitive process, our goal is to add all three parsed sub-DRSs to the goal buffer (we still have to add the location and person sub-DRSs), so that we have spreading activation from each of them to main DRSs in declarative memory. Once we achieve this state for the goal buffer, we will be able to semantically evaluate the parsed test sentence. That is, we will place a retrieval request for a main DRS that can verify it.

In order to store all the sub-DRSs of the parsed test sentence in the goal buffer, we have to place one final retrieval request to recall the person DRS (lines 16–20 in (39)), which was the first semantic product of our process of parsing the test sentence. Crucially, this request is modulated by the constraint of not retrieving a DRS that was recently retrieved (lines 16–17 in (39)). This additional constraint is essential: had it not been present, the actual retrieval request for a DRS (any DRS) on lines 18–20 would end up retrieving the *in* DRS all over again. The reason for this is that the *in* DRS was the most activated to begin with, and its most recent retrieval triggered by the rule in (38) above boosted that activation even further.

The constraint to retrieve something that was not recently retrieved (lines 18–20 in (39)) is not *ad hoc*. This type of constraint is necessary in a wide variety of processes involving refractory periods after a cognitive action, and it is modeled in ACT-R (and `pyactr`) in terms of Pylyshyn's FINSTs (fingers of instantiation).

[8]Of course, the number 3 in `expected3` has no actual interpretation, it is only convenient for us as modelers so that we can more easily keep track of the number of DRSs used for spreading activation.

The original motivation for the FINST mechanism in early visual perception is summarized in Pylyshyn (2007):

> When we first came across this problem [of object identity tracking that occurs automatically and generally unconsciously as we perceive a scene] [i.e., the need to keep track of things without a conceptual description using their properties] it seemed to us that what we needed is something like an elastic finger: a finger that could be placed on salient things in a scene so we could keep track of them as being the same token individuals while we constructed the representation, including when we moved the direction of gaze or the focus of attention. What came to mind is a comic strip I enjoyed when I was a young comic book enthusiast, called Plastic Man. It seemed to me that the superhero in this strip had what we needed to solve our identity-tracking or reidentification problem. Plastic Man would have been able to place a finger on each of the salient objects in the figure. Then no matter where he focused his attention he would have a way to refer to the individual parts of the diagram so long as he had one of his fingers on it. Even if we assume that he could not detect any information with his finger tips, Plastic Man would still be able to think 'this finger' and 'that finger' and thus he might be able to refer to individual things that his fingers were touching. This is where the playful notion of FINgers of INSTantiation came on the scene and the term FINST seems to have stuck. (Pylyshyn 2007, pp. 13–14)

ACT-R/`pyactr` uses the FINST mechanism in the visual module, and also generalizes it to other perception modules, including declarative memory, which, as Anderson (2004) puts it, is the perception module for the past.

Specifically, the declarative module maintains a record of the n most recent chunks that have been retrieved ($n = 4$ by default). These are the chunks indexed by a FINST. The FINST remains on them for a set amount of time (3 s by default), after which the FINST is removed and the chunk is no longer marked as recently retrieved.

Retrieval requests can specify whether the declarative memory search should be confined to chunks that are—or are not—indexed by a FINST. In rule (39), we require the DRS retrieval request on lines 18–20 to target only DRSs that have not been recently retrieved (lines 16–17), i.e., the retrieval request targets only non-FINSTed DRSs.

Let us summarize the state of the ACT-R mind, i.e., the cognitive context, immediately after the production rule in (39) fires:

- the `discourse_context` buffer still contains the location DRS;
- the `expected3` feature in the goal buffer (for expected SUBDRS$_3$) has the *in* DRS as its value; this *in* DRS has just been retrieved from declarative memory;
- finally, we have just placed a retrieval request for the person DRS, restricted to non-FINSTed DRSs to avoid retrieving the *in* DRS all over again.

Once this retrieval request is completed, we are ready to fire the rule in (40) below. We ensure that this rule fires only after the retrieval request is completed by means of the precondition on lines 8–9, which requires the `retrieval` buffer to be in a non-`busy`/`free` state.

(40)
```
parser.productionstring(name="recall main DRS by person sub-DRS",     1
string="""                                                            2
    =g>                                                               3
    isa             parsing_goal                                      4
    task            match_subdrs                                      5
    =retrieval>                                6
    isa             drs                        7
    ?retrieval>                                8
    state           free                       9
    =discourse_context>                        10
    isa             drs                        11
    ==>                                        12
    =g>                                        13
    isa             parsing_goal               14
    task            match_main_drs             15
    expected1       =retrieval                 16
    expected2       =discourse_context         17
    +retrieval>                                18
    isa             main_drs                   19
    subdrs1         =retrieval                 20
""", utility=0.5)                              21
```

When its preconditions are met, the rule in (40) finishes setting up the correct environment for spreading activation required by the fan experiment. This means that the three sub-DRSs that we constructed during the incremental interpretation of our test sentence have to be added to the goal buffer.

The previous rule already set the *in* DRS as the value of the `expected3` feature of the goal buffer. We now set the person DRS, who has just been retrieved, as the value of the `expected1` feature of the goal buffer (line 16 in (40)), and the location DRS, who has been in the `discourse_context` buffer since the end of the incremental parsing process, as the value of the `expected2` feature (line 17 in (40)).

All three sub-DRSs in the goal buffer spread activation to declarative memory, giving an extra boost to the correct fact acquired during the training phase of the experiment.

We are now ready to semantically evaluate the truth/falsity of the test sentence. That is, we are ready to place a memory retrieval request for a `main_drs`/fact in declarative memory that makes the test sentence true.

Following the ACT-R account of the fan effect proposed in Anderson and Reder (1999), which we discussed in Sect. 8.1 above, we can place this memory retrieval request via the person sub-DRS, or via the location sub-DRS (see the discussion of foil identification in particular). On lines 18–20 of the rule in (40), we place the retrieval request with the person sub-DRS as the cue (the person sub-DRS is in the retrieval buffer).

Note the difference between the role that three sub-DRSs play as values that spread activation and the role that the person DRS plays as a retrieval cue/filter. Using the person DRS as a filter eliminates any main DRS that does not have the matching person DRS. In contrast, spreading activation only boosts the activation of the main DRSs in declarative memory that happen to store the same person, location and *in* relation, but it does not filter anything out. This means that recalling the main DRS by the person sub-DRS must result in retrieving one of those main DRSs that match the person sub-DRS, but it is possible that the recalled DRS will have a mismatching location sub-DRS, e.g., if spreading activation plays a very small role.

We can alternatively place the retrieval request with the location DRS as cue, as shown on lines 19–21 of the rule in (41) below. This rule is identical to the rule in (40) except that the retrieval cue is now the location DRS stored in the `discourse_context` buffer.

(41)

```
recall_by_location = parser.productionstring(                1
name="recall main DRS by location sub-DRS",                  2
string="""                                                   3
    =g>                                                      4
    isa             parsing_goal                             5
    task            match_subdrs                             6
    =retrieval>                                              7
    isa             drs                                      8
    ?retrieval>                                              9
    state           free                                     10
    =discourse_context>                                      11
    isa             drs                                      12
    ==>                                                      13
    =g>                                                      14
    isa             parsing_goal                             15
    task            match_main_drs                           16
    expected1       =retrieval                               17
    expected2       =discourse_context                       18
    +retrieval>                                              19
    isa             main_drs                                 20
    subdrs2         =discourse_context                       21
""")                                                         22
```

The rules in (40) and (41) are alternatives to each other, and either of them can be selected to fire. When running simulations of this model, we can simply leave it to `pyactr` to select if the retrieval of the `main_drs` should be in terms of the person sub-DRS (hence, rule (40)) or in terms of the location sub-DRS (hence, rule (41)). In the long run, each of these two rules will be selected about half the time, and the resulting RTs for targets and foils will be the average of person and location based retrieval (with a small amount of noise).

But if we want to eliminate this source of noise altogether, we can force the model to choose one rule in one simulation, and the other rule in a second simulation, and average the results. This is the option we pursue here. To achieve this, we make use of production-rule utilities, which induce a preference order over rules that is used when the preconditions of multiple rules are satisfied in a cognitive state. In that case, the rule with the highest utility is chosen.

To average over the recall-by-person and recall-by-location rules, we set the utility of recall-by-person rule in (40) to 0.5 (see line 21 in (40)), and we assign the recall-by-location rule in (41) to a Python3 variable `recall_by_location` (line 1 in (41)). With these two things in place, we can update the utility of the `recall_by_location` rule either to 0, which is less than the 0.5 utility of the recall-by-person rule, or to 1, which is more than 0.5. In the first case, the recall-by-person rule in (40) fires. In the second case, the recall-by-location rule in (41) fires.

In our Bayesian model, we can run two simulations with these two utilities for the recall-by-location rule, and take their mean. This is what the Bayesian estimation code linked to in Sect. 8.6.4 at the end of the chapter actually does. The relevant bit of code is provided in (42) below for ease of reference.[9]

(42)

```
for i in range(2):
    recall_by_location["utility"] = i
    ...
    while True:
        ...
        if re.search("^KEY PRESSED: J", ...):
            ...
            if run_time:
                run_time += parser_sim.show_time()
                run_time = run_time/2
            else:
                run_time = parser_sim.show_time()
```

Once the retrieval request for the `main_drs` verifying the test sentence is completed, the rule in (43) below fires. This rule checks that the retrieved `main_drs` matches the DRS of the parsed test sentence with respect to the person and location DRSs stored in the goal buffer (lines 5–6 and 12–13). If this condition is met, the test sentence is declared true (it is part of the set of facts established in the training phase) and the `'J'` key is pressed (line 18–21). This concludes the simulation, so the goal buffer is flushed (line 17).

(43)

```
parser.productionstring(name="match found", string="""          1
    =g>                                                          2
    isa              parsing_goal                                3
    task             match_main_drs                              4
    expected1        =e1                                         5
    expected2        =e2                                         6
    ?retrieval>                                                  7
    state            free                                        8
    buffer           full                                        9
    =retrieval>                                                  10
    isa              main_drs                                    11
    subdrs1            =e1                                       12
    subdrs2            =e2                                       13
    =discourse_context>                                          14
    isa              drs                                         15
    ==>                                                          16
    ~g>                                                          17
    +manual>                                                     18
    isa              _manual                                     19
    cmd              'press_key'                                 20
    key              'J'                                         21
""")                                                             22
```

The model includes three other rules `"mismatch in person found"`, `"mismatch in location found"` and `"failed retrieval"`, provided in the code linked to at the end of the chapter in Sect. 8.6.2. These rules enable us to model foil retrieval and deal with cases in which a `main_drs` retrieval fails completely. But in this chapter, we focus exclusively on modeling target sentences, i.e., test sentences that were seen in the training phase, and for which the `main_drs` retrieval request succeeds.

[9]In general, it is common to have the model learn utilities rather than set them by hand. Utility learning is possible in ACT-R and `pyactr`. See Taatgen and Anderson (2002) among others for more discussion of utility learning.

The full model can be seen in action by running the script `run_parser_fan.py`, linked to in Sect. 8.6.3 at the end of the chapter. The model reads the sentence *A lawyer is in a cave* displayed on the virtual screen, incrementally interprets/parses it and checks that it is true relative to the database of facts/main DRSs in declarative memory.

The script `estimate_parser_fan.py` linked to in Sect. 8.6.4 embeds the ACT-R model into a Bayesian model and fits it to the experimental data from Anderson (1974). We do not discuss the code in detail since it should be fairly readable at this point: it is a variation on the Bayes+ACT-R models introduced in Chap. 7. Instead, we'll highlight the main features of the Bayesian model.

First, we focus on estimating four subsymbolic parameters:

- `"buffer_spreading_activation"` (`"bsa"` for short), which is the *W* parameter in Sect. 8.1 above;
- `"strength_of_association"` (`"soa"` for short), which is the *S* parameter in Sect. 8.1 above;
- `"rule_firing"` (`"rf"` for short), which is by default set to 50 ms;
- `"latency_factor"` (`"lf"` for short).

Our discussion in Sect. 8.1 above shows that we need to estimate the `"bsa"`, `"soa"` and `"lf"` parameters. We have also decided to estimate the `"rf"` parameter, instead of leaving it to its default value of 50 ms, because it is reasonable for it to be lower for the kind of detailed, complex language models we are constructing. In these models, multiple theoretically motivated rules are needed to fire rapidly in succession, and the total amount of time they can take is highly constrained by the empirical generalization that people take around 300 ms to read a word in an eye-tracking experiment, and they take roughly the same amount in a self-paced reading experiment.

We might be able to revert to the ACT-R default of 50 ms per rule firing if we take advantage of production compilation, which is a process by which multiple production rules and retrieval requests can be aggregated into a single rule. Production compilation is one of the effects of skill practice, and it is very likely that incremental interpretation of natural language, which is a highly practiced skill for adult humans, takes full advantage of it (see Chap. 4 in Anderson 2007 and, also, Taatgen and Anderson 2002 for more discussion of production compilation).

Production compilation is available in `pyactr` and we could make use of it in future developments of this model. However, production compilation will result in production rules that do not transparently reflect the formal syntax and semantics theories we assume here, making the connection between processing and formal linguistics more opaque.

It might very well be that more mature cognitive models of syntax and semantics will have to head in the direction of significant use of production compilation. However, we think that at this early point in the development of computationally explicit processing models for formal linguistics, it is more useful to see that established linguistic theories can be embedded in language processing ACT-R models in an easily recognizable fashion.

At the same time, we also want to show that the resulting processing models can fit experimental data well, and can provide theoretical insight into quantitatively-measured cognitive behavior.

To meet both of these desiderata, we need to depart from various default values for ACT-R subsymbolic parameters. In particular, for the semantic processing models in this chapter and the next one, we need to allow the `"rf"` parameter to vary and we need to estimate it. As we will soon see, the estimated value hovers around 10 ms, that is, it is significantly less than the ACT-R default. The need to lower ACT-R defaults when modeling natural language phenomena in a way that is faithful to established linguistic theories is a common thread throughout this book.

As shown in (44) below, the Bayesian model (i) sets up low information priors for these four parameters, (ii) runs the `pyactr` model to compute the likelihood of the experimental data in (4) above, and (iii) estimates the posterior distributions for these four parameters given the priors and the observed data. Links to the full code are provided in Sect. 8.6.4 at the end of the chapter.

(44)
```
fan_model = Model()

with fan_model:
    # Priors
    buffer_spreading_activation = HalfNormal("bsa", sd=2)
    strength_of_association = HalfNormal("soa", sd=4)
    rule_firing = HalfNormal("rf", sd=0.03)
    latency_factor = HalfNormal("lf", sd=0.2)
    # Likelihood
    pyactr_rt = actrmodel_latency(rule_firing, latency_factor,
                                  buffer_spreading_activation,
                                  strength_of_association)
    mu_rt = Deterministic('mu_rt', pyactr_rt)
    rt_observed = Normal('rt_observed', mu=mu_rt, sd=10, observed=RT)

with fan_model:
    # Compute posteriors
    step = pm.SMC(parallel=True)
    trace = pm.sample(draws=5000, step=step, njobs=1, cores=50)
```

The posterior estimates for the four parameters are provided in Fig. 8.1. The most notable one is the rule firing parameter, whose mean value is 11 ms rather than 50 ms. We will see that a similar value is necessary when we model cataphoric pronouns and presuppositions in the next chapter.

Note that the `Rhat` values for this model are below 1.1:

(45)
```
{'bsa': 1.0112865739232026,
 'soa': 1.0671138466090309,
 'rf': 1.0689415222560494,
 'lf': 1.0738886077317906,
 'mu_rt': array([1.00237602, 1.00518725, 1.00336533, 1.00518725, 1.00455244,
                 1.00423505, 1.00336533, 1.00423505, 1.00906647])}
```

As the plot of the posterior predictions of the model in Fig. 8.2 shows, the model is able to fit the data fairly well. There are slight discrepancies between the predictions of the model and the actual observations, which are due to variance in the data that our fan model ignores from the start. For instance, when we inspect the table in 4, we see that the mean reaction time to recognize targets with a fan of 3 for person and a fan of 1 for location was 1.22 s, while the mean reaction time to recognize targets

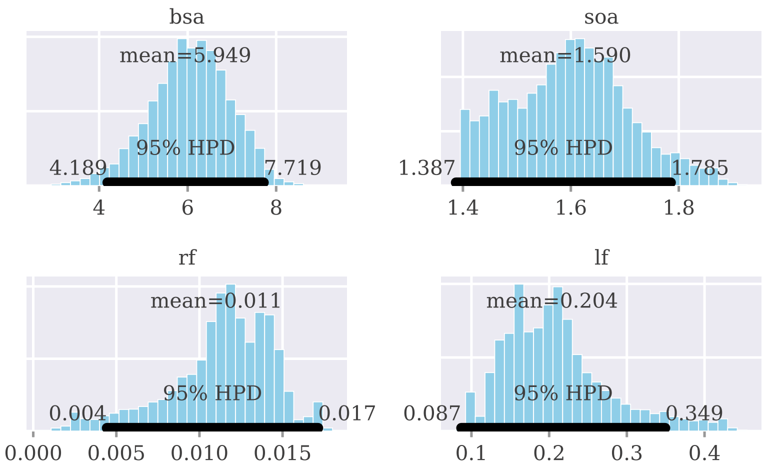


Fig. 8.1 Fan model estimates

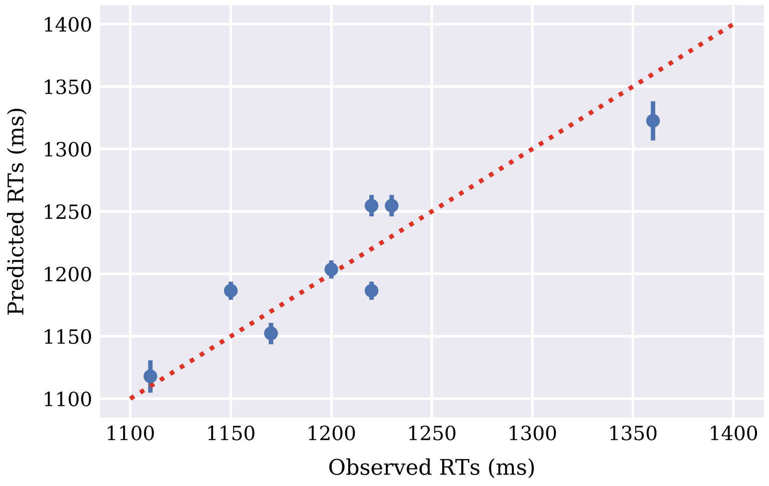


Fig. 8.2 Fan model: observed versus predicted RTs

with a fan of 3 for location and a fan of 1 for person was 1.15 s. This difference of 70 ms cannot be captured by our model, which treats the two cases as identical.

We see that the model captures a large portion of the variance in data, but there is room for improvement. The model can in principle be enhanced in various ways if contrasts like the one we just mentioned turn out to be genuine and robust rather than largely noise, which is our current simplifying assumption.

8.5 Model Discussion and Summary

In this chapter, we modeled the fan experiment in Anderson (1974), bringing together the ACT-R account in Anderson and Reder (1999) and formal semantics theories of natural language meaning and interpretation. The resulting incremental interpreter is the first one to integrate in a computationally explicit way (i) dynamic semantics in its DRT incarnation and (ii) mechanistic processing models formulated within a cognitive architecture.

In developing this cognitive model, we argued that the fan effect provides fundamental insights into the memory structures and cognitive processes that underlie semantic evaluation. By semantic evaluation, we mean the process of determining whether something is true or false relative to a database of known facts/DRSs, i.e., relative to a model in the sense of model-theoretic semantics.

Future directions for this line of research include investigating whether a partial match of known facts is considered good enough for language users in comprehension. This could provide an integrated account of a variety of interpretation-related phenomena: (i) Moses illusions (see Budiu and Anderson 2004 for a relatively recent discussion), (ii) the way we interpret sentences with plural definites like *The boys jumped in the pool*, where the sentence it true without every single boy necessarily jumping in the pool[10], and (iii) 'partial presupposition resolution' cases, where part of the presupposition is resolved and part of it is accommodated (see Kamp 2001a for an argument that this kind of mixture of resolution and accommodation seems to be the rule rather than the exception).

Spreading activation in general can be used for predictive parsing: words (more generally, linguistic information) in the previous context can predictively activate certain other words, i.e., induce expectations for other chunks of linguistic information.

Another direction for future research is reexamining the decisions we made when developing our incremental DRT parser that were not based on cognitive plausibility, but instead were made for pedagogical reasons—in an effort to ensure that the contributions made by semantic theories were still recognizable in the final syntax/semantics parser. These decisions led to unrealistic posterior estimates, e.g., 11 ms for rule firing, which is not a fully satisfactory outcome.

In addition, we oversimplified the model in various ways, again for pedagogical purposes. For example, we initiated the semantic evaluation process (the search for a matching fact in declarative memory) only when the sentence was fully parsed syntactically and semantically. This is unrealistic: parsing, disambiguation and semantic evaluation are most probably interspersed processes, and searches for matching facts/main DRSs in declarative memory are probably launched eagerly after every parsed sub-DRS, if not even more frequently. See Budiu and Anderson (2004) for a similar proposal, and for an argument that such an approach, coupled with a judi-

[10]We want to thank Margaret Kroll for bringing the connection between partial matches and plural definites to our attention.

cious use of spreading activation, might explain the preference to provide given (topic) information earlier in the sentence, and new (focused) information later.

Another oversimplification was the decision to add sub-DRSs to the goal buffer to initiate spreading activation only after the entire sentence was parsed. It is likely that some, possibly all, sub-DRSs are added to the goal buffer and start spreading activation to matching main DRSs as soon as they are parsed. It might even be that such sub-DRSs are added to the goal buffer and start spreading activation even before they are completely parsed. This might be the case for NPs with a relative clause, where the partial sub-DRS obtained by processing the nominal part before the relative clause is added to the goal buffer, and then it is revised once the relative clause is processed.

In sum, we just barely scratched the surface with the semantic processing model we introduced in this chapter. There are many semantic phenomena for which we have detailed formal semantics theories, but no similarly detailed and formalized theories on the processing side. We hope to have shown in this chapter that there is a rich space of such theories and models waiting to be formulated and evaluated. The variety of semantic phenomena to be investigated, the variety of possible choices of semantic frameworks and the variety of detailed processing hypotheses that can be formulated and computationally implemented offer a rich and unexplored theoretical and empirical territory.

The next chapter, which is the last substantial chapter of the book, extends the incremental interpreter we introduced here to account for the interaction between cataphoric pronouns/presuppositions and the (dynamic) semantics of two different sentential operators, conjunction and implication.

8.6 Appendix: End-to-End Model of the Fan Effect with an Explicit Syntax/Semantics Parser

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

8.6.1 File ch8/parser_dm_fan.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch8/parser_dm_fan.py.

8.6.2 File ch8/parser_rules_fan.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch8/parser_rules_fan.py.

8.6.3 File ch8/run_parser_fan.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch8/run_parser_fan.py.

8.6.4 File ch8/estimate_parser_fan.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch8/estimate_parser_fan.py.

Chapter 9
Semantics as a Cognitive Process II: Active Search for Cataphora Antecedents and the Semantics of Conditionals

In this chapter, we generalize our eager left-corner incremental interpreter to cover conditionals and conjunctions. We focus on the (dynamic) semantic contrast between conditionals and conjunctions because the interaction between these sentential operators and anaphora/cataphora provides a strong argument that semantic parsing needs to be incremental, eager and competence-based, as argued in Brasoveanu and Dotlačil (2015a).

An extreme, but clear way to state the main theoretical proposal made by this chapter is the contention that anaphora, and presupposition in general, are properly understood as *processing-level* phenomena that *guide and constrain memory retrieval processes associated with incremental interpretation*. That is, they guide and constrain the cognitive process of integration, or linking, of new and old semantic information.

Under the assumption that anaphora and presuppositions are components of memory retrieval processes associated with the real-time integration of semantic information, the intrusion of world knowledge in these processes, i.e., the "pragmatics of anaphora resolution and presupposition resolution," comes in naturally: world knowledge is stored in declarative memory, so it is natural for memory retrieval processes to be modulated by it.

Thus, the (most probably oversimplifying) hypothesis is that anaphora and presupposition have semantic effects, but anaphora and presupposition are not exclusively, or even primarily, semantics. The proper way to analyze them is as a part of the *processing component* of a broad theory of natural language interpretation.

This proposal is very close in spirit to the DRT account of presupposition proposed in van der Sandt (1992), Kamp (2001a, b), among others. Kamp (2001b), with its extended argument for and extensive use of *preliminary representations*—that is, meaning representations that explicitly include unresolved presuppositions—is a particularly close idea.

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_9

To see the connection between our proposal and the theory of presuppositions proposed in Kamp (2001b), consider the problem associated with the use of preliminary representations raised at the end of Kamp (2001b):

> "[S]emanticists of a model-theoretic persuasion may want to see a formal semantics of […] preliminary representations. […] [T]he possibility of such a semantics is limited. To define a syntax of preliminary representations […] which characterizes them as the expressions of a given representation formalism (or as data structures of a certain form) is not too difficult. Moreover, for those preliminary representations […] in which all presuppositions appear in the highest possible position, an intuitively plausible model-theoretic semantics can be stated without too much difficulty. But for representations with presuppositions in subordinate positions […] I very much doubt that one is to be had." (Kamp 2001b, 250–51)

The solution to this problem that we propose in this chapter is that we shouldn't even ask for a semantics of preliminary representations. This is a category error: preliminary representations are central to natural language interpretation, but they are not semantic representations: they are processing-level representations that support incremental interpretation mechanisms, and have semantic effects because of this.

The chapter is structured as follows. In Sect. 9.1, we discuss why the interaction of conditionals and cataphora provide a strong empirical argument for incremental interpretation *at the semantic level*. We also describe two experiments investigating the interaction between conditionals and pronominal cataphora on one hand (Sect. 9.1.1), and the interaction between conditionals and cataphoric presuppositions on the other hand (Sect. 9.1.2).

Section 9.2 sets up the general theoretical scene for the remainder of the chapter by arguing that mechanistic processing goals should be one of the core explanatory goals for formal semantics, and that a cognitive-architectural approach to natural language meaning and interpretation is one way to pursue that goal.

Section 9.3 moves on to the specifics of an ACT-R processing model for conditionals with a sentence-final *if*-clause that explicitly models their incremental interpretation. We show that the model qualitatively captures the interaction between conditionals and pronominal cataphora in a simple example.

Section 9.4 expands this model to capture the interaction between conditionals and cataphoric presuppositions for the items of the study reported in Sect. 9.1.2. We embed the resulting model in a Bayesian model, and show that it can quantitatively fit the experimental data fairly well. This section shows that the Bayes+ACT-R+DRT framework we introduced enables us to specify in a fully explicit way different competence-performance hypotheses about conditionals, and that experimental evidence from real-time experiments can be used to quantitatively compare alternative theories of conditionals and cataphora.

Finally, Sect. 9.5 concludes with a brief summary and an outline of directions for future research.

9.1 Two Experiments Studying the Interaction Between Conditionals and Cataphora

Brasoveanu and Dotlačil (2015a) investigate whether meaning representations commonly used in formal semantics are built up incrementally and predictively when language is used in real time, similar to the incremental and predictive construction of syntactic representations (Steedman 2001; Lewis and Vasishth 2005; Lau 2009; Hale 2011 among many others).

The main empirical challenge when studying the incremental processing of *semantic* representations is identifying phenomena that can tease apart the syntactic and semantic components of the interpretation process. The pervasive aspects of meaning composition that are syntax-based cannot provide an unambiguous window into the nature of semantic representation building: the incremental and predictive nature of real-time compositional interpretation could be primarily or exclusively due to our processing strategies for building syntactic representations.

There is a significant amount of work in psycholinguistics on incremental interpretation (Hagoort et al. 2004; Pickering et al. 2006 among many others), but this research usually focuses on the processing of lexical semantic and syntactic representations, and the incremental integration of world knowledge into the language interpretation process. The processing of logical representations of the kind formal semanticists are interested in is much less studied.

Similarly, there is a significant amount of work on incremental interpretation in natural language processing/understanding (Poesio 1994; Bos et al. 2004; Bos 2005; Hough et al. 2015 among many others), but this research usually discusses it from a formal and implementation perspective, and focuses much less on the cognitive aspects of processing semantic representations (the research in Steedman 2001 and related work is a notable exception).

Brasoveanu and Dotlačil (2015a) report two studies that argue for the incremental nature of processing formal semantic representations, as distinct from the syntactic representations they supervene on. The crucial evidence is provided by the interaction of anaphora and presupposition resolution on one hand, and conjunctions versus conditionals with a sentence-final antecedent on the other. Consider the contrast between AND and IF in the example below, where the presupposition trigger *again* is cataphoric:

(1) Tina will have coffee with Alex $\underline{\text{again}}$ AND/IF she had coffee with him at the local café.

Assume the construction of semantic representations is *incremental*, i.e., the interpreter processes *if* as soon as it is encountered. Furthermore, assume incremental interpretation is *predictive*, i.e., once *if* is read, the interpreter expects the upcoming *if*-clause to provide (some of) the interpretation context for the previously processed matrix clause. Then we expect to see a facilitation/speed-up in the second clause *she had coffee with him* after *if* is read, compared to when the same clause follows *and*. As we will see, this is what the experimental results show.

Specifically, we expect the second clause in (1) to be more difficult after *and* than after *if* because *and* signals that a potential antecedent for the *again* presupposition is unlikely to come after this point. Dynamic conjunction is interpreted sequentially: the second conjunct is interpreted relative to the context provided by the first conjunct, and not vice-versa. Consequently, an unresolved presupposition in the first conjunct cannot find an antecedent in the second conjunct.

In contrast, *if* leaves open the possibility that a suitable resolution for the *again* presupposition is forthcoming since the first clause (the matrix) is interpreted relative to the context provided by the second clause (the *if*-clause). This possibility allows interpreters to make better predictions about the content of the clause coming after *if*, which should ease its processing.

Crucially, our expectations—which arise from the interaction between the presupposition trigger *again* and the operators *and* versus *if*—are semantically driven. Nothing in the syntax of conjunction versus *if*-adjunction could make a successful presupposition resolution more or less likely.

9.1.1 Experiment 1: Anaphora Versus Cataphora in Conjunctions Versus Conditionals

Elbourne (2009, 1) defines donkey cataphora as "a configuration in which a pronoun precedes and depends for its interpretation on an indefinite that does not c-command it." Some cataphora examples with conditionals are provided below, both with sentence-initial, (2)–(6), and with sentence-final *if*-clauses, (7).

(2) If it is overcooked, a hamburger doesn't taste good. (Chierchia 1995, 129)

(3) If she finds it spectacular, a photographer takes many pictures of a landscape. (Chierchia 1995, 130)

(4) If it enters his territory, a pirate usually attacks a ship. (Chierchia 1995, 130)

(5) If it spots a mouse, a cat attacks it. (Chierchia 1995, 130)

(6) If a foreigner asks him for directions, a person from Milan replies to him with courtesy. (Chierchia 1995, 130)

(7) John won't eat it if a hamburger is overcooked. (Elbourne 2009, 3)

Certain configurations are not acceptable (Elbourne 2009, 2), e.g., (8c) below, due to Principle C violations. Antecedents are marked with a superscript, and the corresponding anaphors/cataphors are marked with a subscript.

(8) a. John^i is upset if he$_i$ sees a donkey.
 b. If John^i sees a donkey, he$_i$ is upset.
 c. *He$_i$ is upset if John^i sees a donkey.

The contrast between (8b) and (8c), as well as the fact that Principle C is not violated if cataphoric pronouns appear in object position (see (7) above and (9) below), provide evidence that a sentence-final *if*-clause is adjoined lower than the matrix-clause subject, but higher than the object. For concreteness, let's say that a sentence-final *if*-clause is VP-adjoined.

(9) Bill visits her$_i$ if Mary$_i$ is sick.

In contrast, a sentence-initial *if*-clause is adjoined higher than the matrix-clause subject. For concreteness, let's say it is CP-adjoined.

Other arguments for these two syntactic structures are provided by VP ellipsis, as in (10), and VP topicalization, as in (11); see Bhatt and Pancheva (2006) for more discussion.

(10) I will leave if you do, and John will [~~leave if you do~~] too / do so too.

(11) I told Peter to take the dog out if it rains, and [take the dog out if it rains] he will. (Iatridou 1991, 12)

Based on these observations, Brasoveanu and Dotlačil (2015a) conclude that there is no 'ordinary' syntax-mediated binding from a c-commanding position for direct object (DO) donkey cataphora in conditionals with sentence-final *if* clauses. That is, donkey cataphora from the DO position of the matrix clause is a 'true' example of donkey cataphora that can be used to test the incrementality and predictiveness of semantic parsing.

Brasoveanu and Dotlačil's Experiment 1 tested donkey anaphora and cataphora in a 2 × 2 design, exemplified in (12):

(12) Brasoveanu and Dotlačil's Experiment 1: AND/IF × DO ANAPHORA/CATAPHORA
  a. An electrician examined <u>a radio</u> for several minutes AND his helpful colleague held it that whole time. AND & ANAPHORA
  b. An electrician examined <u>a radio</u> for several minutes IF his helpful colleague held it that whole time. IF & ANAPHORA
  c. An electrician examined <u>it</u> for several minutes AND his helpful colleague held the radio that whole time. AND & CATAPHORA
  d. An electrician examined <u>it</u> for several minutes IF his helpful colleague held the radio that whole time. IF & CATAPHORA

Kazanina et al. (2007) used an on-line reading methodology (self-paced reading) to show that a cataphoric pronoun triggers an active search for an antecedent in the following material, and that this search takes into account structural constraints (Principle C) from an early stage.[1] Kazanina et al. (2007) take the temporal priority of syntactic information as evidence for the incremental and predictive nature of

[1]That is, cataphoric dependencies are processed with a syntactically constrained search mechanism similar to the mechanism used for processing long-distance *wh*-dependencies (Stowe 1986; Traxler and Pickering 1996; Wagers et al. 2009 among others).

syntactic constraints. The question investigated in this experiment can therefore be further specified as: is this active search mechanism also semantically constrained?

The methodology used in Brasoveanu and Dotlačil's Experiment 1 was also self-paced reading (Just et al. 1982) with a non-cumulative moving window. The regular anaphora cases provide the baseline conditions. Assuming a deep enough incremental and predictive interpretation, the second clause in these conditions is expected to be more difficult after *if* than after *and* because of extra cognitive load coming from two sources.

The first source of difficulty is the semantics of conditionals versus conjunctions. For conditionals, we generate a hypothetical intermediate interpretation context satisfying the antecedent, and we evaluate the consequent relative to this hypothetical context. That is, we need to maintain both the actual, global interpretation context and the intermediate, antecedent-satisfying context, to complete the interpretation of conditionals. There is no similar cognitive load for conjunctions.

The second source of difficulty is specific to conditionals with a sentence-final *if*-clause. When such conditionals are incrementally interpreted, the matrix clause needs to be semantically reanalyzed. The matrix clause is initially interpreted relative to the global context, just like a top-level assertion is. However, when *if* is reached, the matrix clause has to be reinterpreted relative to the intermediate, antecedent-satisfying context: the comprehender realizes that the matrix clause is not a top-level assertion, but is the consequent of a conditional instead. There is no such difficulty for conjunctions: the first conjunct is simply interpreted relative to the global context, and the second conjunct is interpreted relative to the context that is the result of the update contributed by the first-conjunct.

For the cataphora (non-baseline cases), we expect a cognitive load reversal. The conjunction *and* signals that no suitable antecedent for the cataphor is forthcoming since the second clause is interpreted relative to the context provided/updated by the first clause. In contrast, *if* triggers the semantic reanalysis of the matrix clause and leaves open the possibility that a suitable antecedent for the cataphor is forthcoming since the matrix clause is interpreted relative to the context provided/updated by the second clause. This expectation (and the fact that it is confirmed) should speed up the processing. So we expect to see a speed-up in the IF & CATAPHORA cases, i.e., a negative IF × CATAPHORA interaction.

These predictions were only partially confirmed in Brasoveanu and Dotlačil's first experiment: baseline IF was indeed harder (statistically significant) but the IF × CATAPHORA interaction, while negative, did not reach significance.

The regions of interest (ROIs) were primarily (i) the post-connective ROIs *his helpful colleague*, and secondarily (ii) the post-resolution ROIs *that whole*; see (13) below.

The mean log reading times (log RTs) for the 5 ROIs are plotted in Fig. 9.1 (plots created with `R` and `ggplot2`; R Core Team 2014; Wickham 2009).

(13) An electrician examined a radio/it for several minutes and/if $\boxed{\text{his helpful colleague}}$ held it/the radio $\boxed{\text{that whole}}$ time.

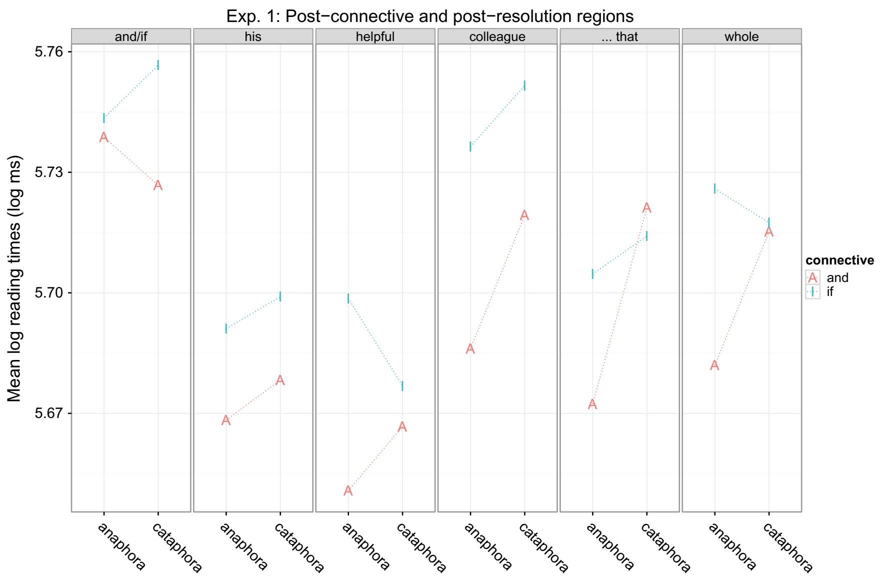


Fig. 9.1 Experiment 1: mean log RTs for the five regions of interest (ROIs)

Brasoveanu and Dotlačil analyzed the data using linear mixed-effects models. The response was the log-transformed readings times (log RTs) for the 3 ROIs immediately following the sentential operator *and*/*if* (RTs are log-transformed to mitigate their characteristic right-skewness). Residualized log RTs (residualized for word length and word position, following Trueswell et al. 1994) were also analyzed, but the pattern of results did not change, so Brasoveanu and Dotlačil report the more easily intelligible models with raw log RT responses.

The predictors (fixed effects) were as follows:

- main effects of CONNECTIVE and ANA/CATAPHORA, and
- their interaction;
- the levels of the CONNECTIVE factor: AND (reference level) versus IF;
- the levels of the ANA/CATAPHORA factor: ANAPHORA (reference level) versus CATAPHORA.

Crossed random effects for subjects and items were included, and the models with maximal random effect structures that converged (Barr et al. 2013) were reported, usually subject and item random intercepts, and subject and item random slopes for at least one of the two main effects. The maximum likelihood estimates (MLEs) and associated standard errors (SEs) and p-values are provided in (14) and (15) below (we omit the intercepts). Significant and nearly significant effects ($p < 0.1$) are boldfaced.

(14)

	his			**helpful**			**colleague**		
	MLE	SE	*p*	MLE	SE	*p*	MLE	SE	*p*
IF	0.02	0.02	0.3	**0.05**	**0.02**	**0.04**	**0.05**	**0.02**	**0.04**
CATA	0.01	0.02	0.7	0.02	0.02	0.4	0.03	0.02	0.16
IF×CATA	−0.003	0.03	0.9	−0.04	0.03	0.15	−0.02	0.03	0.6

(15)

	that			**whole**		
	MLE	SE	*p*	MLE	SE	*p*
IF	0.03	0.02	0.20	0.03	0.02	0.24
CATA	**0.04**	**0.02**	**0.07**	0.03	0.02	0.18
IF×CATA	−0.03	0.03	0.26	−0.03	0.03	0.43

Table 14 shows several effects. First, baseline IF (i.e., IF & ANAPHORA) is more difficult than baseline AND (i.e., AND & ANAPHORA). This effect is compatible with the hypothesis that to interpret conditionals, we need to maintain both the actual, global interpretation context and the intermediate, antecedent-satisfying context. The effect is also compatible with the hypothesis that the matrix clause is reanalyzed in conditionals with a final *if*-clause: it is initially interpreted relative to the global context until *if* is reached, at which point it is reinterpreted relative to the intermediate, antecedent-satisfying context.

CATAPHORA seems to be more difficult than ANAPHORA for AND, but the effect never reaches significance (it is close to significant in the first ROI after cataphora is resolved). Maybe the AND & CATAPHORA condition is simply too hard, so readers stop trying to fully comprehend the sentence and speed up. If so, this will obscure the IF × CATAPHORA interaction: there is a negative interaction between IF and CATAPHORA in all ROIs, i.e., IF seems to facilitate CATAPHORA (as expected if semantic evaluation is incremental and predictive), but this effect is not significant.

The consistent negative interaction is promising, so Brasoveanu and Dotlačil (2015a) elicited it in a follow-up experiment with a hard presupposition trigger (*again*; Abusch 2010; Schwarz 2014 among others), which might have a larger effect. A third '(mis)match' manipulation was also added to control for readers speeding up through conditions that are too hard.

9.1.2 Experiment 2: Cataphoric Presuppositions in Conjunctions Versus Conditionals

The second experiment in Brasoveanu and Dotlačil (2015a) had a 2 × 2 × 2 design, exemplified in (16) below. The match/mismatch manipulation was new, and consisted of verbs in the second clause that matched or didn't match the corresponding verbs in the first clause.

(16) Experiment 2: (MIS)MATCH × AND/IF × NOTHING/CATAPHORA

a. Jeffrey will *argue* with Danielle AND he *argued* with her in the courtyard last night. MATCH & AND & NOTHING

b. Jeffrey will *argue* with Danielle IF he *argued* with her in the courtyard last night. MATCH & IF & NOTHING
c. Jeffrey will *argue* with Danielle again AND he *argued* with her in the courtyard last night. MATCH & AND & CATAPHORA
d. Jeffrey will *argue* with Danielle again IF he *argued* with her in the courtyard last night. MATCH & IF & CATAPHORA
e. Jeffrey will *argue* with Danielle AND he *played* with her in the courtyard last night. MISMATCH & AND & NOTHING
f. Jeffrey will *argue* with Danielle IF he *played* with her in the courtyard last night. MISMATCH & IF & NOTHING
g. Jeffrey will *argue* with Danielle again AND he *played* with her in the courtyard last night. MISMATCH & AND & CATAPHORA
h. Jeffrey will *argue* with Danielle again IF he *played* with her in the courtyard last night. MISMATCH & IF & CATAPHORA

The method was similar to Experiment 1. Self-paced reading with a moving window was used, but each stimulus ended with an acceptability judgment on a 5-point Likert scale, from 1 (very bad) to 5 (very good). The acceptability judgment was elicited on a new screen after every item or filler. Every experimental item was followed by a comprehension question. Each of the 8 conditions was tested 4 times, for 32 items total; one item had a typo, and was discarded from all subsequent analyses. There were 70 fillers: monoclausal and multiclausal, conditionals, conjunctions, *when*-clauses, relative clauses, quantifiers, adverbs, etc.

Thirty-two native speakers of English participated (UCSC undergraduate students). They completed the experiment online for course (extra-)credit on a UCSC hosted installation of the IBEX platform. Each item was passed through all 8 conditions, and 8 lists were generated following a Latin square design. The participants were rotated through the 8 lists. Every participant responded to 102 stimuli (32 items + 70 fillers), the order of which was randomized for every participant. Any two items were separated by at least one filler.

There were fillers that were both acceptable (*Bob ate his burger and he rented something to watch, but he didn't say what*) and unacceptable (*Willem visited Paris because Sarah visited Amsterdam too*). All participants exhibited the expected difference in acceptability ratings between these two types of fillers.

There were 72 comprehension questions with correct/incorrect answers, 32 after experimental items. The accuracy for all participants was above 80%.

The results of this study are visually summarized in Fig. 9.2. The ROIs for Exp. 2 are the words following the verb in the second clause, i.e., the words immediately following the last experimental manipulation, which is (MIS)MATCH. Brasoveanu and Dotlačil examined only the 4 immediately post-verbal ROIs because the fifth word was the final one for some items, and the wrap-up effect associated with sentence-final words would contribute additional, possibly biasing noise.

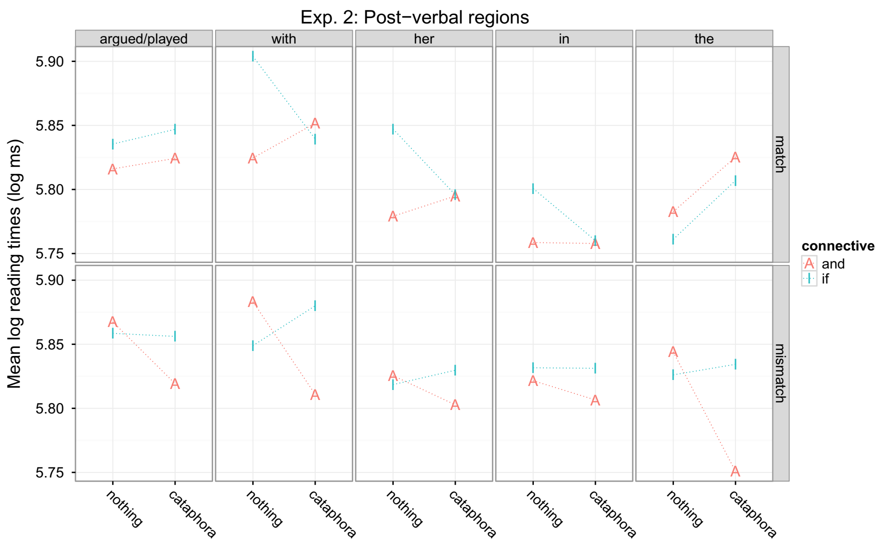


Fig. 9.2 Experiment 2: mean log RTs for the four regions of interest (ROIs)

(17) Jeffrey will argue with Danielle ⊘/again and/if he argued/played $\boxed{\text{with her in the}}$ courtyard last night.

The data analysis was similar to the one conducted for Experiment 1: linear mixed-effects models with log RTs as the response variable, and main effects of CONNECTIVE and NOTHING/CATAPHORA, MATCH/MISMATCH and their 2-way and 3-way interactions as predictors (fixed effects). The levels of the 3 factors were as follows:

- for CONNECTIVE, AND (reference level) versus IF,
- for NOTHING/CATAPHORA: NOTHING (reference level) versus CATAPHORA, and
- for (MIS)MATCH: MATCH (reference level) versus MISMATCH.

The models also included crossed random effects for subjects and items, namely the maximal random effect structure that converged (usually subject and item random intercepts, and subject and item random slopes for at least two of the three main effects). The statistical modeling results are summarized in (18) below (once again, we omit the intercepts).

(18)

	with			**her**			**in**			**the**		
	MLE	SE	*p*	MLE	SE	*p*	MLE	SE	*p*	MLE	SE	*p*
CATA	0.05	0.04	0.21	0.05	0.04	0.30	0.03	0.04	0.50	0.05	0.04	0.23
MISMATCH	0.05	0.04	0.25	0.04	0.04	0.33	0.06	0.05	0.19	0.07	0.05	0.15
IF	**0.08**	**0.04**	**0.054**	**0.07**	**0.04**	**0.084**	0.05	0.04	0.24	−0.003	0.04	0.96
CATA × MISMATCH	**−0.11**	**0.06**	**0.056**	−0.05	0.06	0.42	−0.03	0.06	0.59	**−0.14**	**0.06**	**0.03**
CATA × IF	**−0.13**	**0.06**	**0.026**	**−0.11**	**0.06**	**0.077**	−0.08	0.05	0.15	−0.04	0.06	0.54
MISMATCH × IF	**−0.10**	**0.06**	**0.083**	−0.06	0.06	0.30	−0.02	0.05	0.73	−0.02	0.06	0.76
CATA × MISMATCH × IF	**0.20**	**0.08**	**0.015**	0.10	0.08	0.22	0.06	0.08	0.42	0.11	0.09	0.19

Just as in Experiment 1, baseline IF (i.e., IF & NOTHING & MATCH) is more difficult than baseline AND (i.e., AND & NOTHING & MATCH). This is compatible with the hypothesis that conditionals are harder than conjunctions—because we need to maintain two evaluation contexts, and/or because the matrix clause is semantically reanalyzed when *if* is reached.[2]

There is a significant negative interaction of MISMATCH×IF (note: *again* is not present here), which basically cancels out the main effect of IF. That is, conditionals with non-identical VP meanings in the antecedent and consequent clauses are processed more easily than conditionals with identical VP meanings, about as easily as conjunctions with non-identical VP meanings in the two conjuncts. The difficulties tied to conditionals with identical VP meanings are probably caused by a violation of Maximize Presupposition (Heim 1991), which requires that a presupposed VP meaning should be marked as such by *again*. This penalizes conditionals with matching VP meanings, while conditionals with non-identical VP meanings are not affected. Furthermore, if participants interpret incrementally and predictively, Maximize Presupposition should not affect coordinations (specifically, the first conjunct in a coordination), which corresponds to our findings.

The Maximize Presupposition constraint provides a third possible reason for the cost of baseline IF relative to baseline AND aside from the suggestions discussed before, namely that conditionals are harder than conjunctions because we need to maintain two evaluation contexts, and/or because the matrix clause is semantically reanalyzed. They all might be at work here (distinguishing between them is left for a future occasion), but Maximize Presupposition might be particularly suitable as an explanation for the Experiment 2 results: it explains the cost of IF, but it also explains the negative interaction MISMATCH × IF, which is unexpected under the hypothesis that IF on its own is costly. Furthermore, in Experiment 2, the effect of IF is observed on *with* and *her*, which makes the explanation in terms of reanalysis unlikely given the lateness of the effect. In Experiment 1, the effect of IF was detectable on the second word after *if*, so the reanalysis explanation is more plausible for that experiment.

There are no main effects of CATAPHORA and MISMATCH, but their 2-way interaction is negative and significant (or close to significant) in two out of the four regions. Whenever (close to) significant, this interaction effectively cancels the main effects of both MISMATCH and CATAPHORA. That is, the AND & CATAPHORA & MISMATCH condition is about as difficult as the reference condition AND & NOTHING & MATCH, which suggests that participants stopped trying to properly interpret the difficult condition AND & CATAPHORA & MISMATCH and moved on/sped up.

[2]Since Experiment 2 included acceptability judgments, Brasoveanu and Dotlačil were able to check whether the effect, replicated from Experiment 1, is not due to the fact that the IF condition (*Jeffrey will argue with Danielle if he argued with her in the courtyard last night*) is less felicitous than the corresponding AND condition (*Jeffrey will argue with Danielle and he argued with her in the courtyard last night*). This is not so: the only statistically significant fixed effect was a *positive* main effect for IF, i.e., baseline IF is more acceptable than baseline AND, as estimated by a mixed-effects ordinal probit model (full fixed-effect structure, i.e., main effects + all interactions, & maximal random effect structure that converged).

There is a (close to) significant negative interaction of CATAPHORA×IF in the two regions immediately following the verb (note: we are discussing MATCHING conditions). In both regions, this 2-way interaction effectively cancels out the positive main effects of CATAPHORA and IF put together. This is exactly the configuration Brasoveanu and Dotlačil were looking for in Experiment 1, only it did not reach significance there. That is, IF facilitates the processing of CATAPHORA, even though IF and CATAPHORA on their own are more difficult. This supports the hypothesis that the construction of formal semantic representations is incremental and predictive.

Finally, the statistically significant and positive 3-way interaction CATAPHORA × IF × MISMATCH in the region immediately following the verb provides further empirical support for the hypothesis that the construction of formal semantic representations is incremental and predictive. The MISMATCH is surprising because the human interpreter expects to find a suitable antecedent for the *again* presupposition, and that expectation is not satisfied.

In sum, Experiments 1 and 2 in Brasoveanu and Dotlačil (2015a) provide coherent support for the incremental and predictive nature of the process of constructing meaning representations of the kind employed in formal semantics.

9.2 Mechanistic Processing Models as an Explanatory Goal for Semantics

The main questions at this point are the following. As formal semanticists, should we account for the incremental and predictive nature of the real-time semantic interpretation process? And if so, how?

It is important to remember that addressing these questions is firmly rooted in the tradition of dynamic semantics. Kamp (1981) begins like this:

> "Two conceptions of meaning have dominated formal semantics of natural language. The first of these sees meaning principally as that which determines conditions of truth. […] According to the second conception meaning is, first and foremost, that which a language user grasps when he understands the words he hears or reads. […] these two conceptions […] have remained largely separated for a considerable period of time. This separation has become an obstacle to the development of semantic theory […] The theory presented here is an attempt to remove this obstacle. It combines a definition of truth with a systematic account of semantic representations." (Kamp 1981, 189)

Thus, the implicit overarching goal for us as (cognitive) scientists studying natural language meaning and interpretation is to provide a formally explicit account of natural language interpretive *behavior*, i.e., a mathematically explicit, unified theory of semantic/pragmatic competence *and* performance.

To contextualize our position and outline some possible alternatives, let us consider the corresponding debate on the syntax side. Phillips and Lewis (2013, 14) identify two reasonable positions that working linguists more or less implicitly subscribe to in practice: (i) principled extensionalism, and (ii) strategic extensionalism.

Principled extensionalism takes a grammar/grammatical theory to be merely an abstract characterization of a function whose extension is all and only the well-formed sentences of a given language (plus their denotations, if the grammar incorporates a semantic component).[3] The individual components of the grammatical theory have no independent status as mental objects or processes: they are components of an abstract function, not of a more concrete description of a mental system.

This kind of position cannot be tested using most empirical evidence aside from acceptability (or truth-value/entailment) judgments, since the position only aims to capture the 'end products' of the grammatical system and not the way these products are actually produced/comprehended.

The 'principled' part is that the extensionalist enterprise is understood as an end in itself, relevant even if lower-level characterizations of the human language system are provided (algorithmic/mechanistic, or implementation/neural level; Marr 1982). The linguist's task is to characterize *what* the human language system computes and distinguish it from *how* speakers actually carry out that computation, which is the psycholinguist's task.

The *strategic extensionalism* position takes the goal of formulating a grammatical theory to be a reasonable interim goal, but not an end in itself. The ultimate goal is to move beyond extensional description to a more detailed, mechanistic understanding of the human language system: describing an abstract function that identifies all the grammatical sentences of a language is just a first step in understanding how speakers actually comprehend/produce sentences in real time. We seek theories that capture *how* sentences are put together, and not just *what* their final form is. From this perspective, we *should* try to account for left-to-right structure building mechanisms, both at the syntactic and at the semantic level.

The strategic-extensionalism position is closely related to the cognitive-architecture based approach to research in cognitive science, which we have in fact followed throughout this book. As Anderson (2007, 7–8) puts it:

> "A *cognitive architecture* is a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the mind [...] [i.e.,] human cognition in all of its complexity. [...] Th[is] type of architectural program [...] requires paying attention to three things: brain, mind (functional cognition), and the architectural abstractions that link them. [...] [A]pproaches that tried to get by with less [...] can be viewed as shortcuts to understanding."

Anderson (2007) goes on to consider three such shortcuts. The first one is classical information-processing psychology that completely ignores the brain (at *any* level of abstraction), and that is basically the same as the principled-extensionalism position we characterized above. However, unlike much of formal semantics, cognitive psychology has realized by now that "cognition is not so abstract that our understanding of it can be totally divorced from our understanding of [the] physical reality [underlying it]." (Anderson 2007, 11)

[3]More precisely, assume some background alphabet Σ that consists of the lexicon/set of words ('alphabet' in the sense of formal language theory), and let Σ^* be the set of all finite strings over Σ. Σ^* is the domain of the function, and $\{0, 1\}$ its range, so that the function is a characteristic function of the set of grammatical strings.

This does not mean, of course, that the opposite position—eliminative connectionism—is not a shortcut also. “This approach ignores mental function as a constraint and just provides an abstract characterization of brain structure […] [but these models] work only because we are able to imagine how [they] could serve a useful function in a larger system […] [However, this] functionality is not achieved by a connectionist system.” (Anderson 2007, 11–14)

Finally, a third shortcut that has become recently fairly popular in formal semantics and pragmatics, is the rational-analysis approach to cognition. The basic insight behind this approach is that “a constraint on how the brain achieves the mind is that both the brain and the mind have to survive in the real world: rather than focus on architecture as the key abstraction, focus on adaptation to the environment […] [T]he Bayesian statistical methodology that accompanies much of this research […] comes to take the place of the cognitive architecture.” (Anderson 2007, 15–16)

Anderson’s own work on declarative memory is an early instantiation of this approach; see Anderson (1990); Anderson and Schooler (1991); Schooler and Anderson (1997). As we discussed in detail in Chap. 6, the ACT-R base activation equation encodes that “a memory for something diminishes in proportion to how likely people are to need that memory. […] Human memory […] mirror[s] statistical relationship[s] in the environment. […] Thus, the argument goes, one does not need a description of how memory works, which is what an architecture gives; rather, one just needs to focus on how memory solves the problems it encounters.” (Anderson 2007, 17)

While possibly enlightening for individual cognitive components, this approach falls short of a complete theory of the human mind.

> “[T]he human mind is not just the sum of core competences such as memory, or categorization, or reasoning. It is about how all these pieces and other pieces work together to produce cognition. All the pieces might be adapted to the regularities in the world, but understanding their individual adaptations does not address how they are put together. […] What distinguishes humans is their ability to bring the pieces together, and this unique ability is just what adaptive analyses do not address, and just what a cognitive architecture is all about.” (Anderson 2007, 18)

In this book, we have consistently taken a cognitive-architectural approach to natural language meaning and interpretation. Our ultimate goal is to provide a framework in which we can build mechanistic processing models for natural language comprehension, with pieces that are independently needed for other higher-level cognitive processes.

It is in this context that we introduced and used Bayesian methods: we use them for theoretically-informed *data analysis*. More precisely, we use them as essential bridges that systematically connect independently-motivated semantics and processing theories and cognitive-architectural organization principles and constraints on one hand, and experimental data on the other hand.

9.3 Modeling the Interaction of Conditionals and Pronominal Cataphora

Assuming a cognitive-architecture based approach to semantics, like we have done throughout this book (or a strategic extensionalist position), the next question is: how should we account for the incremental and predictive nature of semantic interpretation? We will not settle this question here, but we will outline two distinct approaches and flesh out in detail one of them.

As far as we can tell, there is a spectrum of approaches to incrementality effects, and the two extremes on that spectrum are accounting for incrementality (i) in *the semantics* versus (ii) in *the processor*.

The first alternative is parallel to the proposal in Phillips (1996, 2003) on the syntax side. The main claim in Phillips (1996, 2003) is that syntactic structures are built left-to-right, not top-down/bottom-up, and the incremental left-to-right system is the *only* structure-building system that humans have ('the parser is the grammar').

A similar proposal on the semantics side is sketched in Brasoveanu and Dotlačil (2015a, b). The idea is to provide a recursive definition of truth and satisfaction for first-order predicate logic that is fully incremental, building on the incremental propositional logic system in Vermeulen (1994). The resulting system, dubbed Incremental Dynamic Predicate Logic (IDPL), builds incrementality into the heart of semantics.

The second alternative is parallel to the proposal in Hofmeister et al. (2013) on the syntax side, the main goal of which is to argue that "many of the findings from studies putatively supporting grammar-based interpretations of island phenomena have plausible, alternative interpretations rooted in specific, well-documented processing mechanisms" (Hofmeister et al. 2013, 44). The remainder of this chapter is dedicated to fleshing out this approach.

Our specific proposal on the processing side is to extend the eager left-corner parser for DRT we introduced in the previous chapter with conjunctions, conditionals and anaphora/cataphora, so that we can explicitly and fully model the two self-paced reading experiments discussed in Sect. 9.1 above.

In this section, we introduce the basic model that captures the qualitative pattern of interactions between cataphora and conjunctions versus conditionals in Experiment 1. In the next section, we introduce the model in its full complexity. The full model can syntactically and semantically parse the items in Experiment 2, which enables us to quantitatively fit it to the data from Experiment 2.

To model pronominal and presuppositional anaphora/cataphora, we add a new goal-like buffer `unresolved_discourse` to our ACT-R mind, which will store the unresolved DRSs contributed by pronouns and the presuppositional trigger *again*. We set the encoding delay for this buffer, as well as the `imaginal` and `discourse_context` buffers we used in the previous chapter to 0:

(19)
```
parser.set_goal(name="imaginal", delay=0)
parser.set_goal(name="discourse_context", delay=0)
parser.set_goal(name="unresolved_discourse", delay=0)
```

In principle, we might be able to model anaphora and cataphora without this additional `unresolved_discourse` buffer, but we decided to use it here for presentational clarity.[4]

9.3.1 *Chunk Types and the Lexical Information Stored in Declarative Memory*

The chunk types we will need are the same as the ones we used in the previous chapter, plus a new chunk type for predicates. They are listed in (20) below.

(20)
```
actr.chunktype("parsing_goal",
               "task stack1 stack2 stack3 \
                arg_stack1 arg_stack2 \
                right_edge_stack1 right_edge_stack2 \
                right_edge_stack3 right_edge_stack4 \
                parsed_word found discourse_status \
                dref_peg event_peg drs_peg prev_drs_peg embedding_level \
                entity_cataphora event_cataphora if_conseq_pred")
actr.chunktype("parse_state",
               "node_cat daughter1 daughter2 daughter3 \
                mother mother_of_mother lex_head")
actr.chunktype("word", "form cat pred1 pred2")
actr.chunktype("pred", "constant_name arity")
actr.chunktype("drs",
               "dref pred1 pred2 event_arg arg1 arg2 \
                discourse_status drs embedding_level")
```

Parsing goal chunks have the expected features:

- the current parsing `task`;
- the stack of syntactic goals driving the parsing process (`stack1 stack2 stack3`);
- a stack for arguments (`arg_stack1 arg_stack2`) that need to be passed across different semantic chunks, e.g., from the subject to the verbal predicate;
- the right-edge stack keeps track of possible points of attachment made available by the current, partially-built syntactic tree (`right_edge_stack1 right_edge_stack2 ...`);
  - we called this stack `right_frontier` in the previous chapter, but we renamed it here for brevity;
  - the right-edge stack we need in this chapter has additional positions because of the need to attach conjuncts and *if*-adjuncts;
- the `parsed_word` and `found` features are used in much the same way as in the previous chapters;
- the `discourse_status` feature will keep track of whether a DRS constructed at some point during the incremental interpretation process:

[4]We are of course aware that determining the exact number of modules and buffers needed for natural language interpretation, as well as their subsymbolic properties, is an empirical issue.

  - contributes to the at_issue meaning, or
  - is unresolved, e.g., it is the presupposition contributed by a pronoun or the adverb *again*, or
  - is presupposed, i.e., it is the resolved presupposition contributed by a pronoun or the adverb *again*;

- just as in the previous chapter, we introduce new discourse referents (drefs) with fresh (previously unused) indices by keeping track of the current dref_peg/index in the goal buffer and updating it as soon as a dref with that peg/index is introduced;
  - but in this chapter, we have event drefs (needed for *again*) in addition to individual-level drefs, so we also keep track of the current event_peg;
  - in addition, we keep track of which sub-DRSs are part of the main DRS by associating them with current drs_peg; this is a flatter/simpler solution than the one we used in the previous chapter, where a main DRS had sub-DRSs stored in its slots;
  - DRS drefs are basically propositional drefs and are independently needed for conditionals, for example; we keep track of the previous DRS peg (prev_drs_peg) to be able to capture the semantic reanalysis triggered by sentence-final *if*-clauses;
- finally, the three features entity_cataphora, event_cataphora and if_conseq_pred are needed to account for the processing of cataphoric pronouns and *again*, and will be discussed in detail later in the chapter.

Chunks of parse_state type have the same structure as before, except for the addition of a mother_of_mother feature. This feature enables us to keep track of the partial syntactic structure constructed by the incremental comprehension process in a little more detail at the local level of an individual chunk.

The lexical entry of a word keeps track of:

- its written form (a proxy for its phonological representation),
- its syntactic category, and
- up to two predicates pred1 and pred2 that represent the meaning of that word.

We use these two predicate slots in various ways. For example, a proper name like *Danielle* contributes:

- a predicate DANIELLE (with a singleton set denotation) as its pred1 value (this follows the analysis of proper names in Kamp and Reyle 1993), and
- the gender predicate FEMALE as its pred2 value that can be leveraged to resolve a subsequent pronoun *she*/*her* anaphoric to the proper name.

The values of these two pred1 and pred2 features are chunks of type pred, which specify the name of the non-logical constant associated with the predicate (the feature constant_name) and the arity of the constant.

Finally, chunks of type `drs` have the same structure as the one used in the previous chapter, with the addition of several new features necessary to capture the interaction of cataphora and conjunctions/conditionals:

- the `dref` feature keeps track of the new propositional dref (if any) introduced by the DRS;
- `pred1` and `pred2` store the predicates that are part of the conditions contributed by the DRS;
- `event_arg` stores the event dref (if any) taken as an argument by `pred1` and/or `pred2`;
- `arg1` and `arg2` store the entity drefs that are the arguments of `pred1` and/or `pred2`;
- `discourse_status` keeps track of the discourse status of the DRS;
  - we only need three possible values for this feature in this chapter: `at_issue`, `unresolved` and `presupposed`;
- the `drs` feature keeps track of the DRS peg that the current DRS is associated with;
- finally, `embedding_level` keeps track of the embedding level of the DRS, the main function of which is to constrain pronoun and anaphora resolution, just as in Kamp and Reyle (1993);
  - discourse-initial main clauses are `embedding_level` 0;
  - conditional antecedents or the second conjunct in a conjunction are `embedding_level` 1;
  - conditional consequents or the third conjunct in a conjunction will be `embedding_level` 2;
  - pronouns and presuppositions, whether anaphoric or cataphoric, can only find antecedents at a higher embedding level (or at the same level in certain cases).

Let us look at some example lexical entries that will be stored in declarative memory (`dm`). The lexical entry of a proper name like *Danielle* is a chunk of the following form:

(21)
```
actr.chunkstring(string="""
    isa  word
    form Danielle
    cat  ProperN
    pred1 DANIELLE
    pred2 FEMALE
""")
```

In (21), we use the `chunkstring` method to assemble the lexical entry for the proper name *Danielle* (of type `word`) from a Python3 string. The values of the `form` and `cat` features are as expected. The `pred1` and `pred2` values are themselves chunks of type `pred`. These predicate chunks are assumed to be already available in `dm` at the time we assemble the lexical entry for *Danielle*, and they are declared as follows:

(22)
```
actr.chunkstring(name="DANIELLE", string="""
    isa           pred
    constant_name _danielle_
    arity         1
""")
```

(23)
```
actr.chunkstring(name="FEMALE", string="""
    isa           pred
    constant_name _female_
    arity         1
""")
```

Just as in the previous chapter, the lexical entry for the determiner *a* does not contain any semantic information. The associated semantic representations and operations, namely, introducing a new dref, predicating the common noun and the verbal predicate of it etc., are all contributed by production rules stored in procedural memory.

A pronoun like *she* has a lexical entry of the following form:

(24)
```
dm.add(actr.chunkstring(string="""
    isa  word
    form she
    cat  PRO
    pred1 EQUALS
    pred2 FEMALE
"""))
```

The gender of the pronoun, which the antecedent of the pronoun will have to satisfy, is stored as the `pred2` value. We follow the semantics for pronouns proposed in Kamp and Reyle (1993) and assume that pronouns introduce their own dref, but they need to equate it with the dref contributed by a suitable antecedent. This is the reason for making `EQUALS` the 'main' predicate contributed by the pronoun, i.e., the value of the `pred1` feature. The exact specification of the `EQUALS` predicate, which has an arity of 2 (as expected), is provided in (25) below.

(25)
```
actr.chunkstring(name="EQUALS", string="""
    isa           pred
    constant_name _equals_
    arity         2
""")
```

It would be natural to have a lexical entry for *again* that would be parallel to that of pronouns, except that it would relate event drefs instead of entity drefs, and it would contribute the predicate `PRECEDES` in (26) below instead of `EQUALS`.

(26)
```
actr.chunkstring(name="PRECEDES", string="""
    isa           pred
    constant_name _precedes_
    arity         2
""")
```

It turns out, however, that it is more convenient to give *again* a semantically empty lexical entry, shown in (27) below, and let suitably formulated production rules make the correct semantic contributions.

(27)
```
actr.chunkstring(string="""
    isa  word
    form again
    cat  Adv
""")
```

This superficial difference between pronouns and *again* is due to the fact that, syntactically, *again* is an adjunct that needs to retrieve the VP it adjoins to and reopen it for adjunction. In addition, semantically, *again* is 'parasitic' on the event dref contributed by the verbal predicate it adjoins to. This is in contrast to pronouns, which introduce their own entity dref.

However, what *again* and pronouns do have in common is that they both need to be resolved: they have to relate their dref, whether they introduce that dref or 'inherit' it from the VP, to another dref. A successful resolution requires the other dref to be available in, and retrieved from, declarative memory.

Just like proper names, common nouns introduce two predicates, one being the common noun itself and the other being the gender. For example, the lexical entry for *car* and its associated predicate chunks are as follows:

(28) a.
```
actr.chunkstring(string="""
    isa  word
    form car
    cat  N
    pred1 CAR
    pred2 NONHUMAN
""")
```
b.
```
actr.chunkstring(name="CAR", string="""
    isa           pred
    constant_name _car_
    arity         1
""")
```
c.
```
actr.chunkstring(name="NONHUMAN", string="""
    isa           pred
    constant_name _nonhuman_
    arity         1
""")
```

Intransitive and transitive verbs introduce a single predicate, with an arity that specifies that an event argument is required, plus 1 or 2 individual-level arguments. For example, *laughed* and *greeted* have the lexical entries in (29) and (30) below.

(29) a.
```
actr.chunkstring(string="""
    isa  word
    form laugh
    cat  Vi
    pred1 LAUGH
""")
```
b.
```
actr.chunkstring(name="LAUGH", string="""
    isa           pred
    constant_name _laugh_
    arity         event_plus_1
""")
```

(30) a.
```
actr.chunkstring(string="""
    isa  word
    form greet
    cat  Vt
    pred1 GREET
""")
```

b.
```
actr.chunkstring(name="GREET", string="""
    isa           pred
    constant_name _greet_
    arity         event_plus_2
""")
```

The items used in Experiment 2 (see Sect. 9.1.2 above) also contain prepositional verbs like *argue/play with*. We analyze them as transitive verbs, but we assign a different syntactic category `VtPP` to them. This will enable us to formulate production rules that will syntactically and semantically integrate them with the subsequent preposition.

(31) a.
```
actr.chunkstring(string="""
    isa  word
    form play
    cat  VtPP
    pred1 PLAY
""")
```
b.
```
actr.chunkstring(name="PLAY", string="""
    isa           pred
    constant_name _play_
    arity         event_plus_2
""")
```

For simplicity, we assume that prepositions like *with* that are part of prepositional verbs do not take an event argument, as shown in (32) below. We make the same simplifying assumption about adjectives like *overcooked*, as shown in (33).

(32) a.
```
actr.chunkstring(string="""
    isa  word
    form with
    cat  P
    pred1 WITH
""")
```
b.
```
actr.chunkstring(name="WITH", string="""
    isa           pred
    constant_name _with_
    arity         2
""")
```

(33) a.
```
dm.add(actr.chunkstring(string="""
    isa  word
    form hungry
    cat  A
    pred1 HUNGRY
"""))
```
b.
```
actr.chunkstring(name="HUNGRY", string="""
    isa           pred
    constant_name _hungry_
    arity         1
""")
```

Finally, we take the lexical entry of the sentential operators *and* and *if* to be semantically empty. The associated semantic representations and operations will all be contributed by production rules.

(34) a.
```
actr.chunkstring(string="""
    isa  word
    form and
    cat  Conj
""")
```

b.
```
actr.chunkstring(string="""
    isa  word
    form if
    cat  C
""")
```

The full code for this part of the model is linked to at the end of the chapter in Appendix 9.6.1.

9.3.2 Rules to Advance Dref Peg Positions, Key Presses and Word-Related Rules

The entire set of production rules is linked to at the end of this chapter (Appendix 9.6.2). Here, we will highlight only the most crucial ones.

We have several families of production rules that advance the dref peg position for (i) entity/individual-level drefs, (ii) event drefs and (iii) DRS drefs. The idea of peg positions was introduced and justified in Chap. 8 (see Sect. 8.3) and the rules we use in this chapter are the same, except we generalize this idea to drefs for types other than entities/individuals, namely event drefs and DRS/propositional drefs.

Turning to word-related rules, the `"encode word"` rule in (35) below fires whenever the parser is not engaged in a set of tasks that should take priority relative to word encoding (lines 4–14). We can think of the `"encode word"` rule as an 'elsewhere' rule: if the parser is not engaged in a more pressing task, it should check whether there is a value in the `visual` buffer that can be encoded (lines 17–19 in (35)). If such a value is available, it is encoded in the goal buffer as the value of the `parsed_word` feature (line 24).

(35)
```
parser.productionstring(name="encode word", string="""                      1
    =g>                                                                     2
    isa                 parsing_goal                                        3
    task                ~move_dref_peg                                      4
    task                ~move_event_peg                                     5
    task                ~move_event_peg_and_wait_for_retrieval              6
    task                ~move_drs_peg                                       7
    task                ~attempting_to_resolve_PRO                          8
    task                ~attempting_to_resolve_AGAIN                        9
    task                ~attempting_to_resolve_cataphoric_PRO               10
    task                ~attempting_to_resolve_cataphoric_AGAIN             11
    task                ~if_reanalysis                                      12
    task                ~stop_resolution_attempt_PRO                        13
    task                ~stop_resolution_attempt_AGAIN                      14
    found               None                                                15
    parsed_word         None                                                16
    =visual>                                                                17
    isa                 _visual                                             18
    value               =val                                                19
    ==>                                                                     20
    =g>                                                                     21
    isa                 parsing_goal                                        22
    task                encoding_word                                       23
    parsed_word         =val                                                24
    ~visual>                                                                25
    ~retrieval>                                                             26
""", utility=-1)                                                            27
```

The ‘elsewhere’ nature of the `"encode word"` rule is also reflected in the fact that we assign it a lower utility of −1 (line 27 in (35)) than the default, which is 0. In a more realistic system that learns rule utilities from data, this utility would be automatically inferred, and we would be able to greatly simplify the rule by removing the long list of negative conditions on lines 4–14: the fact that all the tasks listed on lines 4–14 need to take priority over word encoding should arise from the utilities of the relevant productions rules, rather than being hard-coded in this fashion. However, to make rule preferences transparent, we opted for hard-coding them in this model.

The `"retrieve word"` rule in (36) below requests lexical information about the word we just encoded from declarative memory.

(36)
```
parser.productionstring(name="retrieve word", string="""
    =g>
    isa                 parsing_goal
    task                encoding_word
    parsed_word         =w
    ==>
    +retrieval >
    isa                 word
    form                =w
    =g>
    isa                 parsing_goal
    task                retrieving_word
""")
```

Once the lexical information is retrieved and available in the `retrieval` buffer, we shift and project the word (37). This means that we build a unary-branching syntactic structure in the `imaginal` buffer, with the word as the daughter and its syntactic category as the mother (lines 15–18 in (37)). At the same time, we update the `found` slot in the goal buffer to the same syntactic category (line 14), so that other syntax/semantics processing steps necessary to integrate the retrieved word are triggered. Finally, we start a `key_press` task (line 13). The motor operations associated with this task will execute in parallel to the additional syntax/semantics processing steps associated with the retrieved word.

(37)
```
parser.productionstring(name="shift and project word (not N)", string="""   1
    =g>                                                                     2
    isa                 parsing_goal                                        3
    task                retrieving_word                                     4
    =retrieval>                                                             5
    isa                 word                                                6
    form                =w                                                  7
    cat                 =c                                                  8
    cat                 ~N                                                  9
    ==>                                                                     10
    =g>                                                                     11
    isa                 parsing_goal                                        12
    task                key_press                                           13
    found               =c                                                  14
    +imaginal>                                                              15
    isa                 parse_state                                         16
    node_cat            =c                                                  17
    daughter1           =w                                                  18
""")                                                                        19
```

The shift-and-project rule in (37) applies to all words except nouns (N). The shift-and-project N rule is different only because the preceding Det has already created an NP syntactic structure in the `imaginal` buffer that the N needs to update before creating its own unary-branching structure in the same buffer. We do not list the rule here; the complete set of rules is linked to in Appendix 9.6.2.

The `key_press` task consists of only one rule: the `"press spacebar"` rule in (38) below. This rule adds a command to press the spacebar to the `manual` buffer (lines 13–16). Once that is done, we revert to the default task of `parsing` (line 12).

(38)
```
parser.productionstring(name="press spacebar", string="""        1
    =g>                                                         2
    isa                 parsing_goal                            3
    task                key_press                               4
    ?manual>                                                    5
    state               free                                    6
    ?retrieval>                                                 7
    state               free                                    8
    ==>                                                         9
    =g>                                                         10
    isa                 parsing_goal                            11
    task                parsing                                 12
    +manual>                                                    13
    isa                 _manual                                 14
    cmd                 press_key                               15
    key                 'space'                                 16
""")                                                            17
```

Finally, when there are no more words to be read on the virtual screen, we end the syntax/semantics parsing process with the `"finished: no visual input"` rule in (39) below. This rule flushes all the goal/goal-like buffers.

(39)
```
parser.productionstring(name="finished: no visual input", string="""       1
    =g>                                                                    2
    isa                 parsing_goal                                       3
    task                parsing                                            4
    stack1              None                                               5
    ?visual>                                                               6
    state               free                                               7
    buffer              empty                                              8
    ?manual>                                                               9
    state               free                                               10
    buffer              empty                                              11
    ==>                                                                    12
    ~g>                                                                    13
    ~imaginal>                                                             14
    ~discourse_context>                                                    15
    ~unresolved_discourse>                                                 16
""")                                                                       17
                                                                           18
```

9.3.3 Phrase Structure Rules

The project and project-and-complete phrase structure rules encode most of the syntactic parsing and all the semantic parsing work. Consequently, these rules tend to have relative large lists of actions to be executed. We will only discuss here some of the most important phrase structure rules. The remaining ones, linked to in Appendix 9.6.2, have the same kind of structure and should be straightforward to understand.

Consider first the `"project: NP ==> Det N"` rule in (40) below. This rule is triggered once a determiner is shifted and projected, so the `found` feature stores a `Det` value (line 10). Another important condition for this rule is that the top parsing goal is to parse an `S` (line 5). That is, this rule is triggered by determiners in subject position. Determiners in object position trigger a different rule (`"project and complete: NP ==> Det N"`; see Appendix 9.6.2), which is very similar except that the top parsing goal is an `NP` rather than an `S`.

(40)
```
parser.productionstring(name="project: NP ==> Det N", string="""       1
    =g>                                                                2
    isa                 parsing_goal                                   3
    task                parsing                                        4
    stack1              S                                              5
    arg_stack1          =a1                                            6
    right_edge_stack1   =re1                                           7
    right_edge_stack2   =re2                                           8
    parsed_word         =w                                             9
    found               Det                                            10
    dref_peg            =dref_peg                                      11
    drs_peg             =drs_peg                                       12
    embedding_level     =el                                            13
    =retrieval>                                                        14
    isa                 word                                           15
    pred1               =p1                                            16
    pred2               =p2                                            17
    ==>                                                                18
    =g>                                                                19
    isa                 parsing_goal                                   20
    task                move_dref_peg                                  21
    stack1              N                                              22
    stack2              NP                                             23
    stack3              S                                              24
    found               None                                           25
    parsed_word         None                                           26
    arg_stack1          =dref_peg                                      27
    arg_stack2          =a1                                            28
    +imaginal>                                                         29
    isa                 parse_state                                    30
    node_cat            NP                                             31
    daughter1           Det                                            32
    daughter2           N                                              33
    mother              =re1                                           34
    mother_of_mother    =re2                                           35
    +discourse_context>                                                36
    isa                 drs                                            37
    dref                =dref_peg                                      38
    arg1                =dref_peg                                      39
    drs                 =drs_peg                                       40
    embedding_level     =el                                            41
    discourse_status    at_issue                                       42
    ~retrieval>                                                        43
""")                                                                   44
```

The `"project: NP ==> Det N"` rule in (40) also conditions on the `retrieval` buffer making available the semantics of the word we just parsed, specifically, the two predicates `=p1` and `=p2` it contributes (lines 14–17). This is spurious in the present case because we have just parsed a determiner, which does not contribute any predicates, but we include it for uniformity with the phrase structure rules for proper names, nouns, verbs, pronouns etc.

The `"project: NP ==> Det N"` rule in (40) triggers three main parsing actions: (i) it updates the `goal` buffer, thereby setting the context for subsequent parsing rules (lines 19–28); (ii) it updates the `imaginal` buffer with the expected NP node with two daughters (Det and N), and it attaches the NP to the top right-edge attachment point (the S node) (lines 29–35); (iii) finally, it updates the `discourse_context` buffer with a new DRS (lines 36–42).

The new DRS has a similar structure to the sub-DRSs contributed by indefinites that we discussed in the previous chapter (Chap. 8): we introduce a new individual-level dref (line 38) and we plug it in as the argument of the upcoming N (line 39).

However, in contrast to what we did in the previous chapter, we mark the current sub-DRS as being part of a larger DRS (i.e., part of a larger propositional update) by means of a `drs` slot which has the current DRS dref `=drs_peg` as its value (line 40). Thus, the fact that this sub-DRS is part of a larger main DRS is only implicit in this representation, unlike the more explicitly hierarchical structure we built in the previous chapter. We prefer this flatter structure for sub-DRSs and main DRSs in this chapter because it is simpler to use, and sufficient for our purposes.

However, because we are modeling pronoun and presupposition resolution, we need to keep track of two other semantic features: the embedding level of the current sub-DRS (line 41), which is crucial for the resolution process, and the discourse status of the current sub-DRS (line 42). The discourse status is `at_issue`, to be distinguished from either (i) the `unresolved` discourse status associated with unresolved pronouns/presuppositions, or (ii) the `presupposed` discourse status associated with resolved pronouns/presuppositions.

Given these `imaginal` and `discourse_context` buffer updates, we update the goal buffer in a variety of ways:

- we update the task to `move_dref_peg` (line 21 in (40)): we have just introduced a new dref indexed with the current `dref_peg`, so we need to move the peg to the next position in preparation for subsequent indefinites;
- we set up the stack of parsing goals in the expected way (lines 22–24): we have just parsed a Det in subject position, so we expect to parse an N next, after which we expect to finish parsing the subject NP, and once that is completed, we revert to the initial goal of parsing an S;
- we reset the `found` and `parsed_word` features to `None` (lines 25–26) to indicate that we are finished parsing the current word;
- we push the dref we just introduced on the argument stack (lines 27–28) so that it is available as an argument for the predicate(s) introduced by the upcoming VP.

Finally, we flush the retrieval buffer (line 43 in (40)), since we have no further use for the lexical information associated with the current word.

The syntactic and semantic parsing actions triggered by the project-NP rule set up the context for the `"project and complete: N"` rule in (41) below. If the syntactic category at the top of the goal stack is N (line 5) and we have just shifted and projected an N (line 8), the semantics of which is available in the retrieval buffer (lines 9–12), we take two main parsing actions. On one hand, we add the two predicates `=p1` and `=p2` lexically contributed by the N to the DRS in the `discourse_context` buffer, which was contributed by the preceding indefinite determiner (lines 23–26). On the other hand, we pop the N goal off the goal-buffer stack and reset the `found` and `parsed_word` features back to `None`. To wrap up, we do some cognitive-state clean-up and flush the `retrieval` and `imaginal` buffers (lines 27–28).

(41)
```
parser.productionstring(name="project and complete: N", string="""     1
    =g>                                                                  2
    isa                parsing_goal                                      3
    task               parsing                                           4
    stack1             N                                                 5
    stack2             =s2                                               6
    stack3             =s3                                               7
    found              N                                                 8
    =retrieval>                                                          9
    isa                word                                             10
    pred1              =p1                                              11
    pred2              =p2                                              12
    ?discourse_context>                                                 13
    buffer             full                                             14
    ==>                                                                 15
    =g>                                                                 16
    isa                parsing_goal                                     17
    stack1             =s2                                              18
    stack2             =s3                                              19
    stack3             None                                             20
    found              None                                             21
    parsed_word        None                                             22
    =discourse_context>                                                 23
    isa                drs                                              24
    pred1              =p1                                              25
    pred2              =p2                                              26
    ~retrieval>                                                         27
    ~imaginal>                                                          28
""")                                                                    29
```

At this point, we have fully parsed the subject NP, so we need to pop that goal off the top of the goal-buffer stack. Moreover, we have found the left corner of the S, namely the subject NP, so we can also pop the S goal off the stack and replace it with the goal of finding the VP that will complete the S. The `"project and complete: S ==> NP VP"` in (42) below triggers all these parsing actions, and also builds the binary-branching [$_S$NP VP] structure in the `imaginal` buffer.

(42)
```
parser.productionstring(name="project and complete: S ==> NP VP",       1
 string="""                                                              2
    =g>                                                                  3
    isa                parsing_goal                                      4
    task               parsing                                           5
    stack1             NP                                                6
    stack2             S                                                 7
    stack3             =s3                                               8
    right_edge_stack1  =re1                                              9
    right_edge_stack2  =re2                                             10
    right_edge_stack3  =re3                                             11
    right_edge_stack4  =re4                                             12
    ==>                                                                 13
    =g>                                                                 14
    isa                parsing_goal                                     15
    stack1             VP                                               16
    stack2             =s3                                              17
    stack3             None                                             18
    right_edge_stack1  VP                                               19
    right_edge_stack2  =re1                                             20
    right_edge_stack3  =re2                                             21
    right_edge_stack4  =re3                                             22
    found              None                                             23
    parsed_word        None                                             24
    +imaginal>                                                          25
    isa                parse_state                                      26
    node_cat           S                                                    27
    daughter1          NP                                                   28
    daughter2          VP                                                   29
    ~retrieval>                                                             30
    ~imaginal>                                                              31
    ~discourse_context>                                                     32
""")                                                                        33
```

At this point, various project-and-complete-VP rules can be triggered, depending on the syntactic category of the finite verb. We will only discuss here the simplest case, namely intransitive verbs, and then work through a simple example. The `"project and complete: VP ==> Vi"` rule is provided in (43) below. This rule is triggered if the top syntactic category in the goal-buffer stack is VP (line 5), and we have just shifted and projected an intransitive verb (line 6), the semantics of which is available in the `retrieval` buffer (lines 16–18).

(43)
```
parser.productionstring(name="project and complete: VP ==> Vi", string="""  1
    =g>                                                                     2
    isa                parsing_goal                                         3
    task               parsing                                              4
    stack1             VP                                                   5
    found              Vi                                                   6
    parsed_word        =w                                                   7
    right_edge_stack1  VP                                                   8
    right_edge_stack2  =re2                                                 9
    right_edge_stack3  =re3                                                 10
    right_edge_stack4  =re4                                                 11
    arg_stack1         =a1                                                  12
    drs_peg            =drs_peg                                             13
    event_peg          =ev_peg                                              14
    embedding_level    =el                                                  15
    =retrieval>                                                             16
    isa                word                                                 17
    pred1              =p1                                                  18
    ==>                                                                     19
    =g>                                                                     20
    isa                parsing_goal                                         21
    task               move_event_peg                                       22
    stack1             None                                                 23
    found              None                                                 24
    parsed_word        None                                                 25
    right_edge_stack1  =re2                                                 26
    right_edge_stack2  =re3                                                 27
    right_edge_stack3  =re4                                                 28
    right_edge_stack4  None                                                 29
    +imaginal>                                                              30
    isa                parse_state                                          31
    mother             =re2                                                 32
    mother_of_mother   =re3                                                 33
    node_cat           VP                                                   34
    daughter1          Vi                                                   35
    lex_head           =w                                                   36
    +discourse_context>                                                     37
    isa                drs                                                  38
    dref               =ev_peg                                              39
    event_arg          =ev_peg                                              40
    arg1               =a1                                                  41
    pred1              =p1                                                  42
    drs                =drs_peg                                             43
    embedding_level    =el                                                  44
    discourse_status   at_issue                                             45
    ~retrieval>                                                             46
    ~imaginal>                                                              47
    ~discourse_context>                                                     48
""")                                                                        49
```

If these pre-conditions are met, the `"project and complete: VP ==> Vi"` rule in (43) triggers three main parsing actions, just like the project-NP rule in (40) above, or most other rules associated with words contributing essential semantic information and operations. Just as before, the three main parsing actions involve: (i) the `goal` buffer (lines 20–29); (ii) the `imaginal` buffer, where new syntactic information is encoded (lines 30–36); (iii) the `discourse_context` buffer, where new semantic information is encoded (lines 37–45).

The `imaginal` buffer update (lines 30–36 in (43) simply builds a unary-branching structure [$_{\text{VP}}$Vi] and specifies its lexical head and its attachment point in the larger syntactic structure.

The `discourse_context` buffer update on lines 37–45 is more substantial:

- we introduce a new DRS and mark it as a sub-DRS of the current main DRS by setting the `drs` feature to the current `=drs_peg` (line 43), its `embedding_level` feature to the current embedding level `=el` (line 44), and its discourse status to `at_issue` (line 45);
- since this DRS is contributed by a verb, we introduce a new event dref and index it with the current event dref peg `=ev_peg` (line 39);
- the intransitive verb contributes a predicate `=p1` (line 42) that takes two arguments: an event argument that is set to the newly introduced event dref `=ev_peg` (line 40), and an entity argument that is set to the dref `=a1` previously introduced by the subject and stored at the top of the goal-buffer argument stack (line 41).

With the syntactic and semantic updates in place, the `goal` buffer can be updated accordingly (lines 20–29):

- since we just introduced a new event dref, we need to update the event dref peg, so the new task is set to `move_event_peg` (line 22);
- we have just completely finished parsing the intransitive verb, so we pop the VP category off the goal-buffer stack (line 23) and reset the `found` and `parsed_word` features to `None` (lines 24–25);
- we also pop the S node off the right-edge stack since this node is not available for future attachments anymore (lines 26–29).

We are now ready to parse a simple example. When the model reads the sentence in (44) below word by word (as in a self-paced reading task), it goes through a cognitive process whose temporal trace is provided in (45). To run the model on the sentence in (44) and obtain the temporal trace, uncomment the relevant line in the `run_parser.py` file (linked to in Appendix 9.6.3) and run the file in the terminal with the command:

- `python3 run_parser.py`

The full `pyactr` output is very detailed, so we edit it down significantly in (45) below to be able to focus on the main steps of the incremental interpretation process.

(44) A woman smiled.

(45)

```
****Environment: {1: {’text’: ’A’, ’position’: (320, 180)}}
(0.0255, ’PROCEDURAL’, ’RULE FIRED: encode word’)
(0.038, ’PROCEDURAL’, ’RULE FIRED: retrieve word’)
(0.1762, ’retrieval’, ’RETRIEVED: word(cat=Det, form=A, pred1=, pred2=)’)
(0.1887, ’PROCEDURAL’, ’RULE FIRED: shift and project word (not N)’)
(0.1887, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=A,
          node_cat=Det)’)
(0.2012, ’PROCEDURAL’, ’RULE FIRED: press spacebar’)
(0.2137, ’PROCEDURAL’, ’RULE FIRED: project: NP ==> Det N’)
(0.2137, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=Det,
          daughter2=N, daughter3=, mother=S, node_cat=NP)’)
(0.2137, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=x1, arg2=,
          discourse_status=at_issue, dref=x1, drs=d1, embedding_level=0,
          event_arg=, pred1=, pred2=)’)
(0.2262, ’PROCEDURAL’, ’RULE FIRED: move entity/individual dref peg to x2’)
(0.3512, ’manual’, ’KEY PRESSED: SPACE’)
****Environment: {1: {’text’: ’woman’, ’position’: (320, 180)}}
(0.3712, ’PROCEDURAL’, ’RULE FIRED: encode word’)
(0.3837, ’PROCEDURAL’, ’RULE FIRED: retrieve word’)
(0.5302, ’retrieval’, ’RETRIEVED: word(cat=N, form=woman,
          pred1=pred(arity=1, constant_name=_woman_),
          pred2=pred(arity=1, constant_name=_female_))’)
(0.5427, ’PROCEDURAL’, ’RULE FIRED: shift and project N’)
(0.5427, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=woman,
          node_cat=N)’)
(0.5552, ’PROCEDURAL’, ’RULE FIRED: press spacebar’)
(0.5677, ’PROCEDURAL’, ’RULE FIRED: project and complete: N’)
(0.5677, ’discourse_context’, ’MODIFIED’)
(0.5802, ’PROCEDURAL’, ’RULE FIRED: project and complete: S ==> NP VP’)
(0.5802, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=NP,
          daughter2=VP, node_cat=S)’)
(0.7052, ’manual’, ’KEY PRESSED: SPACE’)
****Environment: {1: {’text’: ’smiled’, ’position’: (320, 180)}}
(0.7285, ’PROCEDURAL’, ’RULE FIRED: encode word’)
(0.741, ’PROCEDURAL’, ’RULE FIRED: retrieve word’)
(0.8899, ’retrieval’, ’RETRIEVED: word(cat=Vi, form=smiled,
          pred1=pred(arity=event_plus_1, constant_name=_smile_))’)
(0.9024, ’PROCEDURAL’, ’RULE FIRED: shift and project word (not N)’)
(0.9024, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=smiled,
          node_cat=Vi)’)
0.9149, ’PROCEDURAL’, ’RULE FIRED: press spacebar’)
(0.9274, ’PROCEDURAL’, ’RULE FIRED: project and complete: VP ==> Vi’)
(0.9274, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=Vi,
          lex_head=smiled, mother=S, node_cat=VP)’)
(0.9274, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=x1,
          discourse_status=at_issue, dref=e1, drs=d1,
          embedding_level=0, event_arg=e1,
          pred1=pred(arity=event_plus_1, constant_name=_smile_))’)
(0.9399, ’PROCEDURAL’, ’RULE FIRED: move event dref peg to e2’)
(1.2274, ’PROCEDURAL’, ’RULE FIRED: finished: no visual input’)

Parse states in declarative memory at the end of the simulation
ordered by time of (re)activation:
0.2137   parse_state(daughter1=A, node_cat=Det)
0.5427   parse_state(daughter1=Det, daughter2=N, lex_head=woman,
                     mother=S, node_cat=NP)
0.5677   parse_state(daughter1=woman, node_cat=N)
0.9024   parse_state(daughter1=NP, daughter2=VP, node_cat=S)
0.9274   parse_state(daughter1=smiled, node_cat=Vi)
1.2274   parse_state(daughter1=Vi, lex_head=smiled, mother=S,
                     node_cat=VP)

DRSs in declarative memory at the end of the simulation
ordered by time of (re)activation:
0.5802   drs(arg1=x1, discourse_status=at_issue, dref=x1,
             drs=d1, embedding_level=0,
             pred1=pred(arity=1, constant_name=_woman_),
             pred2=pred(arity=1, constant_name=_female_))
1.2274   drs(arg1=x1, discourse_status=at_issue, dref=e1,
             drs=d1, embedding_level=0, event_arg=e1,
             pred1=pred(arity=event_plus_1, constant_name=_smile_))
```

Line 1 of the temporal trace in (45) indicates that the first word, namely the indefinite determiner *A*, is displayed on the virtual screen from which the model takes its visual input. This word is encoded 25.5 ms later (line 2), after which a retrieval request for its lexical information is placed (line 3, time: 38 ms after the start of the cognitive process). The lexical information is retrieved after $\approx$140 ms (line 4), then the word is shifted and projected (line 5), and as a result, our first syntactic structure $[_{\text{Det}}$A] is built in the imaginal buffer (lines 6–7). At this point, the cognitive process branches into two sub-processes running in parallel: (i) a motor sub-process that will culminate in pressing the spacebar to reveal the next work (line 8), and (ii) the continuation of the incremental parsing process triggered by the indefinite determiner *A* (lines 9–15).

The incremental parsing process continues with the project-NP rule (line 9), which results in the creation of an $[_{\text{NP}}$Det N] syntactic structure in the imaginal buffer (lines 10–11), and the creation of our first DRS in the `discourse_context` buffer (lines 12–14). Using the familiar DRT format, the DRS can be represented as follows:

(46) DRS contributed by the indefinite *A*:

x_1
PREDICATES- NOT- YET- SPECIFIED [part of main DRS d_1] [at-issue] [embedding level: 0]

Given that we just introduced a new dref x_1 for entities/individuals, we have to move the entity dref peg to the next position x_2 (line 15 in (45)). After this, the incremental interpretation sub-process has nothing left to do, so we wait until the motor sub-process completes and the spacebar is pressed (line 16).

At that point, the next word *woman* is displayed on the virtual screen (line 17), and we go through the same cycle of encode-retrieve-project rules for the new word (lines 18–31). Specifically, we encode the word (line 18) and place a retrieval request for its lexical information (line 19). The lexical entry for the noun *woman* is retrieved at $\approx$530 ms after the start of the entire cognitive process (lines 20–22). Notably, the noun contributes two predicates: `_woman_` and `_female_`, both of arity 1. The explicit gender specification is useful for pronoun resolution.

The noun is then shifted and projected (line 23) and, as a result, the unary branching structure $[_{\text{N}}$woman] is created in the imaginal buffer (lines 24–25). At this point, the cognitive process branches again into a motor process that will culminate in pressing the spacebar (line 26) and the continuation of the incremental parsing process. Incremental parsing continues with the project-and-complete-N rule (line 27), which results in an update of the DRS contributed by the indefinite determiner *A* and stored in the `discourse_context` buffer (line 28). The DRS is updated with the two predicates contributed by the noun *woman*, and can be represented in the familiar DRT format as follows.

(47) DRS contributed by the indefinite *A* and updated by the noun *woman*:

x_1
WOMAN(x_1)
FEMALE(x_1)
[part of main DRS d_1]
[at-issue]
[embedding level: 0]

The project-S rule is then fired (line 29) and the binary-branching structure [$_S$NP VP] is created in the imaginal buffer (lines 30–31). This completes the sequence of incremental interpretation steps triggered by the noun *woman*. The process then waits for the motor sub-process to complete and the spacebar to be pressed, which happens ≈120 ms later (line 32).

At this point, the final word *smiled* is displayed on the virtual screen (line 33). After it is encoded (line 34) and the request for its lexical entry is placed (line 35), we have access to its lexical information in the retrieval buffer (lines 36–37). The most notable aspect of this lexical entry is the arity `event_plus_1` of the predicate `_smile_` (line 37): this simply means that the predicate `_smile_` takes an event argument and, in addition, an entity argument. The intransitive verb is then shifted and projected (line 38), at which point the unary branching syntactic structure [$_{Vi}$smiled] is built in the imaginal buffer (lines 39–40).

Once again, and for the final time, the cognitive process branches into a motor sub-process that will culminate with a spacebar press (line 41) and a sub-process that continues with the incremental parsing triggered by the intransitive verb *smiled*. The parsing process continues with the project-and-complete-VP rule (line 42), which simultaneously creates a syntactic structure [$_{VP}$Vi] in the imaginal buffer (lines 43–44) and a DRS in the `discourse_context` buffer (lines 45–48). This DRS can be represented in the usual DRT format as shown below.

(48) DRS contributed by the intransitive verb *smiled*:

e_1
SMILE(e_1, x_1)
[part of main DRS d_1]
[at-issue]
[embedding level: 0]

The parsing process is basically done, so once the event-dref peg is updated (line 49), the `"finished"` rule ends the entire cognitive process because of the lack of visual input (no more words on the virtual screen). For convenience, we list the syntactic structures built during the parsing process together with their time stamps on lines 54–61 in (45), and the two DRSs with their time stamps on lines 65–71.

The model includes a variety of other rules—for transitive verbs, prepositional verbs, NPs in object position, adjectives etc. They are all available in the file linked to in Appendix 9.6.2. We encourage you to run the model on the variety of sentences in the `run_parser.py` file (Appendix 9.6.3) and examine the temporal-trace outputs to understand the time-course predictions of the syntax/semantics interpretation process implemented by this model.

9.3.4 *Rules for Conjunctions and Anaphora Resolution*

We are now ready to model basic bi-clausal examples, specifically, conjunctions. The project-and-complete rule for *and* is provided in (49) below. The rule starts a new sentence S, that is, the second conjunct, by placing the category S at the top of the goal-buffer stack (line 13), advancing the DRS dref peg (line 12: the task is updated to `move_drs_peg`), and setting the embedding level for the second conjunct to 1 (line 22).

Incrementing the embedding level ensures that drefs in the second conjunct are not available as antecedents for pronouns in the first conjunct. In general, we use embedding levels to model both the explicit and the implicit aspects of the discourse accessibility relation used in DRT. The fact that discourse referents in the second conjunct cannot serve as antecedents to pronouns in the first conjunct is not explicitly encoded in the accessibility relation defined in Kamp and Reyle (1993), but it is a by-product of the DRS construction algorithm that requires the DRS construction for the second conjunct to take place in the context of the DRS constructed based on the first conjunct. In our model, we use embedding level uniformly to constrain dref accessibility, both in conjunctions and conditionals.

Finally, the *and* rule creates a (non-headed) ternary-branching structure $[_{\text{ConjS}}$S Conj S] in the imaginal buffer (lines 23–31).

(49)
```
parser.productionstring(name="project and complete: and", string="""     1
    =g>                                                                  2
    isa                 parsing_goal                                     3
    task                parsing                                          4
    found               Conj                                             5
    parsed_word         and                                              6
    parsed_word         =w                                               7
    embedding_level     0                                                8
    ==>                                                                  9
    =g>                                                                  10
    isa                 parsing_goal                                     11
    task                move_drs_peg                                     12
    stack1              S                                                13
    arg_stack1          None                                             14
    arg_stack2          None                                             15
    found               None                                             16
    parsed_word         None                                             17
    right_edge_stack1   S                                                18
    right_edge_stack2   ConjS                                            19
    right_edge_stack3   None                                             20
    right_edge_stack4   None                                             21
    embedding_level     1                                                22
    +imaginal>                                                           23
    isa                 parse_state                                      24
    daughter1           S                                                25
    daughter2           Conj                                             26
    daughter3           S                                                27
    node_cat          ConjS                                    28
    mother            None                                     29
    mother_of_mother  None                                     30
    lex_head          =w                                       31
    ~discourse_context>                                        32
""")                                                           33
```

We need to introduce three more rules, related to pronoun resolution, before we can go through an example. The first rule is the `"project: NP ==> PRO"` rule in (50) below, which completes the syntax/semantics parsing of pronouns in subject position. Given that a PRO has just been found (line 11) and is available in the retrieval buffer (lines 15–18), we take the usual three types of parsing actions and update the goal, imaginal and `discourse_context` buffers. In addition, we also add an unresolved DRS to the `unresolved_discourse` buffer. Let's discuss them in turn.

(50)
```
parser.productionstring(name="project: NP ==> PRO", string="""     1
    =g>                                                        2
    isa               parsing_goal                             3
    task              parsing                                  4
    stack1            S                                        5
    stack2            =s2                                      6
    arg_stack1        =a1                                      7
    right_edge_stack1 =re1                                     8
    right_edge_stack2 =re2                                     9
    parsed_word       =w                                       10
    found             PRO                                      11
    dref_peg          =dref_peg                                12
    drs_peg           =drs_peg                                 13
    embedding_level   =el                                      14
    =retrieval>                                                15
    isa               word                                     16
    pred1             =p1                                      17
    pred2             =p2                                      18
    ==>                                                        19
    =g>                                                        20
    isa               parsing_goal                             21
    task              move_dref_peg                            22
    stack1            NP                                       23
    stack2            S                                        24
    stack3            =s2                                      25
    arg_stack1        =dref_peg                                26
    arg_stack2        =a1                                      27
    found             None                                     28
    +imaginal>                                                 29
    isa               parse_state                              30
    node_cat          NP                                       31
    daughter1         PRO                                      32
    mother            =re1                                     33
    mother_of_mother  =re2                                     34
    lex_head          =w                                       35
    +discourse_context>                                        36
    isa               drs                                      37
    dref              =dref_peg                                38
    arg1              =dref_peg                                39
    pred1             =p2                                      40
    drs               =drs_peg                                 41
    embedding_level   =el                                      42
    discourse_status  at_issue                                 43
    +unresolved_discourse>                                     44
    isa               drs                                      45
    arg1              =dref_peg                                46
    arg2              UNKNOWN                                  47
    pred1             =p1                                      48
    pred2             =p2                                      49
    drs               =drs_peg                                 50
```

```
        embedding_level   =el                                            51
        discourse_status  unresolved                                     52
        ~retrieval>                                                      53
        ~imaginal>                                                       54
        ~discourse_context>                                              55
    """)                                                                 56
```

The imaginal buffer update on lines 29–35 in (50) is straightforward: we create the expected unary branching [$_{\text{NP}}$PRO] structure.

The DRS created in the `discourse_context` buffer on lines 36–43 follows the analysis of pronouns in Kamp and Reyle (1993). We introduce a new dref (line 38) and predicate the pronoun gender of it (line 40).

The `unresolved_discourse` buffer stores a DRS that encodes the unresolved presupposition of the pronoun (lines 44–52):

- the main predicate `=p1` contributed by the pronoun (line 48) is identity (`EQUALS`), relating the dref introduced by the pronoun (line 46) and an `UNKNOWN` second dref that needs to be retrieved;
- the `UNKNOWN` dref is the antecedent dref that needs to be found to complete the pronoun resolution;
- the antecedent dref needs to be accessible, which is why we keep track of the embedding level of the pronoun (line 51), and it also needs to satisfy the gender predicate `=p2` contributed by the pronoun (line 49);
- finally, the entire DRS is marked as having an `unresolved` discourse status (line 52).

The goal buffer is updated in the expected way after parsing an NP in subject position (lines 20–28): the task is updated to `move_dref_peg` (line 22), the dref introduced by the pronoun is added to the top of the argument stack (line 26), and NP becomes the top goal on the goal-buffer stack (line 23) in preparation for the project-S rule.

After fully parsing a pronoun, the cognitive state satisfies the conditions for attempting to resolve it. There are three different rules for anaphoric pronoun resolution, depending on the embedding level of the pronoun. We only discuss here the rule for embedding level 1, provided in (51) below. The other rules, linked to in Appendix 9.6.2, are very similar. The rule in (51) is triggered only if the `unresolved_discourse` buffer contains an unresolved DRS (lines 15–21).

Pronoun resolution rules are fired when other, higher-ranked tasks have already been completed (lines 5–11 in 51). Once these higher-ranked tasks are completed, pronoun resolution has high priority (it has a utility of 5—line 33). As we mentioned before, ordering rule firing preferences in this fashion (multiple negative specifications for the `task` slot & manually setting the utility to a non-default, i.e., non-0, value) is not cognitively realistic, and does not scale up to larger systems of rules. The conditions for these rules, as well as their utility, should emerge as a result of a learning algorithm that leverages both production compilation (for new rule generation) and reinforcement learning (for utility 'tuning').

The rule triggers two actions. The main action is a retrieval request for a suitable antecedent for the pronoun (lines 26–32): we need to retrieve a DRS in which a new dref was introduced (line 28) that was not an event dref (line 29: `event_arg None` ensures that only entities, not events, will be considered as possible antecedents). Furthermore, this DRS should not be indexed with the same DRS dref (line 30): this is a way to enforce Binding Principle B (Chomsky 1981). Finally, the antecedent should have an embedding level higher than 2 (line 31) and an `at_issue` discourse status (line 32).

The second action is updating the task so that the resolution attempt concludes with this retrieval request (line 25), and we do not keep attempting to retrieve an antecedent again and again.

(51)
```
parser.productionstring(name="attempting to resolve pronoun;\      1
pronoun at embedding level 1", string="""                           2
    =g>                                                             3
    isa                 parsing_goal                                4
    task                ~reading_word                               5
    task                ~move_dref_peg                              6
    task                ~move_event_peg                             7
    task                ~attempting_to_resolve_PRO                  8
    task                ~attempting_to_resolve_cataphoric_PRO       9
    task                ~stop_resolution_attempt_PRO               10
    task                ~if_reanalysis                             11
    found               None                                       12
    ?retrieval>                                                    13
    state               free                                       14
    =unresolved_discourse>                                         15
    isa                 drs                                        16
    arg2                UNKNOWN                                    17
    pred2               =p2                                        18
    drs                 =drs                                       19
    embedding_level     1                                          20
    discourse_status    unresolved                                 21
    ==>                                                            22
    =g>                                                            23
    isa                 parsing_goal                               24
    task                stop_resolution_attempt_PRO                25
    +retrieval>                                                    26
    isa                 drs                                        27
    dref                ~None                                      28
    event_arg           None                                       29
    drs                 ~=drs                                      30
    embedding_level     ~2                                         31
    discourse_status    at_issue                                   32
""", utility=5)                                                    33
```

The retrieval request for an antecedent either succeeds or fails. If it succeeds, the rule in (52) below fires. The rule takes the antecedent available in the retrieval buffer (lines 6–10) and the unresolved presupposition in the `unresolved_discourse` buffer (lines 11–18) and merges information from them into a new DRS added to the `discourse_context` buffer.

This DRS encodes the resolved pronominal presupposition: it basically takes the unresolved presupposition from the `unresolved_discourse` buffer and specifies its second, `UNKNOWN` argument to be the same dref as the dref of the antecedent DRS available in the retrieval buffer (line 27). The discourse status of the resolved presupposition is marked as `presupposed` (line 32).

With the pronoun resolved, we can flush the `unresolved_discourse` and retrieval buffers (lines 33–34).

(52)
```
parser.productionstring(name="resolution of PRO succeeded", string="""     1
    =g>                                                                    2
    isa                 parsing_goal                                       3
    task                stop_resolution_attempt_PRO                        4
    embedding_level     =el                                                5
    =retrieval>                                                            6
    isa                 drs                                                7
    dref                ~None                                              8
    dref                =dref                                              9
    pred2               =p2                                               10
    =unresolved_discourse>                                                11
    isa                 drs                                               12
    arg1                =a1                                               13
    arg2                UNKNOWN                                           14
    pred1               =p1                                               15
    pred2               =p2                                               16
    drs                 =drs                                              17
    discourse_status    unresolved                                        18
    ==>                                                                   19
    =g>                                                                   20
    isa                 parsing_goal                                      21
    task                parsing                                           22
    entity_cataphora    None                                              23
    +discourse_context>                                                   24
    isa                 drs                                               25
    arg1                =a1                                               26
    arg2                =dref                                             27
    pred1               =p1                                               28
    pred2               =p2                                               29
    drs                 =drs                                              30
    embedding_level     =el                                               31
    discourse_status    presupposed                                       32
    ~retrieval>                                                           33
    ~unresolved_discourse>                                                34
""")                                                                      35
```

If the retrieval request for an antecedent fails, we trigger the rule in (53) below. The rule checks that the unsuccessfully resolved presupposition targets an entity, not an event (line 12): we check that `arg2` is `UNKNOWN`. For events, we will see that the `arg1` slot will be marked as `UNKNOWN`. The retrieval of a suitable entity dref has failed, but we do not simply mark the pronoun as unresolved and move on: we assume that the pronoun is in fact cataphoric, so we set the `entity_cataphora` feature to `True` (line 18). This will ensure that when entity drefs will be subsequently introduced in discourse, a cataphoric search will be triggered to check if they could be suitable antecedents for the unresolved pronoun. Finally, the rule flushes the `unresolved_discourse` and retrieval buffers (lines 20–21).

(53)
```
parser.productionstring(name="resolution of PRO failed: no antecedent",   1
string="""                                                                 2
    =g>                                                                    3
    isa                 parsing_goal                                       4
    task                stop_resolution_attempt_PRO                        5
    ?retrieval>                                                            6
    state               error                                              7
    ?unresolved_discourse>                                                 8
    buffer              full                                               9
    =unresolved_discourse>                                                10
    isa                 drs                                               11
    arg2                UNKNOWN                                           12
    discourse_status    unresolved                                        13
    ==>                                                                   14
    =g>                                                                   15
    isa                 parsing_goal                                      16
    task                parsing                                           17
    entity_cataphora    True                                              18
```

```
        found              no_antecedent
        ~unresolved_discourse>
        ~retrieval>
    """)
```

There is another way that pronoun resolution could fail. Recall that the retrieval request for an antecedent placed in (51) above does not constrain the gender of the antecedent, it only constrains its embedding level and DRS peg (in addition to requiring the `dref` slot to be non-empty and the `event_arg` slot to be empty). We could, therefore, retrieve an antecedent dref that is suitable with respect to all features, but that does not match in gender. This is very much like the process of retrieving foil sentences in the fan experiment and the corresponding fan model discussed in the previous chapter (Chap. 8).

The `"resolution of PRO failed: antecedent with non-matching gender"` rule in (54) below takes care of this case: if the retrieved DRS has a certain gender specification `=p2` (line 10) and the unresolved pronoun presupposition has a different gender specification `~=p2` (line 13), we declare the pronoun unresolved and assume that it is a cataphoric pronoun (line 19).

(54)
```
parser.productionstring(name="resolution of PRO failed:\            1
antecedent with non-matching gender", string="""                    2
    =g>                                                             3
    isa                parsing_goal                                 4
    task               stop_resolution_attempt_PRO                  5
    ?unresolved_discourse>                                          6
    buffer             full                                         7
    =retrieval>                                                     8
    isa                drs                                          9
    pred2              =p2                                          10
    =unresolved_discourse>                                          11
    isa                drs                                          12
    pred2              ~=p2                                         13
    discourse_status   unresolved                                   14
    ==>                                                             15
    =g>                                                             16
    isa                parsing_goal                                 17
    task               parsing                                      18
    entity_cataphora   True                                         19
    found              no_antecedent                                20
    ~retrieval>                                                     21
    ~unresolved_discourse>                                          22
""")                                                                23
```

To bring all these rules together, let us work through two examples of conjoined discourses, one in which the pronoun resolution succeeds and one in which the resolution fails because of a gender mismatch. The following subsection will discuss cataphora, so we will show an example of a ‘no retrieved antecedent’ resolution failure at that point.

Let us first simulate the example in (55) below. The temporal trace is provided in (56). We omit the parsing steps associated with the first conjunct since they are identical to the ones in (45) above, except for minor random noise in visual, motor and retrieval timings. To obtain the temporal trace, we uncomment the relevant sentence in the `run_parser.py` file and run the file with the command `python3 run_parser.py`, as we did before.

(55) A woman smiled and she left.

(56)
```
(1.0599, ’manual’, ’KEY PRESSED: SPACE’)
****Environment: {1: {’text’: ’and’, ’position’: (320, 180)}}
(1.0787, ’PROCEDURAL’, ’RULE FIRED: encode word’)
(1.0912, ’PROCEDURAL’, ’RULE FIRED: retrieve word’)
(1.2415, ’retrieval’, ’RETRIEVED: word(cat=Conj, form=and)’)
(1.2540, ’PROCEDURAL’, ’RULE FIRED: shift and project word (not N)’)
(1.2540, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=and,
          node_cat=Conj)’)
(1.2665, ’PROCEDURAL’, ’RULE FIRED: press spacebar’)
(1.279, ’PROCEDURAL’, ’RULE FIRED: project and complete: and’)
(1.279, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=S,
         daughter2=Conj, daughter3=S, lex_head=and, mother=None,
         mother_of_mother=None, node_cat=ConjS)’)
(1.2915, ’PROCEDURAL’, ’RULE FIRED: move DRS/propositional dref peg to d2’)
(1.4165, ’manual’, ’KEY PRESSED: SPACE’)
****Environment: {1: {’text’: ’she’, ’position’: (320, 180)}}
(1.4354, ’PROCEDURAL’, ’RULE FIRED: encode word’)
(1.4479, ’PROCEDURAL’, ’RULE FIRED: retrieve word’)
(1.5993, ’retrieval’, ’RETRIEVED: word(cat=PRO, form=she,
          pred1=pred(arity=2, constant_name=_equals_),
          pred2=pred(arity=1, constant_name=_female_))’)
(1.6118, ’PROCEDURAL’, ’RULE FIRED: shift and project word (not N)’)
(1.6118, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=she,
          node_cat=PRO)’)
(1.6243, ’PROCEDURAL’, ’RULE FIRED: press spacebar’)
(1.6368, ’PROCEDURAL’, ’RULE FIRED: project: NP ==> PRO’)
(1.6368, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=PRO,
          lex_head=she, mother=S, mother_of_mother=ConjS,
          node_cat=NP)’)
(1.6368, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=x2,
          discourse_status=at_issue, dref=x2, drs=d2,
          embedding_level=1,
          pred1=pred(arity=1, constant_name=_female_))’)
(1.6368, ’unresolved_discourse’, ’CREATED A CHUNK: drs(arg1=x2,
          arg2=UNKNOWN, discourse_status=unresolved, drs=d2,
          embedding_level=1,
          pred1=pred(arity=2, constant_name=_equals_),
          pred2=pred(arity=1, constant_name=_female_))’)
(1.6493, ’PROCEDURAL’, ’RULE FIRED: move entity/individual dref peg to x3’)
(1.6618, ’PROCEDURAL’, ’RULE FIRED: attempting to resolve pronoun;\
          pronoun at embedding level 1’)
(1.7743, ’manual’, ’KEY PRESSED: SPACE’)
****Environment: {1: {’text’: ’left’, ’position’: (320, 180)}}
(1.8005, ’retrieval’, ’RETRIEVED: drs(arg1=x1, discourse_status=at_issue,
          dref=x1, drs=d1, embedding_level=0,
          pred1=pred(arity=1, constant_name=_woman_),
          pred2=pred(arity=1, constant_name=_female_))’)
(1.8130, ’PROCEDURAL’, ’RULE FIRED: resolution of PRO succeeded’)
(1.8130, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=x2, arg2=x1,
         discourse_status=presupposed, drs=d2, embedding_level=1,
         pred1=pred(arity=2, constant_name=_equals_),
         pred2=pred(arity=1, constant_name=_female_))’)
(1.8255, ’PROCEDURAL’, ’RULE FIRED: project and complete: S ==> NP VP’)
(1.8255, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=NP,
          daughter2=VP, node_cat=S)’)
(1.8380, ’PROCEDURAL’, ’RULE FIRED: encode word’)
(1.8505, ’PROCEDURAL’, ’RULE FIRED: retrieve word’)
(2.0028, ’retrieval’, ’RETRIEVED: word(cat=Vi, form=left,
          pred1=pred(arity=event_plus_1, constant_name=_left_))’)
(2.0153, ’PROCEDURAL’, ’RULE FIRED: shift and project word (not N)’)
(2.0153, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=left,
          node_cat=Vi)’)
(2.0403, ’PROCEDURAL’, ’RULE FIRED: project and complete: VP ==> Vi’)
(2.0403, ’imaginal’, ’CREATED A CHUNK: parse_state(daughter1=Vi,
          lex_head=left, mother=S, mother_of_mother=ConjS, node_cat=VP)’)
(2.0403, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=x2,
          discourse_status=at_issue, dref=e2, drs=d2,
          embedding_level=1, event_arg=e2,
          pred1=pred(arity=event_plus_1, constant_name=_left_))’)
(2.0528, ’PROCEDURAL’, ’RULE FIRED: move event dref peg to e3’)
(2.3403, ’PROCEDURAL’, ’RULE FIRED: finished: no visual input’)
```

```
                                                                        72
DRSs in declarative memory at the end of the simulation                 73
ordered by time of (re)activation:                                      74
0.5761   drs(arg1=x1, discourse_status=at_issue, dref=x1, drs=d1,       75
             embedding_level=0,                                         76
             pred1=pred(arity=1, constant_name=_woman_),                77
             pred2=pred(arity=1, constant_name=_female_))               78
1.2790   drs(arg1=x1, discourse_status=at_issue, dref=e1, drs=d1,       79
             embedding_level=0, event_arg=e1,                           80
             pred1=pred(arity=event_plus_1, constant_name=_smile_))     81
1.8130   drs(arg1=x2, discourse_status=at_issue, dref=x2, drs=d2,       82
             embedding_level=1,                                         83
             pred1=pred(arity=1, constant_name=_female_))               84
1.8130   drs(arg1=x2, arg2=UNKNOWN, discourse_status=unresolved,        85
             drs=d2, embedding_level=1,                                 86
             pred1=pred(arity=2, constant_name=_equals_),               87
             pred2=pred(arity=1, constant_name=_female_))               88
1.8130   drs(arg1=x1, discourse_status=at_issue, dref=x1, drs=d1,       89
             embedding_level=0,                                         90
             pred1=pred(arity=1, constant_name=_woman_),                91
             pred2=pred(arity=1, constant_name=_female_))               92
1.8255   drs(arg1=x2, arg2=x1, discourse_status=presupposed,            93
             drs=d2, embedding_level=1,                                 94
             pred1=pred(arity=2, constant_name=_equals_),               95
             pred2=pred(arity=1, constant_name=_female_))               96
2.3403   drs(arg1=x2, discourse_status=at_issue, dref=e2,               97
             drs=d2, embedding_level=1, event_arg=e2,                   98
             pred1=pred(arity=event_plus_1, constant_name=_left_))      99
```

The first point at which the temporal trace in (56) differs from the previous one in (45) is the word *and*. This word is displayed on the virtual screen at 1.06 s after the model starts reading the sentence in (55)—see lines 1–2 in (56).

The word *and* is encoded, its lexical information is retrieved and then the word is projected, i.e., a unary branching structure [$_{\text{Conj}}$ and] is created in the imaginal buffer (lines 7–8). The by-now familiar split into two cognitive sub-processes happens at this point: on one hand, a motor process to press the spacebar is started (line 9), on the other hand, we continue to process the word *and*. Specifically, the project-and-complete-*and* rule is fired (line 10), and a ternary branching structure [$_{\text{ConjS}}$ S Conj S] is created in the imaginal buffer (lines 11–13). Furthermore, since a new clause (the second conjunct) is about to start, we update the DRS dref peg from d_1 to d_2 (line 14).

The space bar is pressed and the next word, namely the pronoun *she*, appears on the virtual screen (lines 15–16) at time 1.4165 s. We go through the usual encode-retrieve-project-spacebar sequence of rules (lines 17–25), after which the project-NP rule for pronouns is fired (line 26). At that point, three chunks are created in the imaginal, `discourse_context` and `unresolved_discourse` buffers. The parse state in the imaginal buffer (lines 27–29) encodes the unary branching [$_{\text{NP}}$PRO] structure.

The DRS in the `discourse_context` buffer (lines 30–33) is the contribution made by the pronoun to at-issue content, and is represented in familiar DRT format as follows.

(57) At-issue DRS contributed by the pronoun *she*:

x_2
FEMALE(x_2)
[part of main DRS d_2]
[at-issue]
[embedding level: 1]

The DRS in the `unresolved_discourse` buffer (lines 34–38) is the contribution made by the pronoun to the unresolved presupposed content, and is represented in familiar DRT format as shown in (58) below. The double contribution of the pronoun to both at-issue and unresolved presupposed content follows the account of presupposition projection as anaphora resolution in van der Sandt (1992) (see also Kamp 2001a).

(58) Unresolved presupposed DRS contributed by the pronoun *she*:

x_2 = UNKNOWN
FEMALE(x_2)
[part of main DRS d_2]
[unresolved]
[embedding level: 1]

After the dref peg is updated to x_3 (line 39), we attempt to resolve the pronoun (lines 40–41), which means that a retrieval request is placed for a suitable antecedent. While we wait for the retrieval to complete, the motor module presses the space bar and reveals the final word *left* on the virtual screen (lines 42–43). At 1.8005 s, we successfully retrieve an antecedent for the pronoun (lines 44–47), namely the DRS contributed by the indefinite NP *A woman* in the first conjunct. In DRT format, the retrieved DRS is represented as follows.

(59) Potential antecedent DRS retrieved at time 1.8005 s:

x_1
WOMAN(x_1)
FEMALE(x_1)
[part of main DRS d_1]
[at-issue]
[embedding level: 0]

Since the antecedent matches in gender, we declare the pronoun successfully resolved (line 48) and add a resolved presupposition DRS to the `discourse_context` buffer at time 1.8130 s (lines 49–52). This DRS is represented in the usual DRT format as shown below.

(60) Presupposed DRS resulting from a successful pronoun resolution, added to the `discourse_context` buffer at time 1.8130 s:

$$\begin{array}{|c|}\hline \\ \hline x_2 = x_1 \\ \text{FEMALE}(x_2) \\ \\ \text{[part of main DRS } d_2\text{]} \\ \text{[presupposed]} \\ \text{[embedding level: 1]} \\ \hline \end{array}$$

After successfully resolving the pronoun, the incremental parsing process proceeds as expected. The project-S rule (line 53) is followed by:

- encoding, retrieving and projecting ('shifting') the intransitive verb *left* (lines 56–62),
- the project-VP rule and the syntactic structure and the DRS contributed by this rule (lines 63–69),
- the move-event-dref-peg rule (line 70), and finally,
- the `"finished: no visual input"` rule (line 71).

At the end of the simulation, we have seven DRSs in memory, listed on lines 75–99 in (56) above, together with their time stamps. The first two DRSs (lines 75–81) are part of the main DRS d_1 contributed by the first conjunct. The next two DRSs (lines 82–88) are contributed by the pronoun *she* in the second conjunct. The fifth DRS (lines 89–92) is just the first DRS (contributed by *A woman*) that is recalled to serve as the antecedent of the pronoun. The sixth DRS (lines 93–96) encodes the resolved presupposition of the pronoun, while the seventh DRS is the one contributed by the intransitive verb *left*.

Let us turn now to the example in (61) below, where the pronoun resolution fails because an antecedent with a non-matching gender is retrieved.

The only part of the temporal trace that differs from the one in (55) above is provided in (62) below. We see that the DRS contributed by the indefinite *A woman* is retrieved when we attempt to resolve the pronoun *he* (lines 5–8). We therefore declare the resolution failed because the antecedent and the pronoun do not match in gender.

As was the case in Chap. 8, the model predicts that recalling the mismatching antecedent should take slightly more time than recalling a match because the mismatching antecedent does not receive any spreading activation from the gender of the pronoun held in one of the buffers at the moment of recall.

At the end of the simulation in (62), we see that only six DRSs are stored in declarative memory (lines 14–34). These are basically the same DRSs as the ones stored after the simulation in (55), except for the seventh DRS that encoded the successfully resolved presupposition of the pronoun.

(61) A woman smiled and he left.

(62)
```
(1.6640, 'PROCEDURAL', 'RULE FIRED: attempting to resolve pronoun;\      1
          pronoun at embedding level 1')                                  2
(1.7765, 'manual', 'KEY PRESSED: SPACE')                                  3
****Environment: {1: {'text': 'left', 'position': (320, 180)}}            4
(1.8143, 'retrieval', 'RETRIEVED: drs(arg1=x1, discourse_status=at_issue,\ 5
          dref=x1, drs=d1, embedding_level=0, event_arg=None,\            6
          pred1=pred(arity=1, constant_name=_woman_),\                    7
          pred2=pred(arity=1, constant_name=_female_))')                  8
(1.8268, 'PROCEDURAL', 'RULE FIRED: resolution of PRO failed:\            9
          antecedent with non-matching gender')                           10
                                                                          11
DRSs in declarative memory at the end of the simulation                   12
ordered by time of (re)activation:                                        13
0.5765   drs(arg1=x1, arg2=None, discourse_status=at_issue, dref=x1,      14
             drs=d1, embedding_level=0, event_arg=None,                   15
             pred1=pred(arity=1, constant_name=_woman_),                  16
             pred2=pred(arity=1, constant_name=_female_))                 17
1.2807   drs(arg1=x1, discourse_status=at_issue, dref=e1, drs=d1,         18
             embedding_level=0, event_arg=e1,                             19
             pred1=pred(arity=event_plus_1, constant_name=_smile_))       20
1.8268   drs(arg1=x2, arg2=UNKNOWN, discourse_status=unresolved,          21
             drs=d2, embedding_level=1,                                   22
             pred1=pred(arity=2, constant_name=_equals_),                 23
             pred2=pred(arity=1, constant_name=_male_))                   24
1.8268   drs(arg1=x1, discourse_status=at_issue, dref=x1,                 25
             drs=d1, embedding_level=0,                                   26
             pred1=pred(arity=1, constant_name=_woman_),                  27
             pred2=pred(arity=1, constant_name=_female_))                 28
1.8393   drs(arg1=x2, discourse_status=at_issue, dref=x2,                 29
             drs=d2, embedding_level=1,                                   30
             pred1=pred(arity=1, constant_name=_male_))                   31
2.3542   drs(arg1=x2, discourse_status=at_issue, dref=e2,                 32
             drs=d2, embedding_level=1, event_arg=e2,                     33
             pred1=pred(arity=event_plus_1, constant_name=_left_))        34
```

9.3.5 Rules for Conditionals and Cataphora Resolution

We are now ready to discuss conditionals and cataphora resolution.

The project-and-complete rule for conditionals with a sentence-final *if*-clause needed to model Experiments 1 and 2 above follows the pattern of the project-and-complete rule for *and*. As shown in (63) below, we start a new sentence S for the conditional antecedent (line 14), we advance the DRS dref peg (line 13), and we mark the embedding level of the conditional antecedent as 1 (line 23), ensuring that pronouns in a matrix clause (with embedding level 0) won't be able to access drefs introduced by expressions in the conditional antecedent.

Finally, the rule creates a [$_{\text{CP}}$[$_{\text{C}}$if] S] structure in the imaginal buffer (lines 24–31) that will be Chomsky-adjoined to the previous (matrix clause) S by the next rule we will examine.

(63)
```
parser.productionstring(name="project and complete: sentence-final if",   1
string="""                                                                 2
    =g>                                                                    3
    isa                 parsing_goal                                       4
    task                parsing                                            5
    found               C                                                  6
    parsed_word         if                                                 7
    parsed_word         =w                                                 8
    embedding_level     0                                                  9
    ==>                                                                   10
    =g>                                                                   11
    isa                 parsing_goal                                      12
    task                move_drs_peg                                      13
    stack1              S                                                 14
    arg_stack1          None                                              15
    arg_stack2          None                                              16
    found               None                                              17
    parsed_word         None                                              18
    right_edge_stack1 S                                                   19
    right_edge_stack2 CP                                                  20
    right_edge_stack3 S                                                   21
    right_edge_stack4 none                                                22
    embedding_level     1                                                 23
    +imaginal>                                                            24
    isa                 parse_state                                       25
    daughter1           C                                                 26
    daughter2           S                                                 27
    node_cat            CP                                                28
    mother              S                                                 29
    mother_of_mother    None                                              30
    lex_head            =w                                                31
    ~discourse_context>                                                   32
""")                                                                      33
```

The project-and-complete rule for sentence final *if* sets the stage for a sequence of rules reanalyzing the previous (matrix) clause. We already mentioned this when we discussed Experiments 1 and 2 earlier in this chapter: a sentence-final *if*-clause triggers the reanalysis of the previous matrix clause because, until *if* is read, the incremental processor assumes the sentence is an unconditionalized assertion, so it has an embedding level of 0. When *if* is read, the previous clause has to be reanalyzed from a main clause/main assertion to a conditional consequent. Specifically, all the DRSs contributed as part of that clause should have their embedding level changed from 0 to 2.

The `"start if-triggered reanalysis"` rule in (64) below begins the process of recalling these DRSs for reanalysis. The task is updated to `if_reanalysis` (line 21), the CP structure created by the project-and-complete-*if* rule in (63) above is Chomsky-adjoined to the S node of the previous clause, and the structure is encoded in the imaginal buffer (lines 23–29). Most importantly, a retrieval request is placed for a DRS that has an embedding level of 0 and is indexed with the DRS dref peg of the previous sentence (lines 30–33).

(64)
```
parser.productionstring(name="start if-triggered reanalysis\
(for sentence-final if)", string="""
    =g>
    isa               parsing_goal
    task              parsing
    found             None
    parsed_word       None
    right_edge_stack2 =re2
    right_edge_stack3 =re3
    drs_peg           =drs_peg
    prev_drs_peg      =prev_drs_peg
    =imaginal>
    isa               parse_state
    daughter1         C
    node_cat          CP
    lex_head          if
    lex_head          =w
    ==>
    =g>
    isa               parsing_goal
    task              if_reanalysis
    right_edge_stack4 None
    +imaginal>
    isa               parse_state
    daughter1         S
    daughter2         =re2
    node_cat          =re3
    mother            None
    lex_head          =w
    +retrieval>
    isa               drs
    drs               =prev_drs_peg
    embedding_level   0
""")
```

Once the first DRS is recalled for reanalysis, we want to update its embedding level. To do this, we trigger one of two rules:

- `"if-triggered reanalysis (no event recalled)"`, provided in (65) below, or
- `"if-triggered reanalysis (event recalled)"`, provided in (66).

We need an 'event recalled' version of the rule because we want to capture the Maximize Presupposition effect we observed in Experiment 2; see the discussion of *Maximize Presupposition* in Sect. 9.1.2 above.

The default version of the rule, i.e., `"if-triggered reanalysis (no event recalled)"` in (65), is triggered when a DRS has been successfully retrieved and is available in the retrieval buffer (lines 8–21 in (65)). The rule triggers two actions. First, an identical DRS, except with an embedding level of 2, is placed in the `discourse_context` buffer (lines 25–37). Second, a new retrieval request for a DRS that has an embedding level of 0 and is indexed with the DRS dref peg of the previous sentence is placed (lines 40–43). Crucially, however, if a new DRS is to be retrieved, it should be different from the previous-sentence DRSs that have already been retrieved (lines 38–39)—we use here the FINST ('fingers of instantiation') feature we discussed in the previous chapter.

(65)
```
parser.productionstring(name="if-triggered reanalysis\      1
(no event recalled)", string="""                            2
    =g>                                                     3
    isa                 parsing_goal                        4
    task                if_reanalysis                       5
    drs_peg             =drs_peg                            6
    prev_drs_peg        =prev_drs_peg                       7
    =retrieval>                                             8
    isa                 drs                                 9
    drs                 =drs                                10
    discourse_status    =dstatus                            11
    embedding_level     0                                   12
    dref                =dref                               13
    pred1               =p1                                 14
    pred2               =p2                                 15
    dref                =dref                               16
    drs                 =drs                                17
    event_arg           =ea                                 18
    event_arg           None                                19
    arg1                =a1                                 20
    arg2                =a2                                 21
    ?retrieval>                                             22
    state               free                                23
    ==>                                                     24
    +discourse_context>                                     25
    isa                 drs                                 26
    discourse_status    =dstatus                            27
    drs                 =drs                                28
    embedding_level     2                                   29
    dref                =dref                               30
    pred1               =p1                                 31
    pred2               =p2                                 32
    dref                =dref                               33
    drs                 =drs                                34
    event_arg           =ea                                 35
    arg1                =a1                                 36
    arg2                =a2                                 37
    ?retrieval>                                             38
    recently_retrieved False                                39
    +retrieval>                                             40
    isa                 drs                                 41
    drs                 =prev_drs_peg                       42
    embedding_level     0                                   43
""")                                                        44
```

The `"if-triggered reanalysis (event recalled)"` rule in (66) below is very similar to the no-event-recalled rule in (65) above. The only difference is that when a DRS with an event dref is recalled, we keep track of its main predicate `=p1` in the goal buffer: this predicate is stored as the value of an `if_conseq_pred` feature (line 40), i.e., it is indexed as the predicate that was contributed by the conditional consequent.

(66)
```
parser.productionstring(name="if-triggered reanalysis (event recalled)",   1
string="""                                                  2
    =g>                                                     3
    isa                 parsing_goal                        4
    task                if_reanalysis                       5
    drs_peg             =drs_peg                            6
    prev_drs_peg        =prev_drs_peg                       7
    =retrieval>                                             8
    isa                 drs                                 9
    drs                 =drs                                10
    discourse_status    =dstatus                            11
    embedding_level     0                                   12
    dref                =dref                               13
    pred1               =p1                                 14
    pred2               =p2                                 15
```

```
        dref                =dref                                        16
        drs                 =drs                                         17
        event_arg           =ea                                          18
        event_arg           ~None                                        19
        arg1                =a1                                          20
        arg2                =a2                                          21
        ?retrieval>                                                      22
        state               free                                         23
        ==>                                                              24
        +discourse_context>                                              25
        isa                 drs                                          26
        discourse_status    =dstatus                                     27
        drs                 =drs                                         28
        embedding_level     2                                            29
        dref                =dref                                        30
        pred1               =p1                                          31
        pred2               =p2                                          32
        dref                =dref                                        33
        drs                 =drs                                         34
        event_arg           =ea                                          35
        arg1                =a1                                          36
        arg2                =a2                                          37
        =g>                                                              38
        isa                 parsing_goal                                 39
        if_conseq_pred      =p1                                          40
        ?retrieval>                                                      41
        recently_retrieved False                                         42
        +retrieval>                                                      43
        isa                 drs                                          44
        drs                 =prev_drs_peg                                45
        embedding_level     0                                            46
    """)                                                                 47
```

The two rules for *if*-triggered reanalysis (no-event-recalled and event-recalled) run repeatedly until all the DRSs contributed by the previous sentence are recalled and reanalyzed, i.e., their embedding level gets set to 2. Once there are no more DRSs to be recalled, the *if*-triggered reanalysis is complete and the `"stop if-triggered reanalysis"` rule in (67) below is triggered. This rule simply resets the task to `reading_word` (line 12) and flushes the retrieval, imaginal and `discourse_context` buffers (lines 15–17).

(67)
```
    parser.productionstring(name="stop if-triggered reanalysis", string="""   1
        =g>                                                              2
        isa                 parsing_goal                                 3
        task                if_reanalysis                                4
        ?retrieval>                                                      5
        state               error                                        6
        ?manual>                                                         7
        state               free                                         8
        ==>                                                              9
        =g>                                                              10
        isa                 parsing_goal                                 11
        task                reading_word                                 12
        found               None                                         13
        parsed_word         None                                         14
        ~retrieval>                                                      15
        ~imaginal>                                                       16
        ~discourse_context>                                              17
    """)                                                                 18
```

We are almost ready to parse the conditional & cataphora example in (7) above (*John won't eat it if a hamburger is overcooked*, Elbourne 2009), we only need to introduce two kinds of rules related to cataphoric search.

First, there are three rules that trigger a cataphoric search for an antecedent, depending on the embedding level of the potential antecedent. We only discuss the ‘embedding level 1’ rule, provided in (68) below. The other rules (linked to in Appendix 9.6.2) are similar.

(68)
```
parser.productionstring(name="attempting to resolve cataphoric pronoun;\    1
antecedent at embedding level 1", string="""                               2
    =g>                                                                    3
    isa                 parsing_goal                                       4
    task                ~reading_word                                      5
    task                ~move_dref_peg                                     6
    task                ~move_event_peg                                    7
    task                ~attempting_to_resolve_PRO                         8
    task                ~attempting_to_resolve_cataphoric_PRO              9
    task                ~stop_resolution_attempt_PRO                      10
    task                ~if_reanalysis                                    11
    found               None                                              12
    entity_cataphora    True                                              13
    ?retrieval>                                                           14
    state               free                                              15
    =discourse_context>                                                   16
    isa                 drs                                               17
    dref                ~None                                             18
    dref                =dref                                             19
    event_arg           None                                              20
    pred2               ~None                                             21
    pred2               =p2                                               22
    drs                 =drs                                              23
    embedding_level     1                                                 24
    ==>                                                                   25
    =g>                                                                   26
    isa                 parsing_goal                                      27
    task                stop_resolution_attempt_PRO                       28
    +unresolved_discourse>                                                29
    isa                 drs                                               30
    dref                =dref                                             31
    arg2                UNKNOWN                                           32
    pred2               =p2                                               33
    drs                 =drs                                              34
    embedding_level     1                                                 35
    discourse_status    unresolved                                        36
    +retrieval>                                                           37
    isa                 drs                                               38
    dref                None                                              39
    arg1                ~None                                             40
    arg2                UNKNOWN                                           41
    pred1               ~None                                             42
    drs                 ~=drs                                             43
    embedding_level     ~0                                                44
    discourse_status    unresolved                                        45
""", utility=5)                                                           46
```

Just as the anaphoric search rules, the cataphoric search rules are ‘elsewhere’ rules: they have a set of negative constraints for the current task (lines 5–11 in (68)), and a high utility (line 46). The rule is triggered if the `entity_cataphora` feature is set to `True` (line 13) and the `found` feature is set to `None` (line 12). Consequently, the rule cannot immediately follow a failed anaphoric search because such a search sets the `found` feature to `no_antecedent`. Most importantly, a cataphoric search is triggered only if a suitable antecedent is available in the `discourse_context` buffer (lines 16–24).

If these conditions are met, a cataphoric search is triggered. Cataphoric searches are mirror images of anaphoric searches. For cataphora, we have a potential antecedent in place, and place a retrieval request for an unresolved presupposition DRS that could be resolved by the antecedent. In contrast, for anaphora, we have an unresolved presupposition and we place a retrieval request for an antecedent that could resolve it.

The `"attempting to resolve cataphoric pronoun"` rule triggers three actions. The most important one is placing a retrieval request for an unresolved presupposition (lines 37–45 in (68)). We specify that no new drefs should be introduced in this unresolved DRS (line 39), that the `arg1` and `pred1` slots should be non-empty (lines 40 and 42), that the `arg2` slot should be set to `UNKNOWN` (line 41), that the DRS should be part of a main DRS that is different than the one that the potential antecedent belongs to (line 43), and that the embedding level of the unresolved presupposition should not be 0 (line 44) since the potential antecedent has an embedding level of 1.

The second action is to store information about the current potential antecedent in the `unresolved_discourse` buffer (lines 29–36). This is not strictly necessary since the information will be maintained in the `discourse_context` buffer, but we do it here just to show how specific buffers can be used to safeguard information that might otherwise be flushed by subsequent rules. And involving the `unresolved_discourse` buffer in the process of cataphoric search is a natural choice. We save the dref information (line 31), the gender predicate `=p2` (line 33), the DRS peg (line 34) and the embedding level of 1 (line 35).

The third and final action is to update the goal buffer (lines 26–28) so that the resolution attempt is stopped with this one retrieval request and does not enter a loop. We therefore update the `task` to `stop_resolution_attempt_PRO`.

The retrieval request, i.e., the cataphoric resolution attempt, can either succeed or fail. If the attempt fails, the same failure rules as for anaphoric attempts are triggered—see (53) and (54) above.

But if the cataphoric search succeeds, the `"resolution of cataphoric PRO succeeded"` rule in (69) below is triggered. This rule adds a DRS to the `discourse_context` buffer that resolves the retrieved unresolved presupposition to the currently available antecedent (lines 25–33). The DRS has a `presupposed` discourse status (line 33), and is otherwise identical to the DRS contributed by the `"resolution of PRO succeeded"` rule in (52) above.

(69)
```
parser.productionstring(name="resolution of cataphoric PRO succeeded",   1
string="""                                                                2
    =g>                                                                   3
    isa                 parsing_goal                                      4
    task                stop_resolution_attempt_PRO                       5
    =retrieval>                                                           6
    isa                 drs                                               7
    dref                None                                              8
    arg1                ~None                                             9
    arg1                =a1                                               10
    arg2                UNKNOWN                                           11
    pred1               =p1                                               12
    pred2               =p2                                               13
    =unresolved_discourse>                                                14
    isa                 drs                                               15
    dref                =dref                                             16
    pred2               =p2                                               17
    embedding_level     =el                                               18
    drs                 =drs                                              19
    ==>
    =g>
    isa               parsing_goal
    task              parsing
    entity_cataphora  None
    +discourse_context>
    isa               drs
    arg1              =a1
    arg2              =dref
    pred1             =p1
    pred2             =p2
    drs               =drs
    embedding_level   =el
    discourse_status  presupposed
    ~unresolved_discourse>
    ~retrieval>
""")
```

We can now see how the model parses the conditional + cataphora example in (70) below, repeated from (7) above. The temporal trace is provided in (71). We omit all the output that is not directly relevant to incremental semantic interpretation.

(70) John won’t eat it if a hamburger is overcooked. (Elbourne 2009)

(71)
```
****Environment: {1: {'text': 'John', 'position': (320, 180)}}
(0.2075, 'PROCEDURAL', 'RULE FIRED: project: NP ==> ProperN')
(0.2075, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x1,\
          discourse_status=at_issue, dref=x1, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_john_),\
          pred2=pred(arity=1, constant_name=_male_))')
****Environment: {1: {'text': 'wont', 'position': (320, 180)}}
(0.5645, 'PROCEDURAL', 'RULE FIRED: project and complete:\
          VP ==> VauxNeg VP')
(0.5645, 'discourse_context', 'CREATED A CHUNK: drs(drs=d1,\
          discourse_status=at_issue, embedding_level=0, pred1=NOT)')
****Environment: {1: {'text': 'eat', 'position': (320, 180)}}
(0.9244, 'PROCEDURAL', 'RULE FIRED: project and complete: VP ==> Vt NP')
(0.9244, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x1, event_arg=e1,\
          discourse_status=at_issue, dref=e1, drs=d1, embedding_level=0,\
          pred1=pred(arity=event_plus_2, constant_name=_eat_))')
****Environment: {1: {'text': 'it', 'position': (320, 180)}}
(1.2893, 'PROCEDURAL', 'RULE FIRED: project and complete: NP ==> PRO')
(1.2893, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x2,\
          discourse_status=at_issue, dref=x2, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_nonhuman_))')
(1.2893, 'unresolved_discourse', 'CREATED A CHUNK: drs(embedding_level=0,\
          arg1=x2, arg2=UNKNOWN, discourse_status=unresolved, drs=d1,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_nonhuman_))')
(1.3143, 'PROCEDURAL', 'RULE FIRED: attempting to resolve pronoun;\
          pronoun at embedding level 0')
****Environment: {1: {'text': 'if', 'position': (320, 180)}}
(1.5168, 'retrieval', 'RETRIEVED: None')
(1.5293, 'PROCEDURAL', 'RULE FIRED: resolution of PRO failed:\
          no antecedent')
(1.7435, 'PROCEDURAL', 'RULE FIRED: project and complete:\
          sentence-final if')
(1.7560, 'PROCEDURAL', 'RULE FIRED:\
          move DRS/propositional dref peg to d2')
(1.7685, 'PROCEDURAL', 'RULE FIRED: start if-triggered reanalysis\
          (for sentence-final if)')
****Environment: {1: {'text': 'a', 'position': (320, 180)}}
(1.9053, 'retrieval', 'RETRIEVED: drs(arg1=x2, discourse_status=at_issue,\
          dref=x2, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_nonhuman_))')
(1.9178, 'PROCEDURAL', 'RULE FIRED: if-triggered reanalysis\
          (no event recalled)')
(1.9178, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x2,\
          discourse_status=at_issue, dref=x2, drs=d1, embedding_level=2,\
          pred1=pred(arity=1, constant_name=_nonhuman_))')
(2.0643, 'retrieval', 'RETRIEVED: drs(arg1=x2, arg2=UNKNOWN,\
          discourse_status=unresolved, drs=d1, embedding_level=0,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_nonhuman_))')
(2.0768, 'PROCEDURAL', 'RULE FIRED: if-triggered reanalysis\
          (no event recalled)')
(2.0768, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x2, arg2=UNKNOWN,\
          discourse_status=unresolved, drs=d1, embedding_level=2,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_nonhuman_))')
(2.2259, 'retrieval', 'RETRIEVED: drs(arg1=x1, arg2=x2, dref=e1, drs=d1,\
          discourse_status=at_issue, embedding_level=0, event_arg=e1,\
          pred1=pred(arity=event_plus_2, constant_name=_eat_))')
(2.2384, 'PROCEDURAL', 'RULE FIRED: if-triggered reanalysis\
          (event recalled)')
(2.2384, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x1, arg2=x2,\
          discourse_status=at_issue, dref=e1, drs=d1, embedding_level=2,\
          event_arg=e1,\
          pred1=pred(arity=event_plus_2, constant_name=_eat_))')
(2.3894, 'retrieval', 'RETRIEVED: drs(discourse_status=at_issue, drs=d1,\
          embedding_level=0, pred1=NOT)')
(2.4019, 'PROCEDURAL', 'RULE FIRED: if-triggered reanalysis\
          (no event recalled)')
(2.4019, 'discourse_context', 'CREATED A CHUNK: drs(drs=d1,\
          discourse_status=at_issue, embedding_level=2, pred1=NOT)')
(2.5548, 'retrieval', 'RETRIEVED: drs(arg1=x1, discourse_status=at_issue,\
          dref=x1, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_john_),\
          pred2=pred(arity=1, constant_name=_male_))')
(2.5673, 'PROCEDURAL', 'RULE FIRED: if-triggered reanalysis\
          (no event recalled)')
(2.5673, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x1,\
          discourse_status=at_issue, dref=x1, drs=d1, embedding_level=2,\
          pred1=pred(arity=1, constant_name=_john_),\
          pred2=pred(arity=1, constant_name=_male_))')
(2.7698, 'retrieval', 'RETRIEVED: None')
(2.7823, 'PROCEDURAL', 'RULE FIRED: stop if-triggered reanalysis')
(2.9987, 'PROCEDURAL', 'RULE FIRED: project: NP ==> Det N')
(2.9987, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x3,\
          discourse_status=at_issue, dref=x3, drs=d2, embedding_level=1)')
****Environment: {1: {'text': 'hamburger', 'position': (320, 180)}}
(3.3592, 'PROCEDURAL', 'RULE FIRED: project and complete: N')
(3.3717, 'PROCEDURAL', 'RULE FIRED: attempting to resolve cataphoric\
          pronoun; antecedent at embedding level 1')
(3.3717, 'unresolved_discourse', 'CREATED A CHUNK: drs(arg2=UNKNOWN,\
          discourse_status=unresolved, dref=x3, drs=d2, embedding_level=1,\
          pred2=pred(arity=1, constant_name=_nonhuman_))')
****Environment: {1: {'text': 'is', 'position': (320, 180)}}
(3.5159, 'retrieval', 'RETRIEVED: drs(arg1=x2, arg2=UNKNOWN,\
          discourse_status=unresolved, drs=d1, embedding_level=2,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_nonhuman_))')
(3.5284, 'PROCEDURAL', 'RULE FIRED: resolution of cataphoric PRO\
          succeeded')
(3.5284, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x2, arg2=x3,\
          discourse_status=presupposed, drs=d2, embedding_level=1,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_nonhuman_))')
****Environment: {1: {'text': 'overcooked', 'position': (320, 180)}}
(4.1232, 'PROCEDURAL', 'RULE FIRED: project and complete: AP ==> A')
(4.1232, 'discourse_context', 'CREATED A CHUNK: drs(arg1=x3,\
          discourse_status=at_issue, drs=d2, embedding_level=1,\
          pred1=pred(arity=1, constant_name=_overcook_))')
(4.4232, 'PROCEDURAL', 'RULE FIRED: finished: no visual input')
```

The first word, namely the proper name *John*, contributes the first DRS to the `discourse_context` buffer—see lines 3–6 in (71) above. This DRS can be represented in the familiar DRT format as shown below.

(72) DRS contributed by the proper name *John*:

x_1
JOHN(x_1)
MALE(x_1)
[part of main DRS d_1]
[at-issue]
[embedding level: 0]

Providing an analysis of negation is outside the scope of this model, so the negated auxiliary *won't* contributes a 'placeholder' DRS with no drefs or arguments and a predicate `NOT` that simply marks that the DRS was contributed by a form of sentential negation (lines 10–11).

The transitive verb *eat* contributes the DRS below (see lines 14–16 in (71)):

(73) DRS contributed by the transitive verb *eat*:

e_1
EAT(e_1, x_1, _)
[part of main DRS d_1]
[at-issue]
[embedding level: 0]

The pronoun *it* further specifies the DRS introduced by the verb *eat* (not shown in the temporal trace) and introduces its own DRSs (line 19–25):

(74) DRS contributed by the transitive verb *eat* and updated by the pronoun *it*:

e_1
EAT(e_1, x_1, x_2)
[part of main DRS d_1]
[at-issue]
[embedding level: 0]

(75) At-issue DRS contributed by the pronoun *it*:

x_2
NONHUMAN(x_2)
[part of main DRS d_1]
[at-issue]
[embedding level: 0]

(76) Unresolved-presupposition DRS contributed by the pronoun *it*:

x_2 = UNKNOWN
NONHUMAN(x_2)
[part of main DRS d_1]
[unresolved]
[embedding level: 0]

An attempt to resolve the pronoun *it* (lines 26–27) ends in failure (lines 29–31), since there is no suitable antecedent for it. The parsing process then moves on to the complementizer *if* (lines 32–33), which triggers the reanalysis of the previous clause from a main assertion to a conditional consequent. This means recalling all the four DRSs contributed by the previous clause and creating four new DRSs with the same content, except that the embedding level is set to 2 (conditional consequent) instead of 0 (main assertion).

The first DRS that gets recalled as part of the *if*-triggered reanalysis is the at-issue DRS in (75) above contributed by the pronoun *it* (lines 39–41). The reanalysis of this DRS (which sets the embedding level to 2) creates a new DRS in the `discourse_context` buffer (lines 44–46), shown below.

(77) The first DRS contributed by *if*-triggered reanalysis:

x_2
NONHUMAN(x_2)
[part of main DRS d_1]
[at-issue]
[embedding level: 2]

The second DRS retrieved as part of *if*-reanalysis is the unresolved-presupposition DRS contributed by the pronoun (lines 47–50). We see that recency is the most important factor for DRS activation in the *if*-reanalysis process. A new, reanalyzed unresolved-presupposition DRS is added to the `discourse_context` buffer (lines 53–56), which can be represented as follows.

(78) The second DRS contributed by *if*-triggered reanalysis:

x_2 = UNKNOWN
NONHUMAN(x_2)
[part of main DRS d_1]
[unresolved]
[embedding level: 2]

The third DRS retrieved during the *if*-reanalysis process is the one contributed by the transitive verb *eat* (lines 57–59). Because this DRS introduces an event, it triggers the 'event recalled' version of the `"if-triggered reanalysis"` rule, which does do things. On one hand, it adds the verbal predicate to the goal buffer as the value of the `if_conseq_pred` feature. On the other hand, it resets the embedding level of the recalled DRS to 2 and adds the modified DRS to the `discourse_context` buffer (lines 62–65). This DRS can be represented in DRT format as follows.

(79) DRS contributed by the transitive verb *eat* and updated by the pronoun *it*:

e_1
EAT(e_1, x_1, x_2)
[part of main DRS d_1]
[at-issue]
[embedding level: 2]

After this, the 'dummy' negative DRS contributed by the negated auxiliary *won't* is retrieved (lines 66–67) and reanalyzed (lines 68–71). Finally, the DRS contributed by the proper name *John* is retrieved (lines 72–75) and reanalyzed (lines 76–81). The new DRS, which is the final one contributed by the *if*-reanalysis process, can be represented as shown below.

(80) DRS contributed by the proper name *John*:

x_1
JOHN(x_1)
MALE(x_1)
[part of main DRS d_1]
[at-issue]
[embedding level: 2]

At this point, the reanalysis process concludes and we continue with the incremental parsing process—specifically, the interpretation of the indefinite article *a* (in …*a hamburger is* …). We should note here that this process is cognitively unrealistic: it predicts that we need about 1.2 s to process sentence-final *if*, because the process of retrieving all these DRSs and reanalyzing them is very time consuming. We implement it here just to show that this type of process can be modeled in our framework, and leave a more appropriate model for a future occasion.[5]

The next DRS is contributed by the indefinite NP *a hamburger* in the *if*-clause (lines 84–88), and can be represented as follows.

(81) DRS contributed by the indefinite *a hamburger*:

$$\boxed{\begin{array}{c} x_3 \\ \hline \text{HAMBURGER}(x_3) \\ \\ \text{[part of main DRS } d_2\text{]} \\ \text{[at-issue]} \\ \text{[embedding level: 1]} \end{array}}$$

The presence of this new DRS in the `discourse_context` buffer and the fact that the `entity_cataphora` feature in the goal buffer is turned on after the unsuccessful resolution of the pronoun *it* trigger an attempt to resolve the cataphoric pronoun (lines 89–90). The rule places the relevant information about the potential antecedent *a hamburger* in the `unresolved_discourse` buffer (lines 91–93) and places a retrieval request for an unresolved presupposition contributed by a pronoun. This request is successfully completed, and as a result, the unresolved DRS contributed by *it* is available in the retrieval buffer (lines 95–98).

The resolution of the cataphoric pronoun is declared a success (line 99–100) and the resolved presupposition, provided below for convenience, is added to the `discourse_context` buffer (lines 101–104).

[5]One improvement would be to spread the reanalysis process over several words in the *if*-clause rather than eagerly complete it.

Another improvement would be to explicitly model a form of clause-final wrap-up (maybe supplemented by clause-medial wrap-ups) that integrates/merges some of the DRSs that are indexed with the same DRS dref. This would ensure that there are fewer DRSs to recall during the *if*-reanalysis process.

Yet another improvement would be to explicitly model the main DRS like we did in the previous chapter (Chap. 8), in which case we might just need to recall only one DRS—the main one—and reanalyze its embedding level only once.

A related improvement would be to maintain the distributed representation of main DRSs we use in this chapter, but somehow 'centralize' their embedding level and DRS dref peg encodings in a separate DRS that would act as a 'mother-node' DRS to all the sub-DRSs. That is, we would build tree-like structures for discourse contexts; see Vermeulen (1994) and its Incremental Dynamic Predicate Logic extension in Brasoveanu and Dotlačil (2015a, b) for one way to model discourse contexts as tree-like update histories.

(82) Presupposed DRS resolving the pronoun *it* to the indefinite *a hamburger*:

$$\begin{array}{|c|}\hline \\ \hline x_2 = x_3 \\ \text{NONHUMAN}(x_2) \\ \\ \text{[part of main DRS } d_2\text{]} \\ \text{[presupposed]} \\ \text{[embedding level: 1]} \\ \hline \end{array}$$

The copula *is* and the adjectival participle *overcooked* contribute a final DRS (lines 107–109) to the `discourse_context` buffer:

(83) Final at-issue DRS contributed by *overcooked*:

$$\begin{array}{|c|}\hline \\ \hline \text{OVERCOOK}(x_3) \\ \\ \text{[part of main DRS } d_2\text{]} \\ \text{[at-issue]} \\ \text{[embedding level: 1]} \\ \hline \end{array}$$

9.4 Modeling the Interaction of Conditionals and Cataphoric Presuppositions

We are now ready to move to the somewhat more complex case of event anaphora/cataphora associated with the adverb *again*. We first introduce the rules for the syntax and semantics of *again* and for the process of presupposition resolution for event anaphora (Sect. 9.4.1). We then discuss one way of capturing the 'maximize presupposition' effect we saw in Experiment 2 above (Sect. 9.4.2). Finally, we discuss the results of fitting the model to part of the Experiment 2 data (Sect. 9.4.3).

9.4.1 Rules for 'Again' and Presupposition Resolution

The resolution of event anaphora/cataphora contributed by the adverb *again* follows the same pattern as the resolution of entity anaphora/cataphora contributed by pronouns. The main difference is in the syntax of *again*: *again* is an adverb/adjunct, and given the optionality of adjuncts, they basically need to be parsed bottom-up.[6] That is, the syntactic attachment of adjuncts requires a form of syntactic reanalysis.

Consider again the conditional + event cataphora example in (16d) above, repeated in (84) below:

(84) Jeffrey will argue with Danielle again if he argued with her in the courtyard last night.

The matrix clause ends with the anaphoric adverb *again*. To parse it, we need to attach it to the VP *will argue with Danielle* that has already been completely parsed and closed by the time we encounter *again*. To build the appropriate syntactic structure, we would therefore need to recall both the VP node and the higher S node so that the adverb *again* can be attached intermediately between them. For simplicity, we will simply recall the higher S node and add the adverb *again* as a third daughter, as shown by the two rules in (85) and (86) below.

(85)
```
parser.productionstring(name="recall S for adjoining adv. AGAIN",
string="""
    =g>
    isa                 parsing_goal
    task                parsing
    stack1              None
    found               Adv
    parsed_word         =w
    embedding_level     =el
    =retrieval>
    isa                 word
    cat                 Adv
    ==>
    =g>
    isa                 parsing_goal
    task                recall_S
    +retrieval>
    isa                 parse_state
    node_cat            S
    daughter1           NP
    daughter2           VP
    ~imaginal>
""")
```

(86)
```
parser.productionstring(name="build S adjunction and\
recall event for AGAIN", string="""
    =g>
    isa                 parsing_goal
    task                recall_S
    stack1              None
    found               Adv
    drs_peg             =drs_peg
    =retrieval>
    isa                 parse_state
    node_cat            S
    daughter1           NP
    daughter2           VP
    ==>
    =g>
    isa                 parsing_goal
    task                recall_event
    +imaginal>
    isa                 parse_state
```

[6]Left-corner, bottom-up and top-down parsing can be all unified under generalized left-corner parsing (Demers 1977). See Hale (2014) for a recent discussion.

```
        node_cat            S
        daughter1           NP
        daughter2           VP
        daughter3           Adv
        +retrieval>
        isa                 drs
        dref                ~None
        event_arg           ~None
        drs                 =drs_peg
    """)
```

In addition to reanalyzing the S node, rule (86) places another retrieval request for the event contributed by the verb modified by the adverb *again*. This event is needed for the semantics of *again*. The fact that we need two separate retrieval requests, one on the syntax side for the S node and one on the semantics side for the event, is an artifact of our setup for syntactic and semantic parsing.

For expository simplicity, the semantic information constructed during incremental interpretation is assembled in chunks and buffers that are separate from the chunks and buffers where syntactic information is constructed. This enabled us to import the left-corner syntax parser we introduced in Chap. 4 basically as-is, and we were able to focus on the semantic aspects of interpretation in this and the previous chapter (Chap. 8 and this chapter) without worrying about a tighter integration of the syntactic and semantic aspects of parsing.

As we investigate more complex structures and their interpretation, these simplifying assumptions are likely to come into focus and require revision. It is possible that the structures constructed during the incremental interpretation process integrate phonetics/phonology, syntax and semantics in a tighter way, for example, along the lines of the linguistic representations countenanced by HPSG or CG.

Once the event contributed by the modified verb is available in the retrieval buffer, we are able to encode the unresolved presupposition contributed by *again*. This is accomplished by the rule in (87) below. The unresolved presupposition DRS contributed by *again* on lines 22–30 is largely parallel to the unresolved presupposition DRS contributed by pronouns. There are three main differences: (i) `arg1` (not `arg2`) is marked as `UNKNOWN` for *again* (line 24), (ii) the first predicate is (temporally) `PRECEDES` (not `EQUALS`; line 26), and (iii) the second predicate is the verbal predicate contributed by the modified verb (not gender; line 27).

(87)
```
parser.productionstring(name="encode unresolved event presupposition\      1
for AGAIN", string="""                                                      2
    =g>                                                                     3
    isa                 parsing_goal                                        4
    task                recall_event                                        5
    stack1              None                                                6
    found               Adv                                                 7
    drs_peg             =drs                                                8
    =retrieval>                                                             9
    isa                 drs                                                 10
    event_arg           =ea                                                 11
    pred1               =p1                                                 12
    drs                 =drs                                                13
    embedding_level     =el                                                 14
    discourse_status    at_issue                                            15
    ==>                                                                     16
    =g>                                                                     17
    isa                 parsing_goal                                        18
    task                parsing                                             19
    found              None
    parsed_word        None
    +unresolved_discourse>
    isa                drs
    arg1               UNKNOWN
    arg2               =ea
    pred1              PRECEDES
    pred2              =p1
    drs                =drs
    embedding_level    =el
    discourse_status   unresolved
    ~retrieval>
    ~imaginal>
""")
```

Once the *again* presupposition is encoded, we can attempt to resolve it. There are three different rules for the three different embedding levels of the presupposition. We provide only the rule for embedding level 1 in (88) below, for ease of comparison with the pronoun rule. The rule has the same structure, the main difference is that we now require the potential antecedent to have a non-empty event argument (lines 31–32).

(88)
```
 1  parser.productionstring(name="attempting to resolve event presupposition;\
 2  presupposition at embedding level 1", string="""
 3      =g>
 4      isa                parsing_goal
 5      task               ~reading_word
 6      task               ~move_dref_peg
 7      task               ~move_event_peg
 8      task               ~attempting_to_resolve_AGAIN
 9      task               ~attempting_to_resolve_cataphoric_AGAIN
10      task               ~stop_resolution_attempt_AGAIN
11      task               ~move_event_peg_and_wait_for_retrieval
12      task               ~if_reanalysis
13      found              None
14      ?retrieval>
15      state              free
16      =unresolved_discourse>
17      isa                drs
18      arg1               UNKNOWN
19      arg2               =ea
20      pred2              =p2
21      drs                =drs
22      embedding_level    1
23      discourse_status   unresolved
24      ==>
25      =g>
26      isa                parsing_goal
27      task               stop_resolution_attempt_AGAIN
28      +retrieval>
29      isa                drs
30      dref               ~None
31      event_arg          ~None
32      event_arg          ~=ea
33      drs                ~=drs
34      embedding_level    ~2
35      discourse_status   at_issue
36  """, utility=5)
```

If the resolution succeeds, we add the resolved presupposition to the `discourse_context` buffer with the `"resolution of AGAIN succeeded"` rule in (89) below.

(89)
```
parser.productionstring(name="resolution of AGAIN succeeded", string="""
    =g>
    isa                 parsing_goal
    task                stop_resolution_attempt_AGAIN
    embedding_level     =el
    =retrieval>
    isa                 drs
    dref                ~None
    dref                =ea
    event_arg           =ea
    pred1               =p2
    =unresolved_discourse>
    isa                 drs
    arg1                UNKNOWN
    arg2                =ea2
    pred1               =p1
    pred2               =p2
    drs                 =drs
    discourse_status    unresolved
    ==>
    =g>
    isa                 parsing_goal
    task                parsing
    event_cataphora     None
    +discourse_context>
    isa                 drs
    arg1                =ea
    arg2                =ea2
    pred1               =p1
    pred2               =p2
    drs                 =drs
    embedding_level     =el
    discourse_status    presupposed
    ~retrieval>
    ~unresolved_discourse>
""")
```

Just as for pronouns, the resolution of the *again* presupposition can fail, either (i) because no suitable antecedent is retrieved or (ii) because an antecedent is retrieved, but the retrieved verbal predicate is different from the verbal predicate of the unresolved *again* presupposition. These two cases are handled by the rules in (90) and (91) below, respectively.

In both cases, we turn the `event_cataphora` feature on when the retrieval of a suitable antecedent event fails. That is, we do not simply mark the presupposition resolution process as a failure and end it. Instead, we assume that the *again* presupposition is cataphoric.

(90)
```
parser.productionstring(name="resolution of AGAIN failed: no antecedent",
string="""
    =g>
    isa                 parsing_goal
    task                stop_resolution_attempt_AGAIN
    ?retrieval>
    state               error
    ?unresolved_discourse>
    buffer              full
    =unresolved_discourse>
    isa                 drs
    arg1                UNKNOWN
    arg2                =ea2
    pred1               =p1
    pred2               =p2
    drs                 =drs
    embedding_level     =el
    discourse_status    unresolved
    ==>
    =g>
    isa                 parsing_goal
    task                parsing
    event_cataphora     True
    found               no_antecedent
    ~unresolved_discourse>
    ~retrieval>
""")
```

(91)
```
parser.productionstring(name="resolution of AGAIN failed:\
antecedent with non-matching verbal predicate", string="""
    =g>
    isa                 parsing_goal
    task                stop_resolution_attempt_AGAIN
    ?unresolved_discourse>
    buffer              full
    =retrieval>
    isa                 drs
    pred1               =p2
    =unresolved_discourse>
    isa                 drs
    dref                None
    pred2               ~=p2
    discourse_status    unresolved
    ==>
    =g>
    isa                 parsing_goal
    task                parsing
    event_cataphora     True
    found               no_antecedent
    ~retrieval>
    ~unresolved_discourse>
""")
```

Once the event presupposition contributed by *again* is marked as cataphoric and a suitable antecedent is available in the discourse_context buffer, we start a cataphoric search, just as we did for pronouns. Once again, there are three rules for cataphoric search depending on the embedding level of the potential event antecedent. We provide only the embedding level 1 rule in (92) below.

(92)
```
parser.productionstring(name="attempting to resolve cataphoric event\
presupposition; antecedent at embedding level 1", string="""
    =g>
    isa                 parsing_goal
    task                ~reading_word
    task                ~move_dref_peg
    task                ~move_event_peg
    task                ~attempting_to_resolve_AGAIN
    task                ~attempting_to_resolve_cataphoric_AGAIN
    task                ~stop_resolution_attempt_AGAIN
    task                ~if_reanalysis
    found               None
    event_cataphora     True
    =discourse_context>
    isa                 drs
    dref                ~None
    dref                =ea
    arg1                ~None
    event_arg           =ea
    pred1               ~None
    pred1               ~None
    pred1               =p1
    pred2               None
    drs                 =drs
    embedding_level     1
    ?retrieval>
    state             free
    ==>
    =g>
    isa               parsing_goal
    task              stop_resolution_attempt_AGAIN
    +unresolved_discourse>
    isa               drs
    dref              =ea
    arg1              UNKNOWN
    pred2             =p1
    drs               =drs
    embedding_level   1
    discourse_status  unresolved
    +retrieval>
    isa               drs
    dref              None
    arg1              UNKNOWN
    arg2              ~None
    pred1             ~None
    drs               ~=drs
    embedding_level   ~0
    discourse_status  unresolved
""", utility=5)
```

If the resolution of cataphoric *again* succeeds, we add the resolved presupposition to the `discourse_context` buffer with the rule in (93) below.

(93)
```
parser.productionstring(name="resolution of cataphoric AGAIN succeeded",
string="""
    =g>
    isa               parsing_goal
    task              stop_resolution_attempt_AGAIN
    =retrieval>
    isa               drs
    dref              None
    arg1              UNKNOWN
    arg2              ~None
    arg2              =a2
    pred1             =p1
    pred2             =p2
    =unresolved_discourse>
    isa               drs
    dref              =dref
    pred2             =p2
    embedding_level   =el
    drs               =drs
    ==>
    =g>
    isa               parsing_goal
    task              parsing
    event_cataphora  None
    +discourse_context>
    isa               drs
    arg1              =dref
    arg2              =a2
    pred1             =p1
    pred2             =p2
    drs               =drs
    embedding_level   =el
    discourse_status  presupposed
    ~unresolved_discourse>
    ~retrieval>
""")
```

If the resolution of cataphoric *again* fails because the antecedent does not match the verbal predicate, the rule in (94) below is triggered. If there is no suitable antecedent, the general rule in (90) above, which applies to both anaphoric and cataphoric searches, is triggered.

(94)
```
parser.productionstring(name="resolution of cataphoric AGAIN failed:\
antecedent with non-matching verbal predicate", string="""
    =g>
    isa                 parsing_goal
    task                stop_resolution_attempt_AGAIN
    ?unresolved_discourse>
    buffer              full
    =retrieval>
    isa                 drs
    pred2               =p2
    =unresolved_discourse>
    isa                 drs
    dref                ~None
    pred2               ~=p2
    discourse_status    unresolved
    ==>
    =g>
    isa                 parsing_goal
    task                parsing
    event_cataphora     True
    found               no_antecedent
    ~retrieval>
    ~unresolved_discourse>
""")
```

We are now ready to see how all these rules get deployed during the incremental interpretation of the conditional & event cataphora example in (95) below, repeated from above. We stop the parsing immediately after the preposition *with* in the *if*-clause since this is when event cataphora is resolved.

(95) Jeffrey will argue with Danielle again if he argued with [her in the courtyard last night].

(96)
```
****Environment: {1: {'text': 'Jeffrey', 'position': (320, 180)}}
(0.2012, 'PROCEDURAL', 'RULE FIRED: project: NP ==> ProperN')
(0.2012, 'discourse_context', 'CREATED A CHUNK: drs(dref=x1, arg1=x1,\
          discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_jeffrey_),\
          pred2=pred(arity=1, constant_name=_male_))')
****Environment: {1: {'text': 'will', 'position': (320, 180)}}
****Environment: {1: {'text': 'argue', 'position': (320, 180)}}
(0.907, 'PROCEDURAL', 'RULE FIRED: project and complete: VP ==> Vt PP\
         (no if_conseq_pred present)')
(0.907, 'discourse_context', 'CREATED A CHUNK: drs(dref=e1, event_arg=e1,\
         arg1=x1, discourse_status=at_issue, drs=d1, embedding_level=0,\
         pred1=pred(arity=event_plus_2, constant_name=_argu_))')
****Environment: {1: {'text': 'with', 'position': (320, 180)}}
(1.2633, 'PROCEDURAL', 'RULE FIRED: project and complete: PP ==> P NP')
****Environment: {1: {'text': 'Danielle', 'position': (320, 180)}}
(1.6189, 'PROCEDURAL', 'RULE FIRED: project and complete: NP ==> ProperN')
(1.6189, 'discourse_context', 'CREATED A CHUNK: drs(dref=x2, arg1=x2,\
          discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_danielle_),\
          pred2=pred(arity=1, constant_name=_female_))')
****Environment: {1: {'text': 'again', 'position': (320, 180)}}
(1.9729, 'PROCEDURAL', 'RULE FIRED: recall S for adjoining adv. AGAIN')
****Environment: {1: {'text': 'if', 'position': (320, 180)}}
(2.1243, 'retrieval', 'RETRIEVED: parse_state(daughter1=NP, daughter2=VP,\
          node_cat=S)')
(2.1353, 'PROCEDURAL', 'RULE FIRED: build S adjunction and\
          recall event for AGAIN')
(2.2828, 'retrieval', 'RETRIEVED: drs(dref=e1, event_arg=e1, arg1=x1,\
          arg2=x2, discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=event_plus_2, constant_name=_argu_),\
          pred2=pred(arity=2, constant_name=_with_))’)
(2.2938, ’PROCEDURAL’, ’RULE FIRED: encode unresolved event presupposition\
          for AGAIN’)
(2.2938, ’unresolved_discourse’, ’CREATED A CHUNK: drs(arg1=UNKNOWN,\
          arg2=e1, discourse_status=unresolved, drs=d1, embedding_level=0,\
          pred1=pred(arity=2, constant_name=_precedes_),\
          pred2=pred(arity=event_plus_2, constant_name=_argu_))’)
(2.3048, ’PROCEDURAL’, ’RULE FIRED: attempting to resolve event\
          presupposition; presupposition at embedding level 0’)
(2.4880, ’retrieval’, ’RETRIEVED: None’)
(2.4990, ’PROCEDURAL’, ’RULE FIRED: resolution of AGAIN failed:\
          no antecedent’)
(2.7075, ’PROCEDURAL’, ’RULE FIRED: project and complete:\
          sentence-final if’)
(2.7295, ’PROCEDURAL’, ’RULE FIRED: start if-triggered reanalysis\
          (for sentence-final if)’)
****Environment: {1: {’text’: ’he’, ’position’: (320, 180)}}
(2.8768, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x2, arg1=x2,\
          discourse_status=at_issue, drs=d1, embedding_level=2,\
          pred1=pred(arity=1, constant_name=_danielle_),\
          pred2=pred(arity=1, constant_name=_female_))’)
(3.0342, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=UNKNOWN, arg2=e1,\
          discourse_status=unresolved, drs=d1, embedding_level=2,\
          pred1=pred(arity=2, constant_name=_precedes_),\
          pred2=pred(arity=event_plus_2, constant_name=_argu_))’)
(3.1991, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x1, arg1=x1,\
          discourse_status=at_issue, drs=d1, embedding_level=2,\
          pred1=pred(arity=1, constant_name=_jeffrey_),\
          pred2=pred(arity=1, constant_name=_male_))’)
(3.3556, ’discourse_context’, ’CREATED A CHUNK: drs(dref=e1,\
          event_arg=e1, arg1=x1, arg2=x2,\
          discourse_status=at_issue, drs=d1, embedding_level=2,\
          pred1=pred(arity=event_plus_2, constant_name=_argu_),\
          pred2=pred(arity=2, constant_name=_with_))’)
(3.5498, ’PROCEDURAL’, ’RULE FIRED: stop if-triggered reanalysis’)
(3.7597, ’PROCEDURAL’, ’RULE FIRED: project: NP ==> PRO’)
(3.7597, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x3, arg1=x3,\
          discourse_status=at_issue, drs=d2, embedding_level=1,\
          pred1=pred(arity=1, constant_name=_male_))’)
(3.7597, ’unresolved_discourse’, ’CREATED A CHUNK: drs(arg1=x3, drs=d2,\
          arg2=UNKNOWN, discourse_status=unresolved, embedding_level=1,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_male_))’)
(3.7817, ’PROCEDURAL’, ’RULE FIRED: attempting to resolve pronoun;\
          pronoun at embedding level 1’)
(3.8984, ’retrieval’, ’RETRIEVED: drs(dref=x1, arg1=x1,\
          discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_jeffrey_),\
          pred2=pred(arity=1, constant_name=_male_))’)
****Environment: {1: {’text’: ’argued’, ’position’: (320, 180)}}
(3.9094, ’PROCEDURAL’, ’RULE FIRED: resolution of PRO succeeded’)
(3.9094, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=x3, arg2=x1,\
          discourse_status=presupposed, drs=d2, embedding_level=1,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_male_))’)
(4.1306, ’PROCEDURAL’, ’RULE FIRED: project and complete: VP ==> Vt PP\
          (if_conseq_pred present, but also event cataphora)’)
(4.1306, ’discourse_context’, ’CREATED A CHUNK: drs(dref=e2, event_arg=e2,\
          arg1=x3, discourse_status=at_issue, drs=d2, embedding_level=1,\
          pred1=pred(arity=event_plus_2, constant_name=_argu_))’)
(4.1526, ’PROCEDURAL’, ’RULE FIRED: attempting to resolve cataphoric event\
          presupposition; antecedent at embedding level 1’)
(4.1526, ’unresolved_discourse’, ’CREATED A CHUNK: drs(arg1=UNKNOWN,\
          discourse_status=unresolved, dref=e2, drs=d2, embedding_level=1,\
          pred2=pred(arity=event_plus_2, constant_name=_argu_))’)
****Environment: {1: {’text’: ’with’, ’position’: (320, 180)}}
(4.2762, ’retrieval’, ’RETRIEVED: drs(arg1=UNKNOWN, arg2=e1,\
          discourse_status=unresolved, drs=d1, embedding_level=2,\
          pred1=pred(arity=2, constant_name=_precedes_),\
          pred2=pred(arity=event_plus_2, constant_name=_argu_))’)
```

```
(4.2872, 'PROCEDURAL', 'RULE FIRED: resolution of cataphoric AGAIN\        102
         succeeded')                                                       103
(4.2872, 'discourse_context', 'CREATED A CHUNK: drs(arg1=e2, arg2=e1,\     104
         discourse_status=presupposed, drs=d2, embedding_level=1,\         105
         pred1=pred(arity=2, constant_name=_precedes_),\                   106
         pred2=pred(arity=event_plus_2, constant_name=_argu_))')           107
                                                                           108
Time to read preposition: 0.3612000000000002                               109
```

In (96), we only list the processing steps that are most relevant to conditionals and cataphora resolution. After the proper name *Jeffrey*, the prepositional verb *(will) argue with* and the proper name *Danielle* contribute DRSs to the discourse context (lines 1–21), we start parsing the adverb *again*. The event DRS contributed by *(will) argue with* is recalled (lines 29–32), and the *again* presupposition is encoded in the `unresolved_discourse` buffer (lines 35–38). An attempt to anaphorically resolve this presupposition fails (lines 39–43), after which the process of *if*-triggered reanalysis begins, which contributes four new DRSs to the discourse context (lines 46–66): these are the four DRSs contributed by *Jeffrey*, *(will) argue with*, *Danielle* and *again*, except their embedding level is set to 2 (conditional consequent) instead of 0 (main assertion).

The pronoun *he* is then parsed and correctly resolved to the dref contributed by the proper name *Jeffrey* (lines 67–86). Then, as soon as the verb *argued* is parsed (lines 87–91), a cataphoric search attempting to resolve *again* is started (lines 92–96). The search is successfully completed (lines 98–101), so the *again*-contributed presupposition is resolved (lines 102–107).

The simulation ends with the time taken to read the preposition *with*, reported on line 109. This time crucially includes the *again* cataphoric search, so it can be used to model the Experiment 2 data for this ROI. We see that the time taken to read the preposition is about 360 ms, which is reasonable. In the next (Sect. 9.4.2), we will see that this time increases under specific conditions; and in the final (Sect. 9.4.3), we will fit the predicted RTs for this region to the Experiment 2 data.

9.4.2 Rules for 'Maximize Presupposition'

In Sect. 9.1.2 above, we noted that conditionals with matching VP meanings and no presuppositional *again*, like the one in (97) below (repeated from above), were significantly slower than conjunctions with matching meanings or conditionals with mismatching meanings.

(97) Jeffrey will argue with Danielle if he argued with her in the courtyard last night.

We conjectured that the processing difficulty associated with these conditionals was an effect of the Maximize Presupposition principle (Heim 1991), which requires that a presupposed VP meaning should be marked as such by *again*. This penalizes conditionals with matching VP meanings, while conditionals with non-identical VP meanings and coordinations should not be affected.

While Maximize Presupposition is commonly used as an explanatory principle in the formal semantics literature, there is no received way to formalize it and no explicit conjectures about the way it could become part of a mechanistic processing model of natural language interpretation.

In this section, we propose a tentative model of Maximize Presupposition processing as noise/error correction. Specifically, we conjecture that upon encountering the verb *argued* in the *if*-clause of example (97) above, the human processor pauses to consider whether it has erroneously failed to activate the `event_cataphora` feature in the goal buffer.

That is, given that the *if*-clause could satisfy a presuppositional *again* in the matrix clause, which would not violate Maximize Presupposition, the human processor hypothesizes that such an unresolved *again* presupposition might actually be present in declarative memory, but the `event_cataphora` feature in the goal buffer has erroneously failed to encode it.

Consequently, a search for an unresolved *again* presupposition is initialized, which adds extra reading time. In sum, the processing difficulty associated with a Maximize Presupposition violation is attributed to an extra retrieval request meant to check whether a goal feature was erroneously encoded.

This account is implemented in our mechanistic processing model by means of several rules. First, when the event DRS is recalled during the *if*-reanalysis process, its verbal predicate is stored in the goal buffer as the value of the `if_conseq_pred` feature (see the rule in (66) above).

Assuming the presence of such a feature, the rule in (98) below is triggered when we parse the matching verb *argued* in the *if*-clause. Specifically, the `if_conseq_pred` features has to be non-empty (line 18), and the `event_cataphora` feature should be turned off (`~True`; line 20). If that is the case, the usual actions associated with a prepositional verb are triggered (lines 25–51). Crucially, we also place a retrieval request for an unresolved event presupposition that we might have mistakenly failed to encode in the `event_cataphora` feature because of communication noise, comprehension noise, encoding noise etc. (lines 52–61).

(98)
```
parser.productionstring(name="project and complete: VP ==> Vt PP\      1
(if_conseq_pred present, matching pred)", string="""                   2
    =g>                                                                3
    isa                 parsing_goal                                   4
    task                parsing                                        5
    stack1              VP                                             6
    found               VtPP                                           7
    parsed_word         =w                                             8
    arg_stack1          =a1                                            9
    right_edge_stack1 VP                                              10
    right_edge_stack1 =re1                                            11
    right_edge_stack2 =re2                                            12
    right_edge_stack3 =re3                                            13
    right_edge_stack4 =re4                                            14
    event_peg           =ev_peg                                       15
    drs_peg             =drs_peg                                      16
    embedding_level     =el                                           17
    if_conseq_pred      ~None                                         18
    if_conseq_pred      =verbal_pred                                  19
    event_cataphora     ~True                                         20
    =retrieval>                                                       21
    isa                 word                                          22
    pred1              =verbal_pred
    ==>
    =g>
    isa                parsing_goal
    task               move_event_peg_and_wait_for_retrieval
    stack1             PP
    found              None
    parsed_word        None
    right_edge_stack1  PP
    right_edge_stack2  =re1
    right_edge_stack3  =re2
    right_edge_stack4  =re3
    +imaginal>
    isa                parse_state
    mother             =re2
    mother_of_mother   =re3
    node_cat           VP
    daughter1          VtPP
    daughter2          PP
    lex_head           =w
    +discourse_context>
    isa                drs
    dref               =ev_peg
    event_arg          =ev_peg
    arg1               =a1
    pred1              =verbal_pred
    drs                =drs_peg
    embedding_level    =el
    discourse_status   at_issue
    +retrieval>
    isa                drs
    dref               None
    arg1               UNKNOWN
    arg2               ~None
    pred1              ~None
    pred2              =verbal_pred
    drs                =drs_peg
    embedding_level    ~0
    discourse_status   unresolved
    ~imaginal>
    ~unresolved_discourse>
""")
```

In our example (97), this search for an unencoded *again* fails, and triggers the rule in (99) below.

(99)
```
parser.productionstring(name="search for an unencoded AGAIN failed",
string="""
    =g>
    isa                parsing_goal
    task               stop_resolution_attempt_AGAIN
    if_conseq_pred     ~None
    ?retrieval>
    state              error
    ?unresolved_discourse>
    buffer             empty
    ==>
    =g>
    isa                parsing_goal
    task               parsing
    ~retrieval>
""")
```

We can see all these rules in action, as well as their consequences for reading times, in (101) below.

(100) Jeffrey will argue with Danielle if he argued with [her in the courtyard last night].

(101)

```
****Environment: {1: {’text’: ’Jeffrey’, ’position’: (320, 180)}}
(0.2021, ’PROCEDURAL’, ’RULE FIRED: project: NP ==> ProperN’)
(0.2021, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x1, arg1=x1,\
          discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_jeffrey_),\
          pred2=pred(arity=1, constant_name=_male_))’)
****Environment: {1: {’text’: ’will’, ’position’: (320, 180)}}
****Environment: {1: {’text’: ’argue’, ’position’: (320, 180)}}
(0.9099, ’PROCEDURAL’, ’RULE FIRED: project and complete: VP ==> Vt PP\
          (no if_conseq_pred present)’)
(0.9099, ’discourse_context’, ’CREATED A CHUNK: drs(dref=e1, event_arg=e1,\
          arg1=x1, discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=event_plus_2, constant_name=_argu_))’)
****Environment: {1: {’text’: ’with’, ’position’: (320, 180)}}
(1.2635, ’PROCEDURAL’, ’RULE FIRED: project and complete: PP ==> P NP’)
****Environment: {1: {’text’: ’Danielle’, ’position’: (320, 180)}}
(1.6187, ’PROCEDURAL’, ’RULE FIRED: project and complete: NP ==> ProperN’)
(1.6187, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x2, arg1=x2,\
          discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_danielle_),\
          pred2=pred(arity=1, constant_name=_female_))’)
****Environment: {1: {’text’: ’if’, ’position’: (320, 180)}}
(1.9741, ’PROCEDURAL’, ’RULE FIRED: project and complete:\
          sentence-final if’)
(1.9961, ’PROCEDURAL’, ’RULE FIRED: start if-triggered reanalysis\
          (for sentence-final if)’)
****Environment: {1: {’text’: ’he’, ’position’: (320, 180)}}
(2.1434, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x2, arg1=x2,\
          discourse_status=at_issue, drs=d1, embedding_level=2,\
          pred1=pred(arity=1, constant_name=_danielle_),\
          pred2=pred(arity=1, constant_name=_female_))’)
(2.3020, ’discourse_context’, ’CREATED A CHUNK: drs(dref=e1, event_arg=e1,\
          arg1=x1, arg2=x2,\
          discourse_status=at_issue, drs=d1, embedding_level=2,\
          pred1=pred(arity=event_plus_2, constant_name=_argu_),\
          pred2=pred(arity=2, constant_name=_with_))’)
(2.4658, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x1, arg1=x1,\
          discourse_status=at_issue, drs=d1, embedding_level=2,\
          pred1=pred(arity=1, constant_name=_jeffrey_),\
          pred2=pred(arity=1, constant_name=_male_))’)
(2.6600, ’PROCEDURAL’, ’RULE FIRED: stop if-triggered reanalysis’)
(2.8687, ’PROCEDURAL’, ’RULE FIRED: project: NP ==> PRO’)
(2.8687, ’discourse_context’, ’CREATED A CHUNK: drs(dref=x3, arg1=x3,\
          discourse_status=at_issue, drs=d2, embedding_level=1,\
          pred1=pred(arity=1, constant_name=_male_))’)
(2.8687, ’unresolved_discourse’, ’CREATED A CHUNK: drs(arg1=x3, drs=d2,\
          arg2=UNKNOWN, discourse_status=unresolved, embedding_level=1,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_male_))’)
(2.8907, ’PROCEDURAL’, ’RULE FIRED: attempting to resolve pronoun;\
          pronoun at embedding level 1’)
(3.0058, ’retrieval’, ’RETRIEVED: drs(dref=x1, arg1=x1,\
          discourse_status=at_issue, drs=d1, embedding_level=0,\
          pred1=pred(arity=1, constant_name=_jeffrey_),\
          pred2=pred(arity=1, constant_name=_male_))’)
****Environment: {1: {’text’: ’argued’, ’position’: (320, 180)}}
(3.0168, ’PROCEDURAL’, ’RULE FIRED: resolution of PRO succeeded’)
(3.0168, ’discourse_context’, ’CREATED A CHUNK: drs(arg1=x3, arg2=x1,\
          discourse_status=presupposed, drs=d2, embedding_level=1,\
          pred1=pred(arity=2, constant_name=_equals_),\
          pred2=pred(arity=1, constant_name=_male_))’)
(3.2370, ’PROCEDURAL’, ’RULE FIRED: project and complete: VP ==> Vt PP\
         (if_conseq_pred present, matching pred)’)
(3.2370, ’discourse_context’, ’CREATED A CHUNK: drs(dref=e2, event_arg=e2,\
         arg1=x3, discourse_status=at_issue, drs=d2, embedding_level=1,\
         pred1=pred(arity=event_plus_2, constant_name=_argu_))’)
****Environment: {1: {’text’: ’with’, ’position’: (320, 180)}}
(3.4202, ’retrieval’, ’RETRIEVED: None’)
(3.4312, ’PROCEDURAL’, ’RULE FIRED: search for an unencoded AGAIN failed’)

Time to read preposition: 0.3978000000000015
```

The simulation in (101) proceeds as expected until we reach the crucial point, namely the project-and-complete-VP rule on lines (62–63). This rule triggers a memory search for an unencoded *again* presupposition, which takes place while the preposition *with* is being read (line 67). The search fails (lines 68–69), but the extra time needed for such a failed search can be seen in the higher reading time associated with the preposition, which is now almost 400 ms (line 71 in (101)).

9.4.3 Fitting the Model to the Experiment 2 Data

We are now ready to fit the model to part of the Experiment 2 data. Specifically, we will focus on the four match conditions in ((102a)–102d) below, and the two mismatch & cataphora conditions in (102e–102f).

(102) a. Jeffrey will argue with Danielle and he argued with her.
b. Jeffrey will argue with Danielle if he argued with her.
c. Jeffrey will argue with Danielle again and he argued with her.
d. Jeffrey will argue with Danielle again if he argued with her.
e. Jeffrey will argue with Danielle again and he played with her.
f. Jeffrey will argue with Danielle again if he played with her.

We do not attempt to model the remaining two conditions of Experiment 2 because our model does not really have to say anything about the mismatch & nothing, i.e., no-cataphora, cases. In fact, our model is not designed to capture the *and* & cataphora cases either, i.e., (102c) (match) or (102e) (mismatch), but we include them here for completeness.

Our model is set up to capture the *if*-conditions in (102b), (102d) and (102f), and we will focus on these conditions for most of our discussion in this subsection.

The mean RTs for the 6 conditions in (102) obtained in Experiment 2 (averaging over both subjects and items) are, in order: 364.05, 429.01, 390.13, 378.83, 374.28, and 387.72. The file `estimate_parser_parallel.py` (linked to in Appendix 9.6.4) lists these 6 conditions and the corresponding mean RTs, and provides the code for the Bayesian model that fits our incremental interpreter to data.

The Bayesian model is particularly simple: we only estimate the latency exponent and keep the other parameters fixed (see Appendix 9.6.1 for the exact values). The prior for the latency exponent is a half-Normal distribution—line 4 in (103) below. The ACT-R model we have introduced provides the likelihood function (lines 6–8). We sample `NDRAWS` (=1500) values from the posterior (lines 10–11).

(103)
```
parser_model = Model()
with parser_model:
    # Priors
    latency_exponent = HalfNormal('le', sd=0.3)
    # Likelihood
    pyactr_rt = actrmodel_latency(latency_exponent)
    mu_rt = Deterministic('mu_rt', pyactr_rt)
    rt_observed = Normal('rt_observed', mu=mu_rt, sd=30, observed=RT)
    # Compute posteriors
    step = pm.SMC()
    trace = sample(draws=NDRAWS, step=step, njobs=1)
```

The posterior estimates for the latency exponent and the 6 mean RTs are provided in (104) below and are plotted in Fig. 9.3.

(104)

	MEAN	SD	$\text{HPD}_{2.5\%}$	$\text{HPD}_{97.5\%}$	OBSERVED
le	0.03	0.02	0.0	0.06	N/A
μ_0 (conj-nothing-match)	352.63	0.84	351.1	354.00	364.05
μ_1 (cond-nothing-match)	385.25	12.53	366.0	408.40	429.01
μ_2 (conj-cata-match)	407.43	12.65	388.0	430.80	390.13
μ_3 (cond-cata-match)	371.55	9.23	353.5	385.70	378.83
μ_4 (conj-cata-mismatch)	407.43	12.65	388.0	430.80	374.28
μ_5 (cond-cata-mismatch)	387.72	0.16	387.5	388.00	387.72

Note that the `Rhat` values for this model are practically 1:

(105)
```
{'le': 0.9996690236773664,
 'mu_rt': array([0.99966776, 0.99966795, 0.99966794,
                 0.99972701, 0.99966794, 0.99976515])}
```

We see that the model captures the conditional & cataphora conditions—both match, μ_3, and mismatch, μ_5—very well. This is a consequence of the spreading

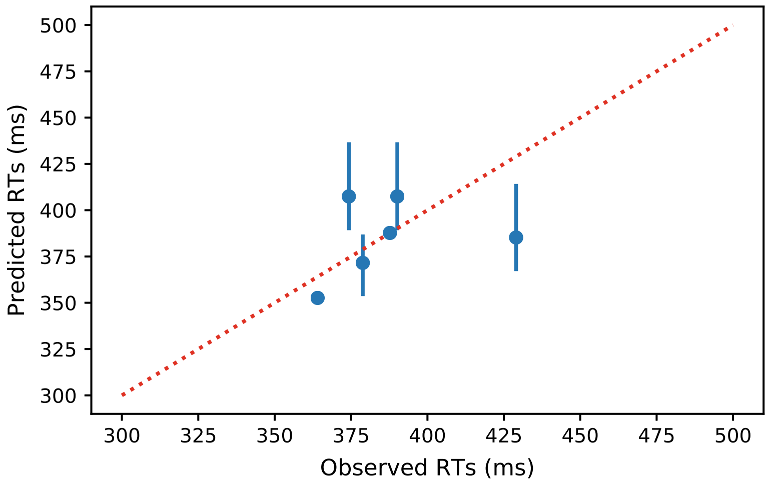


Fig. 9.3 Parser model: observed versus predicted RT

activation from the `discourse_context` buffer, which boosts the activation of the correct antecedent for the match condition μ_3, but which has no effect for the mismatch condition μ_5. The explanatory processing mechanism used here is spreading activation, i.e., the influence of the cognitive context on memory retrieval latency (and accuracy). This is the same explanatory mechanism as the one we used in the previous chapter when we captured the difference in latency between the semantic evaluation of sentences with varying fans.

The conjunction conditions are captured reasonably well, particularly the control condition μ_0 and the conjunction & cataphora & match condition μ_2. The model does not make any distinction between the two conjunction & cataphora conditions μ_2 (match) and μ_4 (mismatch): both retrieval requests proceed and fail the same way. Unfortunately, we overestimate the time required for retrieval failure, particularly for the mismatch μ_4 condition, where the observed value is a low 374.28 ms.

It might be that, by the time the human participants in the experiment read the preposition following the second finite verb, they realize that the stimulus overall is hopeless, and they give up on deeper processing rules like attempting a cataphoric search. This would explain why the observed RT for the μ_4 condition is fairly close to the control (conjunction) condition μ_0. One way to implement this in our model would be to have a rule that turns off the `event_cataphora` feature for overly difficult/incoherent conditions like μ_4. Firing this extra rule would add about 10 ms relative to the control condition μ_0, which would be almost exactly right.

Our attempt to capture the 'maximize presupposition' effect in the μ_1 condition provides a good qualitative fit, but quantitatively, the effect is greatly underestimated: the estimated mean is 385.25 ms, while the observed mean is 429.01 ms. Clearly, a failed attempt to retrieve an unencoded *again* from declarative memory is not sufficient to capture the processing effects of the semantic-pragmatic reasoning involved in ascertaining a failure to 'maximize presupposition'.

But the extra retrieval request and its failure are sufficient to capture the qualitative pattern. The estimated mean for μ_1 (conditionals with matching predicates and no cataphora) is greater than the estimated mean for the control condition μ_0 (conjunctions with matching predicates and no cataphora). It is also greater than the estimated mean for μ_3 (conditionals with matching predicates and cataphora).

Similarly, we capture the qualitative pattern involving the two cataphora & match cases: the estimated mean for μ_2 (conjunctions), where the attempt to resolve the *again* cataphora fails, is higher than the estimated mean for μ_3 (conditionals), where the attempt to resolve the *again* cataphora succeeds.

However, we do not capture the difference between conditionals and conjunctions in the cataphora & mismatch cases, i.e., μ_4 versus μ_5. We predict that conjunctions (estimated mean: 407.43 ms) take longer than conditionals (estimated mean: 387.72 ms). The observed values exhibit the opposite pattern: 374.28 and 387.72 ms, respectively. Once again, the observation we made above about conjunction & cataphora & mismatch cases could resolve this issue: if these conditions are too hard and human participants never even start a cataphoric search process, we expect the predicted pattern to be reversed.

We leave further developments of this model for future work. But given the relatively poor data fit exhibited by our complex model, it is reasonable to ask if going for a simpler, but less explanatory model, for example, a linear model of some sort, is not a better way to proceed.

We believe that the independently-motivated commitments we made to specific (i) formal semantics representations and theories, (ii) cognitive-architectural organization principles and constraints and (iii) language processing theories and models should not be abandoned in future iterations of this modeling endeavor. As Neal (1996) puts it:

> "Sometimes a simple model will outperform a more complex model [...] [But] deliberately limiting the complexity of the model is not fruitful when the problem is evidently complex. Instead, if a simple model is found that outperforms some particular complex model, the appropriate response is to define a different complex model that captures whatever aspect of the problem led to the simple model performing well." (Neal 1996, 103–104)

9.5 Conclusion

In this chapter, we built the first (to our knowledge) formally and computationally explicit mechanistic model of active anaphoric/cataphoric search for an antecedent. This model integrates rich semantic representations (independently motivated in the formal semantics literature) and processing mechanisms (independently motivated in the psycholinguistics literature) into a wide-coverage cognitive architecture (ACT-R).

In the spirit of van der Sandt (1992), Kamp (2001a, b), the model analyzes anaphora and presupposition as fundamentally processing-level phenomena that guide and constrain the cognitive process of integration, or linking, of new and old semantic information. Anaphora and presupposition have semantic effects, but they are not exclusively, or even primarily, semantics.

There are many open questions left for future research about (i) the exact nature of the semantic representations deployed in these models, (ii) the fine details of the processing mechanisms, (iii) how exactly these representations and processes should be integrated into a general, independently-constrained cognitive architecture, and (iv) the exact division of labor between semantics and processing for the analysis of anaphora and presupposition. We hope the modeling endeavor pursued in this chapter provides a framework for formulating these questions in a precise way, and for mounting a systematic search for answers.

We think that the most important lesson to be drawn from the extensive and detailed modeling attempt in this chapter is methodological. Computationally explicit mechanistic processing models that can be fit to experimental data are crucial when working at the interface between theoretical linguistics and experimental psycholinguistics in general, and at the semantics-psycholinguistics interface in particular.

Our argument for this is as follows. We started the chapter with a general question about the processing of semantic representations (is it incremental and predictive?).

To shed light on this question, we collected a reasonably rich amount of real-time experimental data, we analyzed the data with standard methods (mixed-effects linear models) and we informally stated an account that linked the theoretical question and the experimental data in a reasonably adequate way.

However, our ACT-R model, which explicitly formalized the proposed account, was able to quantitatively capture some, but crucially not all the data. This partial quantitative failure of our detailed, computationally-explicit cognitive model is enlightening: it opens up a variety of specific questions for future research that would have been missed had we stayed at the informal level of our initial account.

Specifically, it turns out that for the informal account to really work, we need auxiliary hypotheses that are both crucial for and completely glossed over by that informal account. Looking carefully only at the experimental data (or the results of standard statistical methods applied to that data), or only at the formal semantic representations, or even at both, but separately, is not enough. We need to be formally and computationally explicit about how we link them via mechanistically-specified processing models.

This lesson is not new by any means. In fact, it is very familiar to generative linguists when it comes to formulating competence-level theories:

> "Precisely constructed models for linguistic structure can play an important role, both negative and positive, in the process of discovery itself. By pushing a precise but inadequate formulation to an unacceptable conclusion, we can often expose the exact source of this inadequacy and, consequently, gain a deeper understanding of the linguistic data. More positively, a formalized theory may automatically provide solutions for many problems other than those for which it was explicitly designed. Obscure and intuition-bound notions can neither lead to absurd conclusions nor provide new and correct ones, and hence they fail to be useful in two important respects. [We need] to recognize the productive potential in the method of rigorously stating a proposed theory and applying it strictly to linguistic material with no attempt to avoid unacceptable conclusions by *ad hoc* adjustments or loose formulation." (Chomsky 1957, p. 5)

Computational modeling of cognitive phenomena has become increasingly central to cognitive science for fundamentally the same reason. As Lewandowsky and Farrell (2010, p. 9) put it: "[e]ven intuitively attractive notions may fail to provide the desired explanation for behavior once subjected to the rigorous analysis required by a computational model."

We take the explicit focus on *computationally-specified mechanistic processing models* that are both (i) theoretically informed and (ii) quantitatively fit to experimental data in a statistically informed and thoughtful way to be a distinguishing feature of research at the *semantics-psycholinguistics interface*. We distinguish this kind of work from experimentally-informed semantics, as well as from semantically-informed psycholinguistics.

In our view, work in experimentally-informed semantics engages primarily with semantic theories using empirical investigation methodologies (mostly offline: forced choice, acceptability etc.) that have become standard in psycholinguistics. But the semantic theories are connected to the experimental measurements of linguistic behavior only implicitly and/or informally, hence weakly. In addition, designing properly powered experiments in semantics and pragmatics, where the effects are usually subtle and difficult to detect, is non-trivial,[7] which makes the presumed links between theory and experimental data even more tenuous.

Similarly, work in semantically-informed psycholinguistics engages only in informal ways with formal semantics theories. While insightful, this work falls short of the standard of systematicity and formalization that permeates work in formal semantics, and does not engage in substantial ways with formal semantics frameworks and systems as a whole.

In sum, taking real-time experimental data and explicit computational modeling seriously opens up exciting new directions of research in formal semantics, and new ways of (re)connecting formal semantics and the broader field of cognitive science. This is the reason for the copious amounts of code and behavioral data introduced in this chapter, and in the book overall.

The specifics of the code and models we included in the book will likely become obsolete in the near future, just as many of our other auxiliary assumptions will. That is perfectly fine: their main purpose is to get the larger project off the ground and demonstrate its feasibility. Ultimately, our intention was to argue for a new range of theoretical and empirical goals for semantics, introduce an appropriate research workflow, and help semanticists and psycholinguists start using it.

9.6 Appendix: The Complete Syntax/Semantics Parser

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

9.6.1 File ch9/parser_dm.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch9/parser_dm.py.

9.6.2 File ch9/parser_rules.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch9/ parser_rules.py.

[7]See Kruschke (2011), Vasishth and Nicenboim (2016), Nicenboim and Vasishth (2016) among others for detailed discussions of power and related statistical issues.

9.6.3 File ch9/run_parser.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch9/run_parser.py.

9.6.4 File ch9/estimate_parser_parallel.py

https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch9/estimate_parser_parallel.py.

Chapter 10
Future Directions

Where do we go from here? We will keep this short because, if the reader has made it this far, the answer really is: in whatever direction the reader's research interests lie. The main purpose of this book was to introduce a general framework and workflow that enables us to enhance competence theories with fully specified performance/processing components. The resulting competence-performance theories can furthermore be embedded in Bayesian models, which enables us to fit them to data and quantitatively compare them in a systematic fashion. This Bayes+ACT-R+formal linguistics workflow of model development is in principle applicable to linguistic accounts of many syntactic and/or semantic phenomena—*if* suitable data can be obtained from properly designed experiments, which is far from trivial.

This being said, we think there are five specific directions worth pursuing in the near future:

i. add more structure to the Bayesian models, for example, random effects for participants, grouping participants according to their strategies in self-paced reading tasks, etc.;
ii. data-driven modeling: hand-coding models for specific experiments does not scale up well, and we should find ways to leverage syntactically and semantically annotated corpora to make the process of building ACT-R models for specific tasks and experiments more automatic and data-driven, and more easily comparable across tasks/experiments;
iii. enrich the range of studied semantic phenomena—quantifiers, scope, binding, questions, attitude verbs, modals—and the range of semantic representations that are considered—(trees of) variable assignments in addition to or instead of DRSs, compositionally assembled higher-order terms in a suitable logical system etc.

A. Brasoveanu and J. Dotlačil, *Computational Cognitive Modeling and Linguistic Theory*, Language, Cognition, and Mind 6,
https://doi.org/10.1007/978-3-030-31846-8_10

  - relatively modest extensions of this framework could be used to build on the wealth of experimental results gathered in the last ten years or so and explicitly model and fit to data different theories of presupposition projection, scalar implicature computation etc., not to mention the large amount of experimental data about syntactic phenomena that is available in the literature;

iv. provide a framework for integrating and comparing models and theories of language interpretation that have been developed in largely disparate traditions up to this point:

  - for example, the rise of distributional semantics and neural-network modeling work in formal semantics (Bowman 2016; McNally and Boleda 2017 among others), and linguistics more generally, raises a range of questions about what the appropriate division of labor is in natural language interpretation between symbolic and subsymbolic components; our Bayes+ACT-R+formal linguistics framework enables us to explore a range of hybrid models that would integrate both perspectives and that can be quantitatively compared, for example, models in which more of the cognitive heavy-lifting is performed either by symbolic components (chunks, rules) or subsymbolic components (base/spreading activation, rule utilities); see, for example, Marcus (2018) for a recent discussion of and arguments for hybrid (symbolic and subsymbolic) architectures;
  - a specific example would involve exploring hybrid representations for lexical items that would encode both structural information (like we have done throughout this book) and quantitative information, e.g., dense word embeddings of the kind proposed in Mikolov et al. (2013) or Pennington et al. (2014); these dense word embeddings could be used to modulate spreading activation for lexical items or for larger phrasal units;
  - incorporating drift-diffusion models (Ratcliff 1978; Ratcliff et al. 2017) into ACT-R (cf. (Van Maanen et al. (2012))) and compare the resulting model(s) of language comprehension with other commonly used modeling choices;
  - yet another possibility is to systematically investigate rule learning for natural language interpretation: rules throughout this book were hand-coded, and no theory for how new rules are generated was put forth; this is a common feature of ACT-R modeling, but not a defining and necessary one: ACT-R does have a system for rule learning (production compilation) and we could go further by hypothesizing ‘rule-generating’ mechanisms;
  - similarly, ACT-R has a system for rule utility learning, but recent advances in reinforcement learning might contribute new insights to this component of the cognitive architecture.

v. on the computational side, make improvements to enable faster estimation of posterior distributions for `pyactr` model parameters, e.g., by emulating `pyactr` models with neural networks, Gaussian Processes or other kinds of models; solutions along these lines could also enable us to do Approximate Bayesian Computation (ABC), that is, likelihood-free Bayesian inference for simulation-based models with intractable likelihoods, e.g., ACT-R models with various stochastic components turned on.

In addition, there are several ways in which ACT-R is showing its age for modeling natural language interpretation:

- it has a rule-ordering architecture that effectively employs transformational models of the kind generative linguistics used in the '60s and '70s, and that we moved away from;
- it has a fairly strict ban on hierarchical structures, rather than a softer one that would allow but penalize them, e.g., the way a probabilistic context free grammar penalizes deeper trees;
- the underlying logic for facts/chunks is the logic of feature structures, basically a modal logic with features as modal operators and values as (atomic) non-modal sentence variables; in semantics, we have moved away from this type of theory construction with 'local'-perspective logics, and more towards the 'global'-perspective of classical (many-sorted) first-order or higher-order logic, which makes integrating ACT-R and formal semantics somewhat awkward.

However, ACT-R is a widely used hybrid (symbolic and subsymbolic) cognitive architecture and as such, it was the obvious choice for a framework in which to build mechanistic processing models and integrated competence-performance theories for natural language interpretation. As computational cognitive modeling for natural language phenomena develops further, we expect to see a critical reevaluation of a variety of architectural assumptions that we took for granted in the present work.

Bibliography

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs* (2nd ed.). Cambridge: MIT Press/McGraw-Hill.

Abney, S., & Johnson, M. (1991). Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, *20*, 233–50.

Abusch, D. (2010). Presupposition triggering from alternatives. *Journal of Semantics*, *27*, 37–80.

Anderson, C. (2004). *The structure and real-time comprehension of quantifier scope ambiguity*. Doctoral Dissertation, Northwestern University, Evanston, Illinois.

Anderson, J. R. (1974). Retrieval of propositional information from long-term memory. *Cognitive Psychology*, *6*, 451–474.

Anderson, J. R. (1976). *Language, memory, and thought*. Hillsdale, NJ: Erlbaum.

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, *89*, 369.

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

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

Anderson, J. R., Bothell, D., & Byrne, M. D. (2004). An integrated theory of the mind. *Psychological Review*, *111*, 1036–1060.

Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, J. R., & Reder, L. M. (1999). The fan effect: New results and new theories. *Journal of Experimental Psychology: General*, *128*, 186–197.

Anderson, J. R., Reder, L. M., & Lebiere, C. (1996). Working memory: Activation limitations on retrieval. *Cognitive Psychology*, *30*, 221–256.

Anderson, J. R., & Schooler, L. J. (1991). Reflections of the environment in memory. *Psychological Science*, *2*, 396–408.

Barr, D. J., Levy, R., Scheepers, C., & Tily, H. J. (2013). Random effects structure for confirmatory hypothesis testing: Keep it maximal. *Journal of Memory and Language*, *68*, 255–278.

Bell, C. G., & Newell, A. (1971). *Computer structures: Readings and examples*. New York: McGraw-Hill.

Bhatt, R., & Roumyana, P. (2006). Conditionals. In E. Martin & H. van Riemsdijk (Eds.), *The Blackwell companion to syntax* (pp. 638–687). Wiley.

Bos, J. (2005). Towards wide-coverage semantic representation. In *Proceedings of the 6th International Workshop on Computational Semantics (IWCS '05)* (pp. 42–53). Tilburg: University of Tilburg.

Bos, J., Clark, S., Steedman, M., Curran, J. R., & Hockenmaier, J. (2004). Wide-coverage semantic representations from a CCG parser. In *Proceedings of the 20th international conference on computational linguistics* (p. 1240). Association for Computational Linguistics.

Boston, M. F., Hale, J. T., Vasishth, S., & Kliegl, R. (2011). Parallel processing and sentence comprehension difficulty. *Language and Cognitive Processes*, *26*, 301–349.

Bowman, S. R. (2016). *Modeling natural language semantics in learned representations*. Doctoral Dissertation, Stanford University.

Brasoveanu, A. (2007). *Structured nominal and modal reference*. Doctoral Dissertation, Rutgers Univ.

Brasoveanu, A., & Dotlačil, J. (2015a). Incremental and predictive interpretation: Experimental evidence and possible accounts. *Proceedings of Semantics and Linguistic Theory (SALT)*, *25*, 57–81.

Brasoveanu, A., & Dotlačil, J. (2015b). Incremental interpretation and dynamic semantics. UC Santa Cruz and University of Groningen ms. http://people.ucsc.edu/abrsvn/inc_int_and_dyn_sem.pdf.

Brasoveanu, A., & Dotlačil, J. (2015c). Strategies for scope taking. *Natural Language Semantics*, *23*, 1–19.

Brasoveanu, A., & Dotlačil, J. (2018). An extensible framework for mechanistic processing models: From representational linguistic theories to quantitative model comparison. In *Proceedings of the 2018 international conference on cognitive modelling*.

Brasoveanu, A., & Dotlačil, J. (2020). Donkey anaphora: Farmers and bishops. In L. Matthewson, C. Meier, H. Rullmann, & T. E. Zimmerman (Eds.), *Blackwell companion to semantics*. Wiley.

Brasoveanu, A., & Farkas, D. (2011). How indefinites choose their scope. *Linguistics and Philosophy*, *34*, 1–55.

Budiu, R., & Anderson, J. R. (2004). Interpretation-based processing: A unified theory of semantic sentence processing. *Cognitive Science*, *28*, 1–44.

Budiu, R., & Anderson, J. R.. (2005). Negation in nonliteral sentences. In B. Bara, L. Barsalou & M. Bucciarelli (Eds.), *Proceedings of the 27th annual conference of the cognitive science society* (pp. 354–359). Lawrence Erlbaum Associates.

Carlson, G. N. (1977). A unified analysis of the English bare plural. *Linguistics and Philosophy* 1.

Carlson, G. N. (1980). *Reference to kinds in English*. New York: Garland Publishing.

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

Chater, N., Pickering, M., & Milward, D. (1995). What is incremental interpretation? In D. Milward & P. Sturt (Eds.), *Incremental interpretation (Edinburgh working papers in cognitive science)* (Vol. 11, pp. 1–23). Edinburgh: Edinburgh University.

Chierchia, G. (1995). *Dynamics of meaning: Anaphora, presupposition, and the theory of grammar*. Chicago: University of Chicago Press.

Chomsky, N. (1956). Three models for the description of language. *IEEE Transactions on Information Theory*, *2*, 113–124.

Chomsky, N. (1957). *Syntactic structures*. The Hague: Mouton.

Chomsky, N. (1981). *Lectures on government and binding*. Dordrecht: Foris.

Creal, D. (2012). A survey of sequential monte carlo methods for economics and finance. *Econometric Reviews*, *31*, 245–296.

Cresswell, M. J. (1985). *Structured meanings: The semantics of propositional attitudes. Bradford books*. Cambridge, MA, USA: MIT Press.

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

Dekker, P. (1994). Predicate logic with anaphora. In L. Santelmann & M. Harvey (Eds.), *Proceedings of SALT IV* (pp. 79–95). Ithaca: CLC Publications.

Demers, A. J. (1977). Generalized left corner parsing. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 170–182). ACM.

Dillon, B., Mishler, A., Sloggett, S., & Phillips, C. (2013). Contrasting intrusion profiles for agreement and anaphora: Experimental and modeling evidence. *Journal of Memory and Language*, *69*, 85–103.

Dotlačil, J. (2018). Building an act-r reader for eye-tracking corpus data. *Topics in Cognitive Science*, *10*, 144–160.

Downey, A. (2012). *Think python*. O'Reilly Media, Inc.

Ebbinghaus, H. (1913). *Memory: A contribution to experimental psychology*. New York: Teachers College, Columbia University. http://psychclassics.yorku.ca/Ebbinghaus/index.htm.

Elbourne, P. (2009). Bishop sentences and donkey cataphora: A response to barker and shan. *Semantics and Pragmatics*, *2*, 1–7.

Engelmann, F., Vasishth, S., Engbert, R., & Kliegl, R. (2013). A framework for modeling the interaction of syntactic processing and eye movement control. *Topics in Cognitive Science*, *5*, 452–474.

Farkas, D. F., & de Henriëtte, S. (2003). *The semantics of incorporation: From argument structure to discourse transparency*. Stanford: CSLI Publications.

Forster, K. (1992). Memory-addressing mechanisms and lexical access. In R. Frost & L. Katz (Eds.), *Orthography, phonology, morphology, and meaning* (Vol. 94, pp. 413–434). Amsterdam: North-Holland.

Forster, K. I. (1976). Accessing the mental lexicon. In R. J. Wales & E. Walker (Eds.), *New approaches to language mechanisms* (Vol. 30, pp. 257–287). Amsterdam: North-Holland.

Forster, K. I. (1990a). *Lexical processing*. The MIT Press.

Forster, K. (1990b). Lexical processing. In D. Osherson & H. Lasnik (Eds.), *Language: An invitation to cognitive science* (pp. 95–131). Cambridge, MA: MIT Press.

Frazier, L., & Fodor, J. D. (1978). The sausage machine: A new two-stage parsing model. *Cognition*, *6*, 291–325.

Fuchs, A. (1971). The saccadic system. In *The control of eye movements* (pp. 343–362).

Gallin, D. (1975). *Intensional and higher-order modal logic*. Amsterdam: North-Holland Mathematics Studies.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian data analysis*, 3rd edn. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.

Gelman, A., & Hill, J. (2007). *Data analysis using regression and multilevel/hierarchical models*. Analytical Methods for Social Research: Cambridge University Press.

Gibson, E. (1991). *A computational theory of human linguistic processing: Memory limitations and processing breakdown*. Doctoral Dissertation, Carnegie Mellon University, Pittsburgh, PA.

Gibson, E. (1998). Linguistic complexity: Locality of syntactic dependencies. *Cognition*, *68*, 1–76.

Grodner, D., & Gibson, E. (2005). Consequences of the serial nature of linguistic input for sentenial complexity. *Cognitive Science*, *29*, 261–291.

Groenendijk, J., & Stokhof, M. (1990). Dynamic Montague grammar. In *Proceedings of the second symposium on logic and language* (pp. 3–48).

Groenendijk, J., & Stokhof, M. (1991). Dynamic predicate logic. *Linguistics and Philosophy*, *14*, 39–100.

Hagoort, P., Hald, L., Bastiaansen, M., & Petersson, K. M. (2004). Integration of word meaning and world knowledge in language comprehension. *Science*, *304*, 438–441.

Hale, J. (2011). What a rational parser would do. *Cognitive Science*, *35*, 399–443.

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

Hart, B., & Risley, T. R. (1995). *Meaningful differences in the everyday experience of young american children*. Baltimore: Paul H Brookes Publishing.

Heim, I. (1982). *The semantics of definite and indefinite noun phrases* (published 1988, New York: Garland). Doctoral Dissertation, UMass Amherst, Amherst, MA.

Heim, I. (1991). Artikel und definitheit. In A. von Stechow & D. Wunderlich (Eds.), *Semantik: Ein internationales handbuch der zeitgenössischen forschung* (pp. 40–64). Berlin: Walter de Gruyter.

Heim, I., & Kratzer, A. (1998). *Semantics in generative grammar*. Oxford: Blackwell.

Hofmeister, P., Casasanto, L. S., & Sag, I. A. (2013). Islands in the grammar? standards of evidence. In J. Sprouse & H. Hornstein (Eds.), *Experimental syntax and island effects* (pp. 42–63). Cambridge University Press.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). *Introduction to automata theory, languages, and computation*. Addison-Wesley.

Hough, J., Kennington, C., Schlangen, D., & Ginzburg, J. (2015). Incremental semantics for dialogue processing: Requirements and a comparison of two approaches. In *Proceedings of the International Workshop on Computational Semantics (IWCS)*.

Howes, D. H., & Solomon, R. L. (1951). Visual duration threshold as a function of word-probability. *Journal of Experimental Psychology*, *41*, 401.

Iatridou, S. (1991). *Topics in conditionals*. Doctoral Dissertation, MIT.

Jäger, L. A, Benz, L., Roeser, J., Dillon, B. W, & Vasishth, S. (2015). Teasing apart retrieval and encoding interference in the processing of anaphors. *Frontiers in psychology, 6*.

Jäger, L. A., Benz, L., Roeser, J., Dillon, B. W., & Vasishth, S. (2017). Similarity-based interference in sentence comprehension: Literature review and bayesian meta-analysis. *Journal of Memory and Language*, *94*, 316–339.

Johnson-Laird, P. N. (1983). *Mental models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press.

Johnson-Laird, P. N. (2004). *The history of mental models* (pp. 179–212). Psychology of reasoning: Theoretical and historical perspectives.

Just, M. A., & Carpenter, P. A. (1989). Reasoning by model: The case of multiple quantification. *Psychological Review*, *96*, 658–673.

Just, M. A., Carpenter, P. A., & Woolley, J. D. (1980). A theory of reading: From eye fixations to comprehension. *Psychological Review*, *87*, 329–354.

Kamp, H. (1982). Paradigms and processes in reading comprehension. *Journal of Experimental Psychology: General*, *111*, 228–238.

Kamp, H. (1981). A theory of truth and semantic representation. In C. Rohrer, A. Rossdeutscher, & H. Kamp (Eds.), *Formal methods in the study of language* (pp. 277–322). Amsterdam: Mathematical Centre Tracts.

Kamp, H. (2001a). Presupposition computation and presupposition justification. In M. Bras & L. Vieu (Eds.), *Semantic and pragmatic issues in discourse and dialogue* (pp. 57–84). Amsterdam: Elsevier.

Kamp, H. (2001b). The Importance of Presupposition. In C. Rohrer, A. Rossdeutscher & H. Kamp (Eds.), *Linguistic form and its computation*. Studies in computational linguistics (pp. 207–254). CSLI Publications.

Kamp, H., & Reyle, U. (1993). *From discourse to logic. introduction to model theoretic semantics of natural language, formal logic and Discourse Representation Theory*. Dordrecht: Kluwer.

Kaplan, R. M., Bresnan, J., et al. (1982). Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan (Ed.), *The mental representation of grammatical relations* (pp. 173–281). Cambridge, MA: MIT Press.

Kazanina, N., Lau, E. F., Lieberman, M., Yoshida, M., & Phillips, C. (2007). The effect of syntactic constraints on the processing of backwards anaphora. *Journal of Memory and Language*, *56*, 384–409.

Klima, E. (1964). Negation in English. In J. A. Fodor & J. J. Katz (Eds.), *The Structure of Language: Readings in the Philosophy of Language* (pp. 246–323). Englewood Cliffs: Prentice-Hall.

Kruschke, J. K. (2011). *Doing Bayesian data analysis: A tutorial with R and BUGS*. Academic Press/Elsevier.

Kurtzman, H. S., & MacDonald, M. C. (1993). Resolution of quantifier scope ambiguities. *Cognition*, *48*, 243–279.

Kush, D., Lidz, J., & Phillips, C. (2015). Relation-sensitive retrieval: evidence from bound variable pronouns. *Journal of Memory and Language*, *82*, 18–40.

Ladusaw, W. (1979). *Polarity sensitivity as inherent scope relations*. Doctoral Dissertation, University of Texas.

Lambert, B. (2018). A student's guide to bayesian statistics. *SAGE*,. Publications.

Lamport, L. (1986). *Latex: A document preparation system*. Boston, MA, USA: Addison-Wesley.

Lau, E. F. (2009). *The predictive nature of language comprehension*. Doctoral Dissertation, University of Maryland, College Park.

Lebiere, C. (1999). The dynamics of cognition: An act-r model of cognitive arithmetic. *Kognitionswissenschaft*, *8*, 5–19.

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

Lewis, R., & Vasishth, S. (2005). An activation-based model of sentence processing as skilled memory retrieval. *Cognitive Science*, *29*, 1–45.

Logan, G. D. (1990). Repetition priming and automaticity: Common underlying mechanisms? *Cognitive Psychology*, *22*, 1–35.

Lynch, S. M. (2007). *Introduction to applied Bayesian statistics and estimation for social scientists*. New York, NY: Springer Science & Business Media LLC.

Marcus, G. (2018). Deep learning: A critical appraisal. *CoRR*.,. arXiv:1801.00631.

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. San Francisco: W. H. Freeman and Company.

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

Marslen-Wilson, W. (1975). Sentence perception as an interactive parallel process. *Science*, *189*, 226–228.

Mätzig, P., Vasishth, S., Engelmann, F., Caplan, D., & Burchert, F. (2018). A computational investigation of sources of variability in sentence comprehension difficulty in aphasia. *Topics in Cognitive Science*, *10*, 161–174.

McElree, B. (2006). Accessing recent events. In B. H. Ross (Ed.), *Psychology of learning and motivation* (Vol. 46, pp. 155–200). Academic Press. http://www.sciencedirect.com/science/article/pii/S0079742106460059.

McNally, L., & Boleda, G. (2017). Conceptual versus referential affordance in concept composition. In Y. Winter & J. Hampton (Eds.), *Compositionality and concepts in linguistics and psychology*. *Language, Cognition, and Mind* (Vol. 3, pp. 245–267). Springer.

Meyer, D. E., & Kieras, D. E. (1997). A computational theory of executive cognitive processes and multiple-task performance: Part i. basic mechanisms. *Psychological Review, 104*, 3.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2–4, 2013, Workshop Track Proceedings*. arXiv:1301.3781.

Milward, D., & Cooper, R. (1994). Incremental interpretation: Applications, theory, and relationship to dynamic semantics. In *The 15th International Conference on Computational Linguistics (COLING 94)* (pp. 748–754). Kyoto, Japan: COLING 94 Organizing Comm.

Monsell, S. (1991). The nature and locus of word frequency effects in reading. In D. Besner & G. W. Humphreys (Eds.), *Basic processes in reading: Visual word recognition* (pp. 148–197). Hillsdale, NJ: Erlbaum.

Montague, R. (1970). English as a formal language. In B. Visentini, et al. (Eds.), *Linguaggi nella società e nella tecnica* (pp. 189–224). Milan: Edizioni di Communità.

Montague, R. (1973). The proper treatment of quantification in ordinary English. In P. S. J. Hintikka & J. Moravcsik (Eds.), *Approaches to natural language* (pp. 221–242). Dordrecht: Reidel.

Murry, W. S., & Forster, K. I. (2004). Serial mechanisms in lexical access: the rank hypothesis. *Psychological Review*, *111*, 721.

Muskens, R. (1995a). *Meaning and partiality. Studies in Logic Language and Information*. Cambridge University Press.

Muskens, R. A. (1995b). Tense and the logic of change. In U. Egli, P. E. Pause, C. Schwarze, A. von Stechow, & G. Wienold (Eds.), *Lexical knowledge in the organization of language* (pp. 147–183). Amsterdam: Benjamins.

Muskens, R. A. (1996). Combining Montague Semantics and Discourse Representation. *Linguistics and Philosophy*, *19*, 143–186.

Neal, R. M. (1996). *Bayesian learning for neural networks*. Berlin, Heidelberg: Springer-Verlag.

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

Newell, A. (1973a). Production systems: Models of control structures. In W. G. Chase, et al. (Eds.), *Visual information processing* (pp. 463–526). New York: Academic Press.

Newell, A. (1973b). You can't play 20 questions with nature and win: Projective comments on the papers of this symposium. In W. G. Chase, et al. (Eds.), *Visual information processing* (pp. 283–308). New York: Academic Press.

Newell, A., & Rosenbloom, P. S. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 1–55). Hillsdale, NJ: Erlbaum.

Nicenboim, B., & Vasishth, S. (2016). Statistical methods for linguistic research: Foundational ideas part ii. *Language and Linguistics Compass*, *10*, 591–613. 10.1111/lnc3.12207 https://onlinelibrary.wiley.com/.

Nicenboim, B., & Vasishth, S. (2018). Models of retrieval in sentence comprehension: A computational evaluation using bayesian hierarchical modeling. *Journal of Memory and Language*, *99*, 1–34.

Nouwen, R. (2003). *Plural pronominal anaphora in context: Dynamic aspects of quantification*. Doctoral Dissertation, UIL-OTS, Utrecht University.

Nouwen, R. (2007). On dependent pronouns and dynamic semantics. *Journal of Philosophical Logic*, *36*, 123–154.

Partee, B. (2011). The semantics adventure. https://udrive.oit.umass.edu/partee/Partee2011_MIT150.pdf.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP* (pp. 1532–1543). ACL.

Phillips, C. (1996). *Order and structure*. Doctoral Dissertation, Massachusetts Institute of Technology.

Phillips, C. (2003). Linear order and constituency. *Linguistic Inquiry*, *34*, 37–90.

Phillips, C., & Lewis, S. (2013). Derivational order in syntax: evidence and architectural consequences. *Studies in Linguistics*, *6*, 11–47.

Pickering, Martin J., McElree, Brian, Frisson, Steven, Chen, Lillian, & Traxler, Matthew J. (2006). Underspecification and aspectual coercion. *Discourse Processes*, *42*, 131–155.

Poesio, M. (1994). *Discourse interpretation and the scope of operators*. Doctoral Dissertation, University of Rochester.

Polanyi, M. (1967). *The tacit dimension*. London: Routledge and Kegan Paul.

Pollard, C., & Sag, I. A. (1994). *Head-driven phrase structure grammar*. University of Chicago Press.

Poore, G. M. (2013). Reproducible documents with pythontex. In S. van der Walt, J. Millman & K. Huff (Eds.), *Proceedings of the 12th Python in Science Conference* (pp. 78–84).

Pylkkänen, L., & McElree, B. (2006). The syntax-semantic interface: On-line composition of sentence meaning. In M. Traxler & M. A. Gernsbacher (Eds.), *Handbook of psycholinguistics* (pp. 537–577). New York: Elsevier.

Pylyshyn, Z. W. (1989). The role of location indexes in spatial perception: A sketch of the finst spatial-index model. *Cognition*, *32*, 65–97.

Pylyshyn, Z. W. (2007). *Things and places: How the mind connects with the world*. Mass: Bradford books.

R Core Team. (2014). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/.

Ramalho, L. (2015). *Fluent python: clear, concise, and effective programming*. O'Reilly Media, Inc.

Ratcliff, R. (1978). A theory of memory retrieval. *Psychological Review*, *85*, 59.

Ratcliff, R., Smith, P. L., Brown, S. D., & McKoon, G. (2017). Diffusion decision model: Current issues and history. *Trends in Cognitive Sciences*, *20*, 260–281.

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

Reichle, E. D., Pollatsek, A., Fisher, D. L., & Rayner, K. (1998). Toward a model of eye movement control in reading. *Psychological Review*, *105*, 125.

Reitter, D., Keller, F., & Moore, J. D. (2011). A computational cognitive model of syntactic priming. *Cognitive Science*, *35*, 587–637.

Resnik, P. (1992). Left-corner parsing and psychological plausibility. In *Proceedings of the fourteenth international conference on computational linguistics*. Nantes, France.

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

Sag, I. A. (1992). Taking performance seriously. In C. Martin-Vide (Ed.), *VII Congreso de Languajes Naturales y Lenguajes Formales*. Barcelona. http://lingo.stanford.edu/sag/papers/vic-paper.pdf.

Sag, I. A., & T. Wasow. (2011). Performance-compatible competence grammar. In R. D. Borsley & K. Börjars (Eds.), *Non-transformational syntax: Formal and explicit models of grammar* (pp. 189–199). Blackwell Publishing.

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

van der Sandt, R. (1992). Presupposition projection as anaphora resolution. *Journal of Semantics*, *9*, 333–377.

Schilling, H. E. H., Rayner, K., & Chumbley, J. I. (1998). Comparing naming, lexical decision, and eye fixation times: Word frequency effects and individual differences. *Memory & Cognition*, *26*, 1270–1281.

Schooler, L. J., & Anderson, J. R. (1997). The role of process in the rational analysis of memory. *Cognitive Psychology*, *32*, 219–250.

Schwarz, F. (2014). Presuppositions are fast, whether hard or soft-evidence from the visual world. In M. Wiegand, T. Snider & S. D'Antonio (Eds.), *Semantics and Linguistic Theory (SALT)* (Vol. 24, pp. 1–22). LSA and CLC Publications.

Shieber, S. M. (2003). *An introduction to unification-based approaches to grammar*. Microtome Publishing.

Sisson, S., Fan, Y., & Beaumont, M. (Eds.). (2019). *Handbook of approximate bayesian computation*. New York: Chapman and Hall/CRC.

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

Steedman, M. (2001). *The syntactic process*. Cambridge, MA: MIT Press.

Stowe, L. A. (1986). Parsing wh-constructions: Evidence for on-line gap location. *Language and Cognitive Processes*, *1*, 227–245.

Taatgen, N. A., & Anderson, J. R. (2002). Why do children learn to say "broke"? a model of learning the past tense without feedback. *Cognition*, *86*, 123–155.

Taatgen, N. A., Juvina, I., Schipper, M., Borst, J. P., & Martens, S. (2009). Too much control can hurt: A threaded cognition model of the attentional blink. *Cognitive Psychology*, *59*, 1–29.

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

Traxler, M. J., & Pickering, M. J. (1996). Plausibility and the processing of unbounded dependencies: An eye-tracking study. *Journal of Memory and Language*, *35*, 454–475.

Trueswell, J., Tanenhaus, M., & Garnsey, S. (1994). Semantic influences on parsing: Use of thematic role information in syntactic ambiguity resolution. *Journal of Memory and Language*, *33*, 285–318.

Tunstall, S. (1998). *The interpretation of quantifiers: Semantics and processing*. Doctoral Dissertation, University of Massachusetts, Amherst.

van Maanen, L., van Rijn, H., & Taatgen, N. (2012). RACE/A: An architectural account of the interactions between learning, task control, and retrieval dynamics. *Cognitive Science*, *36*, 62–101.

Vasishth, S., Bruüssow, B., Lewis, R. L., & Drenhaus, H. (2008). Processing polarity: How the ungrammatical intrudes on the grammatical. *Cognitive Science*, *32*, 685–712.

Vasishth, S., & Nicenboim, B. (2016). Statistical methods for linguistic research: Foundational ideas part i. *Language and Linguistics Compass*, *10*, 349–369.

Vermeulen, C. F. M. (1994). Incremental semantics for propositional texts. *Notre Dame Journal of Formal Logic*, *35*, 243–271.

Vermeulen, C. F. M. (1995). Merging without mystery: Variables in dynamic semantics. *Journal of Philosophical Logic*, *24*, 405–450.

Visser, A. (2002). The donkey and the monoid. *Journal of Logic, Language and Information*, *11*, 107–131.

Wagers, M. W., Lau, E. F., & Phillips, C. (2009). Agreement attraction in comprehension: Representations and processes. *Journal of Memory and Language*, *61*, 206–237.

Wagers, M. W., & Phillips, C. (2009). Multiple dependencies and the role of the grammar in real-time comprehension. *Journal of Linguistics*, *45*, 395–433.

Warren, T., White, S. J., & Reichle, E. D. (2009). Investigating the causes of wrap-up effects: Evidence from eye movements and E-Z reader. *Cognition*, *111*, 132–137.

Weaver, R. (2008). Parameters, predictions, and evidence computational modeling: A statistical view informed by ACT-R. *Cognitive Science, 32*.

West, R., Pyke, A., Rutledge-Taylor, M., & Lang, H. (2010). Interference and act-r: New evidence from the fan effect. In D. D. Salvucci & G. Gunzelmann (Eds.), *Proceedings of the 10th international conference on cognitive modeling* (pp. 211–216). Philadelphia, PA: Drexel University.

Wexler, K. (1978). A review of john r. anderson's language, memory, and thought. *Cognition*, *6*, 327–351.

Whaley, C. P. (1978). Word-nonword classification time. *Journal of Verbal Learning and Verbal Behavior*, *17*, 143–154.

Wickham, H. (2009). *ggplot2: elegant graphics for data analysis*. New York: Springer.