

CSU Studies in
Computational Linguistics

Representation and Inference for Natural Language

**A First Course in
Computational Semantics**

**Patrick Blackburn
Johan Bos**

Representation and Inference for Natural Language

CSLI Studies in Computational Linguistics

This series covers all areas of computational linguistics and language technology, with a special emphasis on work which has direct relevance to practical applications, makes formal advances, and incorporates insights into natural language processing from other fields, especially linguistics and psychology. Books in this series describe groundbreaking research or provide an accessible and up-to-date overview of an area for nonspecialists. Also included are works documenting freely available resources for language processing, such as software, grammars, dictionaries, and corpora.

Series edited by Ann Copestake

CSLI Studies in
Computational Linguistics

Representation and Inference for Natural Language

A First Course in Computational Semantics

Patrick Blackburn & Johan Bos

— — —



Center for the Study of
Language and Information
Stanford, California

Copyright © 2005
CSLI Publications
Center for the Study of Language and Information
Leland Stanford Junior University
Printed in the United States
09 08 07 06 05 1 2 3 4 5

Library of Congress Cataloging-in-Publication Data
Blackburn, Patrick, 1959–

Representation and inference for natural language : a first course in computational semantics / by Patrick Blackburn and Johan Bos.

p. cm. – (CSLI studies in computational linguistics)

Includes bibliographical references and index.

ISBN-13: 978-1-57586-495-2 (cloth cover : alk. paper)

ISBN-10: 1-57586-495-9 (cloth cover : alk. paper)

ISBN-13: 978-1-57586-496-9 (pbk. : alk. paper)

ISBN-10: 1-57586-496-7 (pbk. : alk. paper)

1. Semantics—Data processing. I. Bos, Johannes. II. Title. III. Series.

P325.5.D38B59 2005

401'.43'0285-dc22 2004029974

CIP

∞ The acid-free paper used in this book meets the minimum requirements of the American National Standard for Information Sciences—Permanence of Paper for Printed Library Materials, ANSI Z39.48-1984.

CSLI was founded in 1983 by researchers from Stanford University, SRI International, and Xerox PARC to further the research and development of integrated theories of language, information, and computation. CSLI headquarters and CSLI Publications are located on the campus of Stanford University.

CSLI Publications reports new developments in the study of language, information, and computation. Please visit our web site at
<http://cslipublications.stanford.edu/>

for comments on this and other titles, as well as for changes and corrections by the authors and publisher.

Contents

Preface vii

Introduction xi

1 First-Order Logic 1

- 1.1 First-Order Logic 1
- 1.2 Three Inference Tasks 19
- 1.3 A First-Order Model Checker 29
- 1.4 First-Order Logic and Natural Language 44

2 Lambda Calculus 55

- 2.1 Compositionality 55
- 2.2 Two Experiments 59
- 2.3 The Lambda Calculus 66
- 2.4 Implementing Lambda Calculus 73
- 2.5 Grammar Engineering 86

3 Underspecified Representations 105

- 3.1 Scope Ambiguities 105
- 3.2 Montague's Approach 109
- 3.3 Storage Methods 112
- 3.4 Hole Semantics 127

4	Propositional Inference	155
4.1	From Models to Proofs	155
4.2	Propositional Tableaus	158
4.3	Implementing Propositional Tableau	168
4.4	Propositional Resolution	174
4.5	Implementing Propositional Resolution	185
4.6	Theoretical Remarks	191
5	First-Order Inference	203
5.1	A First-Order Tableau System	204
5.2	Unification	209
5.3	Free-Variable Tableaus	215
5.4	Implementing Free-Variable Tableaus	219
5.5	First-Order Resolution	225
5.6	Implementing First-Order Resolution	231
5.7	Off-the-Shelf Theorem Provers	235
5.8	Model Building	242
6	Putting It All Together	259
6.1	Baby Curt	259
6.2	Rugrat Curt	264
6.3	Clever Curt	267
6.4	Sensitive Curt	272
6.5	Scrupulous Curt	277
6.6	Knowledgeable Curt	281
6.7	Helpful Curt	293
A	Running the Software – FAQ	309
B	Propositional Logic	311
C	Automated Reasoning for First-Order Logic	315
D	Notation	319
References		321
Author Index		335
Prolog Index		339
Subject Index		343

Preface

This book developed out of courses on computational semantics that the authors jointly taught at the Department of Computational Linguistics, University of the Saarland, Saarbrücken, Germany, in 1995 and 1998, and at ESSLLI'97, the *9th European Summer School in Logic, Language and Information*, Aix-en-Provence, France, in August 1997. When designing these courses, we found no single source containing all the material we wanted to present. At that time, the only notes exclusively devoted to computational semantics that we knew of were Cooper et al. (1993), probably the first systematic introduction to modern computational semantics. Like the present book, these notes are Prolog based, and cover some of the same ground, often using interestingly different tools and techniques. However we wanted to teach the subject in a way that emphasized inference, underspecification, and grammar engineering and architectural issues. By the end of the 1990s we had a first version of the book, influenced by Pereira and Shieber (1987) and Johnson and Kay (1990) for semantic construction, and Fitting (1996) for inference, which partially realised these goals.

The project then took on a life of its own: it expanded and grew in a variety of (often unexpected) directions. Both the programs and text were extensively rewritten, some parts several times. We first presented a mature, more-or-less stable, version of the newer material at ESSLLI'01, the *13th European Summer School for Logic, Language, and Information*, Helsinki, Finland, in August 2001. We then presented it in a more refined form at NASSLLI'02, the *2nd North American Summer School for Logic, Language, and Information*, Indiana University, Bloomington, Indiana, USA, in June 2002. And finally, many years after we started, we have ended up with the kind of introduction to computational semantics that we wanted all along. It has taken us a long time to get there, but the journey was a lot of fun. We hope that this

comes through, and that the book will be a useful introduction to the challenging and fascinating area known as computational semantics.

Acknowledgments

The path to the finished book has been long and winding, and we have accumulated many debts of gratitude along the way. First of all, we would like to thank Manfred Pinkal and our ex-colleagues at the Department of Computational Linguistics, University of the Saarland, Saarbrücken, Germany; the department was a stimulating environment for working on computational semantics, and the ideal place to start writing this book. Special thanks to David Milward, who during his stay at Saarbrücken developed the test suite for β -conversion, and allowed us to make use of it. We would also like to thank all our ex-students from the University of the Saarland, and in particular Judith Baur, Aljoscha Burchardt, Gerd Fliedner, Malte Gabsdil, Kristina Striegnitz, Stefan Thater, and Stephan Walter, for their enthusiasm and support.

Conversations with Harry Bunt, Paul Dekker, Andreas Franke, Claire Gardent, Hans Kamp, Martin Kay, Michael Kohlhase, Alexander Koller, Karsten Konrad, Alex Lascarides, Christof Monz, Reinhard Muskens, Malvina Nissim, Maarten de Rijke, and Henk Zeevat were helpful in clarifying our goals. We're grateful to Stephen Anthony, David Beaver, Mary Dalrymple, Stina Ericsson, Norbert Fuchs, Björn Gambäck, Durk Gardenier, Michel Généreux, John Kirk, Ewan Klein, Emiel Krahmer, Torbjörn Lager, Shalom Lappin, Staffan Larsson, Stefan Müller, Ian Pratt-Hartmann, Rob van der Sandt, Frank Schilder, and David Schlangen for comments on early drafts. Special thanks go to Carlos Areces and Kristina Striegnitz for their editorial help, to Eric Kow for Perl hints, and to Sébastien Hinderer for proof-reading.

Many members of the automated reasoning community expressed interest in this project and gave us various kinds of support. In particular, we are grateful to William McCune (the author of Otter and Mace) for his timely response to our questions, to Hans de Nivelle (the author of Bliksem) for helping us with our first experiments in off-the-shelf theorem proving, and to Koen Claessen and Niklas Sörensson (the authors of Paradox) for discussion and advice on model building. We'd also like to thank Peter Baumgartner, Rajeev Goré, Andrew Slater, Geoff Sutcliffe, and Toby Walsh for their helpful responses to our questions.

Finally, we are grateful to Robin Cooper and Bonnie Webber, who both taught courses based on the previous drafts of this book, and provided us with feedback on what worked and what didn't.

Patrick Blackburn
Johan Bos
December 2004

Introduction

This book introduces a number of fundamental techniques for computing semantic representations for fragments of natural language and performing inference with the result. Both the underlying theory and their implementation in Prolog are discussed. We believe that the reader who masters these techniques will be in a good position to appreciate (and critically assess) ongoing developments in computational semantics.

Computational semantics is a relatively new subject, and trying to define such a lively area (if indeed it is a single area) seems premature, even counterproductive. However, in this book we take “semantics” to mean “formal semantics” (that is, the business of giving model-theoretic interpretations to fragments of natural language, usually with the help of some intermediate level of logical representation) and “computational semantics” to be the business of using a computer to actually build such representations (*semantic construction*) and reason with the result (*inference*). Thus this book introduces techniques for tackling the following two questions:

1. *How can we automate the process of associating semantic representations with expressions of natural language?*
2. *How can we use logical representations of natural language expressions to automate the process of drawing inferences?*

In the remainder of the Introduction we'll briefly sketch how we are going to tackle these questions, explain where we think computational semantics belongs on the intellectual landscape, suggest how best to make use of the text and the associated software, and (in the Notes at the very end) give a brief historical account of the origins of computational semantics. Some of the discussion that follows may not be completely accessible at this stage, especially if you have never encountered semantics or logic before. But if this is the case, don't worry. Concentrate on the straightforward parts (such as the chapter-by-chapter

outline and the advice on using this book), and then go straight on to the chapters that follow. You can return to the Introduction later; by then you will be in a better position to assess the perspective on computational semantics that has guided the writing of this book.

Representation

As regards semantic construction, this book is fairly orthodox (though see below for some caveats). We first introduce the reader to first-order logic, and then show how various kinds of sentences of natural language can be systematically translated into this formalism. For example, the techniques we discuss (and the software we implement) will enable us to take a sentence like

Every boxer loves Mia.

as input and return the following formula of first-order logic as output:

$\forall x(\text{BOXER}(x) \rightarrow \text{LOVE}(x, \text{MIA}))$.

The technical tool we shall use to drive this translation process is the lambda calculus. We motivate and introduce the lambda calculus from a computational perspective, and show in detail how it can be incorporated into an architecture for building semantic representations.

To put it another way, it's not inaccurate to claim that roughly half of this book is devoted to what is known as Montague semantics. Yes, it's true that we don't discuss a lot of topics that would ordinarily be taught in a first course on Montague semantics (for example, we don't discuss intensional semantics). But in our view Richard Montague was not merely the father of *formal* semantics (or *model-theoretic* semantics, as it is often called), he was also the father of *computational* semantics. Richard Montague made many pioneering contributions to the study of semantics, but in our view the most important was the conceptual leap that opened the door to genuine computational semantics: he showed that the process of constructing semantic representations for expressions of natural language could be formulated *algorithmically*. Many philosophers before Montague (and many philosophers since) have used (various kinds of) logic to throw light on (various aspects of) natural language. But before Montague's work, such comparisons were essentially analogies. Montague showed how to link logic and language in a *systematic* way, and it is this aspect of his work that lies at the heart of this book.

Inference

But this book is not just about representation, it is also about inference. Now, inference is a vast topic, and it is difficult to be precise about what

is (and is not) covered by this term. But, roughly speaking, we view inference as the process of making *implicit* information *explicit*. To keep this book to a manageable size we have focused on one particular aspect of this process, namely the making of logical inferences. We have done so by formulating three inference tasks—the querying task, the consistency checking task, and the informativity checking task—and have looked at inference in natural language through the lens they provide. For example, it is intuitively clear that the discourse

Every boxer loves Mia. Butch is a boxer. Butch does not love Mia.

is incoherent. But why is it incoherent? As we shall show in Chapter 1, the consistency checking task gives us an important theoretical handle on this type of incoherency. Moreover in the second half of the book we shall create a computational architecture that makes use of sophisticated automated reasoning tools to give us a useful (partial) grasp on consistency checking (partial, because of the undecidability of first-order logic). We conclude the book by showing how our semantic construction software (part of Richard Montague’s legacy) and our inference architecture (the legacy of John Alan Robinson and the other pioneers of automated reasoning) can be integrated.

Comments and caveats

Well, that’s where we heading—but before moving on, two remarks should be made. First, above we talked about semantic construction and inference as if they were independent, but in fact they’re not. Indeed, how semantic construction and inference are interleaved is an extremely deep and difficult problem, and we certainly don’t claim to have solved it in this book. Nonetheless, we *do* believe that the issues this problem raises need to be explored computationally, and that architectures of the type discussed here—that is, architectures which draw on both semantic construction and inference modules—will become fundamental research tools.

Second, the working definition of computational semantics given above isn’t quite as innocent as it looks. Many formal semanticists claim that intermediate levels of logical representation are essentially redundant. Richard Montague himself, in his paper “English as a Formal Language”, showed how a small fragment of English could be model-theoretically interpreted without first translating it into an intermediate logical representation. Moreover, in his paper “Universal Grammar”, he showed that (given certain assumptions) it is always possible to interpret fragments of natural language in this way.

Nonetheless, we feel justified in emphasising the role of intermedi-

ate logical representations. For a start, the move to a *computational* perspective on formal semantics certainly increases the *practical* importance of the representation level. Logical representations—that is, formulas of a logical language—encapsulate meaning in a clean and compact way. They make it possible to use well understood proof systems to perform inference, and we shall learn how to exploit this possibility. Models may be the heart of traditional formal semantics, but representations are central to its computational cousin.

Moreover—and this is something that we hope becomes increasingly clear in the course of the book—we feel that the computational perspective vividly brings out the *theoretical* importance of representations. The success of Discourse Representation Theory (DRT) over the past two decades, the explosion of interest in underspecification (which we explore in Chapter 3) and the exploration of glue languages such as linear logic to drive the process of semantic construction all bear witness to an important lesson: semanticists ignore representations at their peril. Representations are well-defined mathematical entities that (among other things) can be manipulated computationally, explored geometrically, and specified indirectly with the aid of constraints. The fact that representations are theoretically eliminable does not mean they should not be taken seriously.

But why computational semantics?

Our discussion so far has taken it for granted that computational semantics is an interesting subject, one well worth studying. But it is probably a good idea to be explicit about why we think this is so.

We believe that the tools and techniques of computational semantics are going to play an increasingly important role in the development of semantics. Now, semantics has made enormous strides since the pioneering work of Richard Montague in the late 1960s and early 1970s. Nonetheless, we believe that its further development is likely to become increasingly reliant on the use of computational tools. Modern formal semantics is still a paper-and-pencil enterprise: semanticists typically examine in detail a topic that interests them (for example, the semantics of tense, or aspect, or focus, or generalized quantifiers), abstract away from other semantic phenomena, and analyse the chosen phenomenon in detail. This “work narrow, but deep” methodology has undeniably served semanticists well, and has lead to important insights about many semantic phenomena. Nonetheless, we don’t believe that it can unveil all that needs to be known about natural language semantics, and we don’t think it is at all suitable for research that straddles the (fuzzy and permeable) border between semantics and pragmatics

(the study of how language is actually used). Rather, we believe that in the coming years it will become increasingly important to study the *interaction* of various semantic (and pragmatic) phenomena, and to model, as precisely as possible, the role played by inference.

Now, it's easy to say that this is what should be done—actually doing it, however, is difficult. Indeed, as long as semanticists rely purely on pencil-and-paper methods, it is hard to see how this style of research can produce detailed results. In our view, computational modelling is required. That is, we believe that flexible computational architectures which make it possible to experiment with semantic representations, semantic construction strategies, and inference, must be designed and implemented. To give an analogy, nowadays it is possible to use sophisticated graphics programs when studying large molecules (such as proteins). Such programs make it possible to grasp the three-dimensional structure of the molecule, and hence to think at a more abstract level about their properties and the reactions they can enter into. Semanticists need analogous tools. The ability to formulate detailed semantic theories, and to compute rapidly what they predict, could open up a new phase of research in semantics. It could also revolutionise the teaching of semantics.

Two comments. First, note that we're *not* claiming that semanticists should abandon their traditional “work narrow, but deep” strategy; this style of research is indispensable. Rather, we are suggesting that it should be augmented by a computer-aided “work broad, and model the interactions” approach. Second—before anyone gets their hopes up prematurely—we would like to emphasise that the software discussed in this book does *not* constitute a genuine research architecture. The design and implementation of the type of “Semantic Workbenches” we have in mind is a serious task, one far beyond the scope of an introduction to computational semantics. But we certainly do hope that the software provided here will inspire readers to design and implement more ambitious systems.

Computational semantics and computational linguistics

So that's our answer to the question “Why computational semantics?”. But although this answer might well interest (or enrage!) formal semanticists, there is another group of researchers who may find it unconvincing. Which group? Computational linguists. We wouldn't be surprised to learn that some computational linguists are dubious about our aims and methods. Where is the statistics? Where is the use of corpora? Why analyse sentences in such depth? Does inference really require the use of such powerful formalisms as first-order logic? Indeed, does inference

really require logic at all? We would like to make two brief remarks here, for we certainly *do* view the techniques taught in this book as an integral part of computational linguistics.

Firstly, what we teach in this book is certainly *compatible* with statistically-oriented approaches to computational linguistics. In essence, we provide some fundamental semantic construction tools (the use of lambda calculus, coupled with methods for coping with scope ambiguities) and inference tools (an architecture for using theorem provers and model builders in parallel) and put them to work. In this book these components are used via a simple Definite Clause Grammar (DCG) architecture, but they certainly don't have to be. They can be—and have been—combined with such tools as speech recognisers and wide coverage statistical parsers to build more interesting systems.

Secondly, we believe that the methods taught in this book are not merely compatible, but might actually turn out to be *useful* to statistically-oriented work. This book was born from the conviction that formal semantics has given rise to the deepest insights into the semantics of natural language that we currently have—and an accompanying belief that a computational perspective is needed to fully unleash their potential. So we find it natural (and important) to look for points of contact with mainstream computational linguistics. For example, many computational linguists want to extend the statistical revolution of the late 1980s and early 1990s (which transformed such areas as speech processing and parsing) to the semantic domain. We don't see any conflict between this goal and the ideas explored in this book. Indeed, we believe that techniques from computational semantics may be important in exploring statistical approaches to semantics: detailed training corpora will be needed, and the techniques of computational semantics may be helpful in producing the requisite “gold standard” material.

It is true that these remarks are somewhat speculative. Nonetheless, in our view the low cost of massive computational power, the ubiquitous presence of the internet, the sophistication of current automated reasoning tools, and the superb linguistic resources now so widely available, all add up to a new era in semantic research. Understanding how natural language works is one of the toughest (and most interesting) problems there is. It's time to get all hands on deck.

Outline

The key aim of this book is to develop a working toolkit for computational semantics. We develop this toolkit as follows:

Chapter 1. First-Order Logic. We begin by introducing the syntax and semantics of first-order logic, the semantic representation language used in this book. We then define and discuss the three inference tasks we are interested in: the querying task, the consistency checking task, and the informativity checking task. Following this, we implement a first-order model checker in Prolog. A model checker is a program that checks whether a formula is true in a given model, or to put it another way, it is a piece of software that performs the querying task.

Chapter 2. Lambda Calculus. Here we start studying semantic construction. We outline the methodology underlying our work (namely, compositionality) and motivate our use of DCGs (Definite Clause Grammars). We then write two rather naive programs that build semantic representations for a very small fragment of English. These experiments lead us to the lambda calculus, the tool that drives this book’s approach to semantic construction. We implement β -conversion, the computational core of the lambda calculus, and then integrate it into the grammatical architecture that will be used throughout the book.

Chapter 3. Underspecified Representations. Here we investigate a fundamental problem for computational semantics: scope ambiguities. These are semantic ambiguities that can arise in syntactically unambiguous expressions, and they pose a problem for compositional approaches to semantic construction. We illustrate the problem, and present four (increasingly more sophisticated) solutions: Montague’s use of quantifier raising, Cooper storage, Keller storage, and hole semantics. We integrate storage and hole semantics into our grammar architecture.

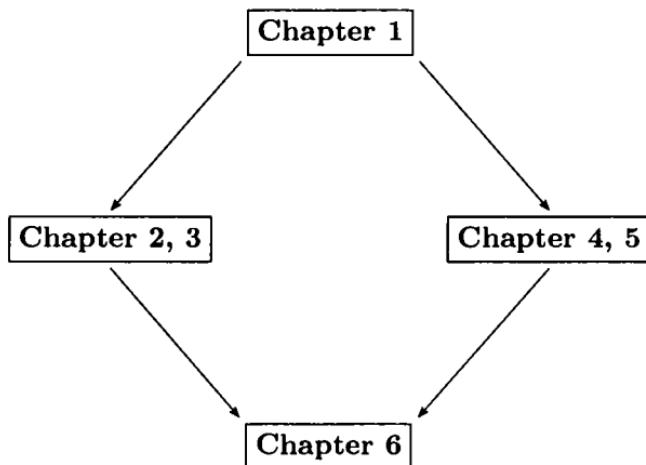
Chapter 4. Propositional Inference. Here we turn to the second major theme of the book: inference. Our approach to inference will be based on first-order theorem proving and model building, and in this chapter we lay the conceptual foundations for the work of Chapter 5 in the simpler setting of propositional logic. We introduce a signed tableau and a resolution system for propositional calculus, implement both in Prolog, and conclude with a discussion of a number of theoretical issues.

Chapter 5. First-Order Inference. In this chapter we explore inference in the setting of full first-order logic—and, as we swiftly learn, that’s a computationally demanding setting. We extend the propositional and resolution theorem provers to deal with first-order logic, but we also see that home-brewed theorem provers simply don’t have the muscle for tackling the consistency and informativity checking tasks in a serious way. So we change tack: instead of building our own, we show the reader how to integrate the sophisticated inference tools created by the automated reasoning community into an inference architec-

ture for computational semantics. By the end of the chapter, we have constructed an architecture for consistency and informativity checking that works by calling (sophisticated) theorem provers and model builders in parallel.

Chapter 6. Putting It All Together. In this chapter we bring together the software developed in earlier chapters and develop a series of programs bearing the name Curt (which stands for Clever Use of Reasoning Tools). Curt starts out as a real baby—it can build semantic representations and detect scope ambiguities, but alas, nothing more. No matter! By making use of our inference architecture, (and the model checker developed in Chapter 1), we are able, step-by-step, to extend Curt’s abilities so that it can handle consistency checking, informativity checking, eliminate logically equivalent readings, incorporate background knowledge, and answer simple questions.

Here’s how the chapters fit together:



That is, Chapter 1 provides the foundation for everything that follows. Chapter 2 and 3 are the representation track of the book, and should be read together. Similarly, Chapters 4 and 5 are the inference track of the book, and again, these two chapters should be read together. The representation and inference tracks are independent of each other, and can be read in either order. Finally, Chapter 6 draws on all that has gone before.

Each chapter concludes with Notes that list references and briefly discuss more advanced topics. Four appendices at the end of the book provide background information.

Using this book

We have tried to make this book relatively self-contained. In fact, there is only one real prerequisite that is *not* introduced here, namely the Prolog programming language. To gain the most out of the computational side of this book, you will need to have some knowledge of this language, and access to a Prolog interpreter. Many good books on Prolog are available, but we'd like to draw your attention to *Learn Prolog Now!*, by Patrick Blackburn, Johan Bos, and Kristina Striegnitz. Written in parallel with the present book, it contains everything needed to understand the code presented here. *Learn Prolog Now!* is available free on the internet at

<http://www.learnprolognow.org/>

For the Prolog interpreters that handle our software, see Appendix A.

Apart from Prolog, we believe that this book covers everything most readers will need. Indeed, we believe that even readers with fairly modest backgrounds in linguistics and logic should be able to follow the discussion. We have taught this material both to linguistics students (with rather weak logical and computational backgrounds) and to computer science students (with stronger logical backgrounds, but no prior exposure to linguistics). Our experience suggests that as long as the instructor is sensitive to the type of background knowledge the students have, it is possible to successfully teach this material. Moreover, the Notes at the end of each chapter provide many references for supplementary reading. So if you are using this book for self study and get stuck at some point, try looking at these.

But there is one point we would like to strongly emphasise to all our readers: *please take the computational component seriously*. Yes, it is certainly possible to read this book with only half an eye on the computational developments. Moreover, we'll also admit that even if you're allergic to computers and computer programming, but want to learn about semantics in a way that emphasises inference, you can use this book for that purpose without worrying too much about the Prolog programs.

Still, while you *can* read the book this way, we feel it's a bit of a shame to do so—after all, this is a book on *computational* semantics, and the reader who does not make the effort to get to grips with the computational issues it discusses is getting (at most) fifty percent of what this book has to offer. Now, there's no denying that for some readers the computational side will be the hard part—readers with weak computational backgrounds will have to put in some extra work (apart from anything else, they'll have to learn something about Prolog). But

the effort is well worth making. Thinking about problems computationally often reveals new perspectives on old ideas. For example, our account of the lambda calculus in Chapter 2 makes no appeal to types, function valued functions, or higher-order logic—rather, lambda calculus is presented as a beautiful piece of data-abstraction that emerges naturally from a declarative analysis of semantic construction. To give another example, the various techniques developed for handling scope ambiguities (from Montague’s method, through storage methods, to modern underspecification methods) display a conceptually clear evolutionary line when viewed computationally.

Thus the computational side of this book should not be viewed as an optional extra. Indeed, we might say that the ideal reader of this book is someone who treats the text as *documentation*. Such a reader might not wish to understand all the details of the programs provided, but he or she would certainly want to play with them, perhaps by extending the grammars, by experimenting with novel semantic constructions, by applying the inference architecture to novel tasks, or by applying these ideas to other languages. In short, don’t think of this as a book. Think of it as a tool-kit for exploring computational semantics. And then put it to work.

Web support

We have set up webpage for this book. The URL is

<http://www.blackburnbos.org/>

There you will find our Prolog programs, pointers to other useful software (such as Prolog interpreters, theorem provers and model builders), and any corrections to the text that need to be made. From time to time we will place material there that extends the present text. For example, Blackburn and Bos (2003), which can be read as a sort of ‘alternative introduction’ to this book, can be found on the website.

Notes

Modern logic and semantics stem from the work of Gottlob Frege (1848–1925). On the logical side, Frege introduced the use of variable binding quantifiers (we shall see such quantifiers in the following chapter when we introduce first-order logic), and on the semantic side he introduced a number of concepts (such as the distinction between the “sense” and the “reference” of an expression) that are still important today. Readers wanting a taste of Frege’s work could try Frege (1892). Translations of this paper have been widely anthologised (for example you can find it in Martinich (1996) under the title “On Sense and Nominatum”).

Frege's logical work and his ideas on the foundations of mathematics became increasingly influential from roughly 1900 onwards, and their influence endures till this day. But his pioneering work on semantics took longer to bear fruit. While some important early work was done (for example, by the philosopher Bertrand Russell) the next big steps were not taken till the middle of the 20th century.

With the benefit of hindsight we can see that this delay was not accidental, for a key idea was missing: the notion of *interpretation in a model*. As we shall see in the following chapter, nowadays logic is conceived of as having *two* main components. First, there is some kind of formal logical language (for example, a language of first-order logic that makes use of such symbols such as \forall , \exists , \wedge , \rightarrow and so on). But in addition, crucial use is made of what are known as models, simple mathematical structures that act as pictures of the world (or at least, that part of the world we happen to be interested in for our application). At the heart of much modern logic is the idea of giving a precise mathematical definition of how formal logical languages are linked with models—or to put it more semantically, to specify how formal languages are to be *interpreted* in models (such a definition is usually called a *satisfaction definition*). One of the fundamental facts that any theory of semantics is going to have to get to grips with is that natural languages (like English) can be used to talk about the world around us (for example, English speakers use the word “woman” for adult human females). When we link a logical language with a model via a satisfaction definition we gain a precise mathematical handle on the language-world relationship.

In 1933 Alfred Tarski (1902–1983) gave the first fully explicit satisfaction definition (see Tarski (1935) for a German translation of the Polish original). Its importance was quickly realised by both philosophers and mathematicians, and stimulated serious work on semantics. In particular, Rudolf Carnap (1891–1979), produced an important body of work. For example, his book “Meaning and Necessity” (Carnap, 1947) is still well worth looking at: among other things it discusses the semantics of belief and necessity (two key examples of intensionality in natural language), examines the notion of “meaning postulate” in detail (such postulates would nowadays be thought of as axioms encoding world knowledge or lexical knowledge), and speculates on the possibility of extending Tarski's model-based approach to semantics to handle pragmatics too.

But it was with the work of Richard Montague (1930–1971) that semantics finally came of age. Although Montague only wrote a handful of papers on the subject, they were destined to shape subsequent research.

Three of his papers, “Pragmatics” (Montague, 1968), “On the Nature of Certain Philosophical Entities” (Montague, 1969), and “Pragmatics and Intensional Logic” (Montague, 1970b) are technically sophisticated developments of the program initiated by Carnap. (And as the titles of two of these papers indicate, Montague was able to extend the model-based approach to semantics to cover a pragmatic phenomenon, namely indexicality.) But it is his last three papers “English as a Formal Language” (Montague, 1970a), “Universal Grammar” (Montague, 1970c), and “The Proper Treatment of Quantification in Ordinary English” (Montague, 1973) that sound a genuinely new note, for it is here that Montague unveils what has become known as the *method of fragments*. What does this mean? Simply that in these papers Montague defined grammars for small portions of English, and showed how the sentences generated by these grammars could be interpreted in models. In “English as a Formal Language” Montague interpreted the English fragment directly (that is, without first translating into an intermediate logical representation) whereas in “Universal Grammar” and “The Proper Treatment of Quantification in Ordinary English” he first translated into higher-order logic. But this difference is far less important than what is common to them all: *in all three papers the interpretation process required is completely explicit*. In essence, all three papers give interpretation algorithms for fragments of English. To be sure, nobody would claim that the interpretations offered by Montague cover all aspects of what we might want to call meaning, and it is also true that the fragments given by Montague were rather small. But such considerations should not blind us to the fact that in these papers something very important has taken place: we see the first glimpse of a possible mechanism underlying natural language semantics.

It is not possible here to give a detailed account of developments in semantics since the work of Montague. Perhaps the most important development, and certainly the one of most relevance to computational semantics, was the birth of Discourse Representation Theory (see Heim (1982), Kamp (1984), and Kamp and Reyle (1993)). DRT (as it is usually called) has enabled Montague’s program to be extended from the level of sentences to entire discourses, and has also enabled semantics to make further inroads on the domain of pragmatics. But DRT and many other interesting developments are beyond the scope of the book, so we refer the reader to Partee (1997a) and Partee (1997b), two useful discussions of Montague’s work and the research it inspired.

We now turn our attention from the development of formal semantics to research conducted in two computational disciplines, namely computational linguistics and Artificial Intelligence (AI). The formal

semantic tradition has been the primary source of much that we teach in this book, but the way we view this material has been indelibly marked by ideas from computational linguistics and AI. So let's round out the picture with a quick look at these traditions.

One of the basic themes of this book is the usefulness of logic as a tool for representation and inference. But this is not a new idea—it's a mainstay of classical AI. It's interesting to look through some of the more influential AI textbooks, say Nilsson (1980), Winston (1981), Charniak and McDermott (1985), Rich and Knight (1990), and Russell and Norvig (1995). All take first-order logic as a fundamental framework for representation and inference (and not merely for natural language tasks either), and discuss inference procedures for first-order logic (notably resolution) in varying degrees of detail. Weaker formalisms (such as semantic nets) are sometimes used and (especially in the later texts) the point is explicitly made that such formalisms are essentially fragments of first-order logic with good computational properties (for example, semantic nets are a forerunner of what are nowadays known as description logics; see Baader et al. (2003)).

In short, many of the fundamental ideas on logic and inference underlying these texts are close to those taught here. Indeed, much what divides this book from these earlier introductions (apart from the obvious fact that the texts just mentioned cover a wide range of topics in AI, such as learning, planning, and image recognition, whereas ours focuses exclusively on computational semantics) is simply due to the explosive pace of contemporary research. When we talk about semantic construction, we can draw on ideas (such as constraint-based underspecification) that hadn't been developed when these books were written. And when we advocate the use of first-order logic and theorem proving, we can point the reader to provers (and indeed, newer tools such as model builders) whose performance dwarfs anything available earlier. Moreover, in this book we emphasise the importance of developing architectures by finding the best available components and linking them. This style of development wasn't so practical in the 1980s and early 1990s; nowadays, given the ubiquity of the internet, it seems likely to become the default option.

So there is a broad similarity of aims and methods between what we teach in this book and much that is done in computational linguistics and AI—and given the fundamental role played by logic in these disciplines, we don't find this surprising. If anything, what *is* surprising is how long it has taken for a real alliance between computational linguistics and formal semantics to be forged. For until the 1990s there seem to have been few systematic attempts by researchers in computational

linguistics (and AI) to make contact with ideas from formal semantics (and attempts by formal semanticists to make serious contact with ideas from computational linguistics and AI seem to have been even thinner on the ground). That said, there *are* some interesting examples of earlier work in computational linguistics that falls (or almost falls) under our working definition of computational semantics. Let's look at a few.

Hobbs and Rosenschein (1978), a paper entitled "Making Computational Sense of Montague's Intensional Logic", suggests that Montague semantics is best thought of in terms of procedural semantics rather than model-theoretic semantics; to add substance to this idea, the paper contains a translation of Montague's intensional logic into the functional programming language Lisp. In Schubert and Pelletier (1982), motivated by the need to carry out inference, the authors define a simple translation from a fragment of English (specified using a context-free grammar) into what they call conventional logic. In Landsbergen (1982), on the other hand, the motivation is machine translation. Montague's logic is used as an interlingua: the source language is translated into it, and the resulting logical expression is used to help build a sentence in the target language. In Main and Benson (1983), ideas from Montague semantics are used in a question answering system. Also from this period is Gunji (1981), a PhD thesis which is not only a pioneering contribution to computational semantics, but to computational pragmatics as well. Gunji, realising that a database in a computer can be regarded as a model, defines a system in which incoming sentences are translated into Montague-style logic. While the database is essentially read-only memory as far as his semantic procedures are caused (that is, logical formulas are evaluated in the database without altering it, much as in a standard database query) there are also pragmatic procedures that can modify the model (or as Gunji puts it, induce context changes) which may well affect the semantic evaluations of later sentences.

It is also interesting to look through anthologies on computational linguistics for evidence of interest in computational semantics. Actually, one such collection "Computational Semantics. An Introduction to Artificial Intelligence and Natural Language Comprehension" (Charniak and Wilks, 1976) contains in its title the earliest usage of the term "computational semantics" that we know of. A collection of classic papers that is well worth consulting is Grosz et al. (1986), and the more recent collection Rosner and Johnson (1992) also contains much of relevance. Finally, mention must be made of the collected papers of Robert Moore, a researcher who has made many contributions to computational semantics, ranging from work on semantic construction to

intensional semantics (see Moore (1995)).

But it was sometime during the 1990s that computational semantics really began to acquire its own identity. For a start, more solid bridges between formal semantics and computational linguistics began to appear. For example, though the *Handbook of Contemporary Semantic Theory* (Lappin, 1997) for the most part contains articles on traditional themes in formal semantics, it also contains an article devoted to the use of attribute-value structure unification (a standard technique in computational linguistics) to build semantic representations (see Nerbonne (1997)). And in the other direction, Jurafsky and Martin (2000), which has established itself as the standard introduction to speech and language processing, teaches an approach to semantic construction that is based on first-order logic and lambda calculus (see in particular Chapters 14 and 15) and refers to the work of Richard Montague and other researchers in formal semantics.

But perhaps the most important coming-of-age landmark was the founding of the International Workshop on Computational Semantics (IWCS) by Harry Bunt. The first was in 1995, and the workshop (which is held in Tilburg in the Netherlands) has taken place every two years ever since. It is the main meeting place for the computational semantics community, and selected proceedings of two of these meetings are available in book form (see Bunt and Muskens (1999) and Bunt et al. (2001)). A more specialised workshop, Inference in Computational Semantics (ICoS), was held in Amsterdam, The Netherlands in 1999, and since then ICoS has been held in Schloss Dagstuhl, Germany (in 2000), in Siena, Italy (in 2001), and in Nancy, France (in 2003). Selected proceedings of the first three meetings are available as special journal issues (see Monz and De Rijke (2000), Bos and Kohlhase (2003) and Kohlhase (2004)). Finally, in 1999 the Association for Computational Linguistics approved the creation of SIGSEM, a Special Interest Group in Computational Semantics. See

<http://www.aclweb.org/sigsem>

for further information.

First-Order Logic

First-order logic is the formalism used in this book to represent the meaning of natural language sentences and to carry out various inference tasks. In this chapter we introduce first-order logic from a model-theoretic (that is, semantic) perspective, and write a Prolog program for handling the simplest of the three inference tasks we shall discuss, the querying task.

In more detail, this is what we'll do. First, we define the syntax and semantics of first-order logic. We pay particular attention to the intuitions and technicalities that lie behind the satisfaction definition, a mathematically precise specification of how first-order languages are to be interpreted in models. We then introduce the three inference tasks we are interested in: the querying task, the consistency checking task, and the informativity checking task. All three tasks are defined model-theoretically. Following this, we write a first-order model checker. This is a tool for handling the querying task: the model checker takes as input a first-order formula and a first-order model, and checks whether the formula is satisfied in the model. By the time we've done all that, the reader will have a fairly clear idea of what first-order logic is, and it becomes profitable to consider more general issues. So, to close the chapter, we discuss the strengths and weaknesses of first-order logic as a tool for computational semantics.

1.1 First-Order Logic

In this section we discuss the syntax and semantics of first-order logic. That is, we introduce *vocabularies*, *first-order models* and *first-order languages*, and tie these concepts together via the crucial *satisfaction definition*, which spells out how first-order languages are to be interpreted in models. We then discuss three extensions of the basic formalism: *function symbols*, *equality*, and *sorted first-order logic*.

Vocabularies

The main goal of this section is to define how first-order formulas (that is, certain kinds of descriptions) are evaluated in first-order models (that is, mathematical idealisations of situations). Simplifying somewhat (we'll be more precise later), the purpose of the evaluation process is to tell us whether a description is true or false in a given situation.

We shall soon be able to do this—but we need to exercise a little care. Intuitively, it doesn't make much sense to ask whether or not an arbitrary description is true in an arbitrary situation. Some descriptions and situations simply don't belong together. For example, suppose we are given a formula (that is, a description) from a first-order language intended for talking about the various relationships and properties (such as *loving*, *being a robber*, and *being a customer*) that hold of and between Mia, Honey Bunny, Vincent, and Yolanda. If we are then given a model (that is, a situation) which records information about something completely different (for example, which household cleaning products are best at getting rid of particularly nasty stains) then it doesn't really make much sense to evaluate this particular formula in that particular model. *Vocabularies* (or *signatures* as they are sometimes called) allow us to avoid such problems; they tell us which first-order languages and models belong together.

Here is our first vocabulary:

```
{ (LOVE,2),
  (CUSTOMER,1),
  (ROBBER,1),
  (MIA,0),
  (VINCENT,0),
  (HONEY-BUNNY,0),
  (YOLANDA,0) }.
```

Intuitively, this vocabulary is telling us two important things: the topic of conversation, and the language the conversation is going to be conducted in. Let's spell this out a little.

First, the vocabulary tells us *what* we're going to be talking about. In the present case, we're going to be talking about the *love* relation. The 2 indicates that we think of loving as a *binary relation* (a *relation of arity 2*). That is, we view loving as a relation that can hold between two individuals. We are also going to be talking about *being a customer* and *being a robber*. The 1s indicate that we think of these as unary relations (or *relations of arity 1*). To put it another way, we view these as properties that can hold of single individuals. In addition to these relations and properties, we're also going to be talking about four specific

individuals, namely *Mia*, *Vincent*, *Honey Bunny*, and *Yolanda*; the 0s indicate that these are individuals and not properties or relations.

Second, the vocabulary also tells us *how* we are going to talk about these things. In this case it tells us that we will be using a symbol **LOVE** for talking about the love relation, the symbols **CUSTOMER** and **ROBBER** for talking about customers and robbers, and four constant symbols (or names), namely **MIA**, **VINCENT**, **HONEY-BUNNY**, and **YOLANDA** for referring to *Mia*, *Vincent*, *Honey Bunny*, and *Yolanda*, respectively.

Incidentally, note that in this vocabulary there are no examples of a symbol being used in two different ways. For example, there is no symbol (say **FLUB**) being used both to refer to individuals and for talking about a binary relation. And indeed, the usual convention in first-order logic is *not* to allow symbols to be used in multiple ways: the same symbol is never used to talk about relations of different arity, or to talk about relations and refer to individuals. Prolog programmers, of course, will be used to the opposite convention: when writing Prolog programs it's not at all unusual (and indeed, it can be very useful) to use the same symbol as (say) both a two place predicate and as a three place predicate.

Summing up, a vocabulary gives us all the information needed to define the class of models of interest (that is, the kinds of situations we want to describe) and the relevant first-order language (that is, the kinds of descriptions we can use). So let's now look at what first-order models and languages actually are.

Exercise 1.1.1 Consider the following situation: *Vincent* is relaxed. The gun rests on the back of the seat, pointing at *Marvin*. *Jules* is driving. *Marvin* is tense. Devise a vocabulary suitable for talking about this situation. Give the vocabulary in the set-theoretic notation used in the text.

Exercise 1.1.2 Devise a simple Prolog notation for representing vocabularies (for example, take our set-theoretic notation and use Prolog lists in place of the curly-brackets and ordered pairs). Write a program which checks that something written in your notation really is a first-order vocabulary. For example, it should check that each symbol is associated with a number giving its arity, and that no symbol is used in two different ways.

First-Order Models

Suppose we've fixed some vocabulary. What should a first-order *model* for this vocabulary be?

Our previous discussion has pretty much given the answer. Intuitively, a model is a situation. That is, it is a *semantic* entity: it contains the kinds of things we want to talk about. Thus a model for a

given vocabulary gives us two pieces of information. First, it tells us which collection of entities we are talking about; this collection is usually called the *domain* of the model, or D for short. Second, for each symbol in the vocabulary, it gives us an appropriate semantic value, built from the items in D . This task is carried out by a function F which specifies, for each symbol in the vocabulary, an appropriate semantic value; we call such functions *interpretation functions*. Thus, in set-theoretic terms, a model M is an ordered pair (D, F) consisting of a non-empty domain D and an interpretation function F specifying semantic values in D .

What are appropriate semantic values? There's no mystery here. As constants are essentially the analogs in first-order logic of ordinary names (that is, we will use constants to pick out individuals) each constant should be interpreted as an element of D . (That is, for each constant symbol c in the vocabulary, $F(c) \in D$.) As n -place relation symbols are intended to denote n -place relations, each n -place relation symbol R should be interpreted as an n -place relation on D . (To put it set-theoretically, $F(R)$ should be a set of n -tuples of elements of D .)

Let's consider an example. We shall define a simple model for the vocabulary given above. Let D be $\{d_1, d_2, d_3, d_4\}$. That is, this four element set is the domain of our little model.

Next, we must specify an interpretation function F . Here's one:

$$\begin{aligned} F(\text{MIA}) &= d_1 \\ F(\text{HONEY-BUNNY}) &= d_2 \\ F(\text{VINCENT}) &= d_3 \\ F(\text{YOLANDA}) &= d_4 \\ F(\text{CUSTOMER}) &= \{d_1, d_3\} \\ F(\text{ROBBER}) &= \{d_2, d_4\} \\ F(\text{LOVE}) &= \{(d_4, d_2), (d_3, d_1)\}. \end{aligned}$$

Note that every symbol in the vocabulary does indeed have an appropriate semantic value: the four names pick out individuals, the two arity 1 symbols pick out subsets of D (that is, properties, or 1-place relations on D) and the arity 2 symbol picks out a 2-place relation on D . In this model d_1 is Mia, d_2 is Honey Bunny, d_3 is Vincent and d_4 is Yolanda. Both Honey Bunny and Yolanda are robbers, while both Mia and Vincent are customers. Yolanda loves Honey Bunny and Vincent loves Mia. Sadly, Honey Bunny does not love Yolanda, Mia does not love Vincent, and nobody loves themselves.

Here's a second model for the same vocabulary. We'll use the same domain (that is, $D = \{d_1, d_2, d_3, d_4\}$) but change the interpretation function. To emphasise that the interpretation function has changed, we'll use a different symbol (namely F_2) for it:

$$\begin{aligned}
 F_2(\text{MIA}) &= d_2 \\
 F_2(\text{HONEY-BUNNY}) &= d_1 \\
 F_2(\text{VINCENT}) &= d_4 \\
 F_2(\text{YOLANDA}) &= d_3 \\
 F_2(\text{CUSTOMER}) &= \{d_1, d_2, d_4\} \\
 F_2(\text{ROBBER}) &= \{d_3\} \\
 F_2(\text{LOVE}) &= \emptyset.
 \end{aligned}$$

In this model, three of the individuals are customers, only one is a robber, and nobody loves anybody (the love relation is empty).

Note that in both the models we have defined so far, every entity in D is named by exactly one constant. It's important to realise that models *don't* have to be like this. Consider the model with $D_3 = \{d_1, d_2, d_3, d_4, d_5\}$ and the following interpretation function F_3 :

$$\begin{aligned}
 F_3(\text{MIA}) &= d_2 \\
 F_3(\text{HONEY-BUNNY}) &= d_1 \\
 F_3(\text{VINCENT}) &= d_4 \\
 F_3(\text{YOLANDA}) &= d_1 \\
 F_3(\text{CUSTOMER}) &= \{d_1, d_2, d_4\} \\
 F_3(\text{ROBBER}) &= \{d_3, d_5\} \\
 F_3(\text{LOVE}) &= \{(d_3, d_4)\}.
 \end{aligned}$$

In this model, not every entity has a name: both d_3 and d_5 are anonymous. Moreover, d_1 has two names. But this *is* a perfectly good first-order model. For a start, there simply is no requirement that every entity in a model must have a name; roughly speaking, we only bother to name entities of special interest. (So to speak, the domain is made up of stars, who are named, and extras, who are not.) Moreover, there simply is no requirement that each entity in a model must be named by at most one constant; just as in real life, one and the same entity may have several names. Incidentally, note that although d_3 and d_5 are anonymous, we do know something about them: they are both robbers.

Exercise 1.1.3 Once again consider the following situation: Vincent is relaxed. The gun rests on the back of the seat, pointing at Marvin. Jules is driving. Marvin is tense. Using the vocabulary you devised in Exercise 1.1.1, present this situation as a model (use the set-theoretic notation used in the text).

Exercise 1.1.4 Consider the following situation: There are four blocks. Two of the blocks are cubical, and two are pyramid shaped. The cubical blocks are small and red. The larger of the two pyramids is green, the smaller is yellow. Three of the blocks are sitting directly on the table, but the small pyramid is sitting on a cube. Devise a suitable vocabulary and present this situation as a model (use the set-theoretic notation used in the text).

First-Order Languages

Given some vocabulary, we build *the first-order language over that vocabulary* out of the following ingredients:

1. All the symbols in the vocabulary. We call these symbols the *non-logical* symbols of the language.
2. A countably infinite collection of variables x, y, z, w, \dots , and so on.
3. The boolean connectives \neg (negation), \wedge (conjunction), \vee (disjunction), and \rightarrow (implication).
4. The quantifiers \forall (the universal quantifier) and \exists (the existential quantifier).
5. The round brackets $)$ and $($ and the comma. (These are essentially punctuation marks; they are used to group symbols.)

Items 2–5 are common to all first-order languages: the only thing that distinguishes one first-order language from another is the choice of non-logical symbols (that is, the choice of vocabulary). The boolean connectives, are named after George Boole, a 19th century pioneer of modern mathematical logic. Incidentally, *countably infinite* means that our supply of variables can be indexed by the natural numbers: we can think of the variables at our disposal as $x_1, x_2, x_3, x_4, \dots$, and so on. So we'll never run out of variables: there will always be an additional new variable at our disposal should we need it.

So, suppose we've chosen some vocabulary. How do we mix these ingredients together? That is, what is the *syntax* of first-order languages? First of all, we define a first-order *term* τ to be any constant or any variable. (Later in this section, when we introduce function symbols, we'll see that some first-order languages allow us to form more richly structured terms than this.) Roughly speaking, terms are the noun phrases of first-order languages: constants can be thought of as first-order analogs of proper names, and variables as first-order analogs of pronouns.

What next? Well, we are then allowed to combine our ‘noun phrases’ with our ‘predicates’ (that is, the various relation symbols in the vocabulary) to form *atomic formulas*:

If R is a relation symbol of arity n , and τ_1, \dots, τ_n are terms, then $R(\tau_1, \dots, \tau_n)$ is an atomic (or basic) formula.

Intuitively, an atomic formula is the first-order counterpart of a natural language sentence consisting of a single clause (that is, what traditional grammars call a simple sentence). The intended meaning of $R(\tau_1, \dots, \tau_n)$ is that the entities named by the terms τ_1, \dots, τ_n stand

in the relation (or have the property) named by the symbol R . For example

$\text{LOVE}(\text{PUMPKIN}, \text{HONEY-BUNNY})$

means that the entity named PUMPKIN stands in the relation denoted by LOVE to the entity named HONEY-BUNNY—or more simply, that Pumpkin loves Honey Bunny. And

$\text{ROBBER}(\text{HONEY-BUNNY})$

means that the entity named HONEY-BUNNY has the property denoted by ROBBER—or more simply, that Honey Bunny is a robber.

Now that we know how to build atomic formulas, we can define more complex descriptions. The following inductive definition tells us exactly which *well formed formulas* (or *wffs*, or simply *formulas*) we can form.

1. All atomic formulas are wffs.
2. If ϕ and ψ are wffs then so are $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $(\phi \rightarrow \psi)$.
3. If ϕ is a wff, and x is a variable, then both $\exists x\phi$ and $\forall x\phi$ are wffs.
(We call ϕ the *matrix* of such wffs.)
4. Nothing else is a wff.

Roughly speaking, formulas built using \neg correspond to natural language expressions of the form “it is not the case that ...”; for example, the formula

$\neg\text{LOVE}(\text{PUMPKIN}, \text{HONEY-BUNNY})$

means it is not the case that Pumpkin loves Honey Bunny, or more simply, Pumpkin does not love Honey Bunny. Formulas built using \wedge correspond to natural language expressions of the form “... and ...”; for example

$(\text{LOVE}(\text{PUMPKIN}, \text{HONEY-BUNNY}) \wedge \text{LOVE}(\text{VINCENT}, \text{MIA}))$

means Pumpkin loves Honey Bunny and Vincent loves Mia. Formulas built using \vee correspond to expressions of the form “... or ...”; for example

$(\text{LOVE}(\text{PUMPKIN}, \text{HONEY-BUNNY}) \vee \text{LOVE}(\text{VINCENT}, \text{MIA}))$

means Pumpkin loves Honey Bunny or Vincent loves Mia. Formulas built using \rightarrow correspond to expressions of the form “if ... then ...”; for example

$(\text{LOVE}(\text{PUMPKIN}, \text{HONEY-BUNNY}) \rightarrow \text{LOVE}(\text{VINCENT}, \text{MIA}))$

means if Pumpkin loves Honey Bunny then Vincent loves Mia.

First-order formulas of the form $\exists x\phi$ and $\forall x\phi$ are called *quantified formulas*. Roughly speaking, formulas of the form $\exists x\phi$ correspond to

natural language expressions of the form “Some ...” (or “Something ...”, or “Somebody ...”, and so on). For example

$$\exists x \text{LOVE}(x, \text{HONEY-BUNNY})$$

means someone loves Honey Bunny. Formulas of the form $\forall x \phi$ correspond to natural language expressions of the form “all ...” (or “every ...”, or “Everything ...”, and so on). For example

$$\forall x \text{LOVE}(x, \text{HONEY-BUNNY})$$

means everyone loves Honey Bunny. And the quantifiers can be combined to good effect:

$$\exists x \forall y \text{LOVE}(x, y)$$

means someone loves everybody, and

$$\forall x \exists y \text{LOVE}(x, y)$$

means everybody loves someone.

In what follows, we sometimes need to talk about the *subformulas* of a given formula. The subformulas of a formula ϕ are ϕ itself and all the formulas used to build ϕ . For example, the subformulas of

$$\neg \forall y \text{PERSON}(y)$$

are $\text{PERSON}(y)$, $\forall y \text{PERSON}(y)$, and $\neg \forall y \text{PERSON}(y)$. However, we shall leave it to the reader to give a precise inductive definition of subformulahood (see Exercise 1.1.8) and turn instead to a more important topic: the distinction between *free* and *bound* variables.

Consider the following formula:

$$\neg(\text{CUSTOMER}(x) \vee (\forall x(\text{ROBBER}(x) \wedge \forall y \text{PERSON}(y))))$$

The first occurrence of x is *free*. The second and third occurrences of x are *bound*; they are bound by the first occurrence of the quantifier \forall . The first and second occurrences of the variable y are also bound; they are bound by the second occurrence of the quantifier \forall . Here's the full inductive definition:

1. Any occurrence of any variable is free in any atomic formula.
2. If an occurrence of any variable is free in ϕ or in ψ , then that same occurrence is free in $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $(\phi \rightarrow \psi)$.
3. If an occurrence of a variable x is free in ϕ , then that occurrence is free in $\forall y \phi$ and $\exists y \phi$ (for any variable y distinct from x). However, no occurrence of x is free in $\forall x \phi$ and $\exists x \phi$.
4. The only free variables in a formula ϕ are those whose freeness follows from the preceding clauses. Any variable in a formula that is not free is said to be bound.

We can now give the following definition: if a formula contains no occurrences of free variables, then it is called a *sentence* of first-order logic.

Although they are both called variables, free and bound variables are really very different. (In fact, some formulations of first-order logic use two distinct kinds of symbol for what we have lumped together under the heading “variable”.) Here’s an analogy. Try thinking of a free variable as something like the pronoun She in

She even has a stud in her tongue.

Uttered in isolation, this would be somewhat puzzling, as we don’t know who She refers to. But of course, such an utterance would be made in an appropriate *context*. This context might be either non-linguistic (for example, the speaker might be pointing to a heavily tattooed biker, in which case we would say that She was being used *deictically* or *demonstratively*) or linguistic (perhaps the speaker’s previous sentence was Honey Bunny is heavily into body piercing, in which case the name Honey Bunny supplies a suitable anchor for an *anaphoric* interpretation of She).

What’s the point of the analogy? Just as the pronoun She required something else (namely, contextual information) to supply a suitable referent, so will formulas containing free variables. Simply supplying a model won’t be enough; we need additional information on how to link the free variables to the entities in the model.

Sentences, on the other hand, are relatively self-contained. For example, consider the sentence $\forall x \text{ROBBER}(x)$. This is a claim that *every* individual is a robber. Roughly speaking, the bound variable x in ROBBER(x) acts as a sort of placeholder. In fact, the choice of x as a variable here is completely arbitrary: for example, the sentence $\forall y \text{ROBBER}(y)$ means exactly the same thing. Both sentences are simply a way of stating that no matter what entity we take the second occurrence of x (or y) as standing for, that entity will be a robber. Our discussion of the interpretation of first-order languages in first-order models will make these distinctions precise (indeed, most of the real work involved in interpreting first-order logic centres on the correct handling of free and bound variables).

But before turning to semantic issues, a few more remarks on first-order syntax are worth making. First, in what follows, we won’t always keep to the official first-order syntax defined above. In particular, we’ll generally try to use as few brackets as possible, as this tends to improve readability. For example, we would rarely write

(CUSTOMER(VINCENT) \wedge ROBBER(PUMPKIN)),

which is the official syntax. Instead, we would (almost invariably) drop the outermost brackets and write

$\text{CUSTOMER(VINCENT)} \wedge \text{ROBBER(PUMPKIN)}.$

To help reduce the bracket count even further, we assume the following precedence conventions for the boolean connectives: \neg binds more tightly than \vee and \wedge , both of which in turn bind more tightly than \rightarrow . What this means, for example, is that the formula

$\forall x(\neg \text{CUSTOMER}(x) \wedge \text{ROBBER}(x) \rightarrow \text{ROBBER}(x))$

is shorthand for

$\forall x((\neg \text{CUSTOMER}(x) \wedge \text{ROBBER}(x)) \rightarrow \text{ROBBER}(x)).$

In addition, we sometimes use the square brackets $]$ and $[$ as well as the official round brackets, as this can make the intended grouping of symbols easier to grasp visually. Further conventions are introduced in Exercise 1.2.3.

Second, the following terminology is quite useful: in a formula of the form $\phi \vee \psi$ the subformulas ϕ and ψ are called *disjuncts*, in a formula of the form $\phi \wedge \psi$ the subformulas ϕ and ψ are called *conjuncts*, and (most useful of all) in a formula of the form $\phi \rightarrow \psi$ the subformula ϕ is called the *antecedent* of the implication and the subformula ψ is called the *consequent*.

Finally, we remark that there is a natural division in first-order languages between formulas which do not contain quantifiers, and formulas which do. Intuitively, formulas which don't contain quantifiers are simpler: after all, then we don't need to bother about variable binding and the free/bound distinction. And indeed, as we shall learn in Chapter 4, in important respects the *quantifier-free fragment* of a first-order language is much simpler than the full first-order language of which it is a part.

Logicians have a special name for the quantifier-free part of first-order logic: they call it *propositional logic*. We suspect that most readers will have had some prior exposure to propositional logic, but for those who haven't, Appendix B discusses it and introduces the notational simplifications standardly used when working with this important sublogic.

Exercise 1.1.5 Represent the following English sentences in first-order logic:

1. If someone is happy, then Vincent is happy.
2. If someone is happy, and Vincent is not happy, then Jules is happy or Butch is happy.

3. Everyone is happy, or Butch and Pumpkin are fighting, or Vincent has a weird experience.
4. Some cars are damaged and there are bullet holes in some of the walls.
5. All the hamburger are tasty, all the fries are good, and some of the milkshakes are excellent.
6. Everybody in the basement is wearing a leather jacket or a dog collar.

Exercise 1.1.6 Consider the sentence Either Butch and Pumpkin are fighting or Vincent has a weird experience. Arguably the sense of Either ... or ... used in this sentence is stronger than the use of plain ... or ... in the sentence Butch and Pumpkin are fighting or Vincent has a weird experience. Explain the difference, and show how the stronger sense of Either ... or ... can be represented in first-order logic.

Exercise 1.1.7 Which occurrences of variables are bound, and which are free, in the following formulas:

1. ROBBER(y)
2. LOVE(x,y)
3. LOVE(x,y) → ROBBER(y)
4. $\forall y(\text{LOVE}(x,y) \rightarrow \text{ROBBER}(y))$
5. $\exists w\forall y(\text{LOVE}(w,y) \rightarrow \text{ROBBER}(y))$.

Exercise 1.1.8 Give an inductive definition of subformulahood. That is, for each kind of formula in the language (atomic, boolean, and quantified) specify exactly what its subformulas are.

Exercise 1.1.9 Use the inductive definition given in the text to prove that any occurrence of a variable in any formula must be either free or bound.

The Satisfaction Definition

Given a model of appropriate vocabulary, any sentence over this vocabulary is either true or false in that model. To put it more formally, there is a relation called *truth* which holds, or does not hold, between sentences and models of the same vocabulary. Now, using the informal explanation given above of what the boolean connectives and quantifiers mean, it is sometimes obvious how to check whether a given sentence is true in a given model: for example, to check the truth of $\forall x \text{ROBBER}(x)$ in a model we simply need to check that every individual in the model is in fact a robber.

But an intuition-driven approach to the truth of first-order sentences is not adequate for our purposes. For a start, when faced with a complex first-order sentence containing many connectives and quantifiers, our intuitions are likely to fade. Moreover, in this book we are interested in

computing with logic: we want a mathematically precise definition of when a first-order sentence is true in a model, a definition that lends itself to computational implementation.

Now, the obvious thing to try to do would be to give an inductive definition of first-order truth in a model; that is, it seems natural to define the truth (or falsity) of a sentence in terms of the truth (or falsity) of the subsentences of which it is composed. But there's a snag: we cannot give a *direct* inductive definition of truth, for the matrix of a quantified sentence typically won't be a sentence. For example, $\forall x \text{ROBBER}(x)$ is a sentence, but its matrix $\text{ROBBER}(x)$ is not. Thus an inductive truth definition defined solely in terms of sentences couldn't explain why $\forall x \text{ROBBER}(x)$ was true in a model, for there are no subsentences for such a definition to bite on.

Instead we proceed indirectly. We define a three place relation—called *satisfaction*—which holds between a formula, a model, and an *assignment of values to variables*. Given a model $M = (D, F)$, an *assignment of values to variables* in M (or more simply, an *assignment* in M) is a function g from the set of variables to D . Assignments are a technical device which tell us what the free variables stand for. By making use of assignment functions, we can inductively interpret *arbitrary* formulas in a natural way, and this will make it possible to define the concept of truth for *sentences*.

But before going further, one point is worth stressing: the reader should *not* view assignment functions simply as a technical fix designed to get round the problem of defining truth. Moreover, the reader should *not* think of satisfaction as being a poor relation of truth. If anything, satisfaction, not truth, is the fundamental notion, at least as far as natural language is concerned. Why is this?

The key to the answer is the word *context*. As we said earlier, free variables can be thought of as analogs of pronouns, whose values need to be supplied by context. An assignment of values to variables can be thought of as a (highly idealised) mathematical model of context; it rolls up all the contextual information into one easy to handle unit, specifying a denotation for every free variable. Thus if we want to use first-order logic to model natural language semantics, it is sensible to think in terms of three components: first-order formulas (descriptions), first-order models (situations) and variable assignments (contexts). The idea of assignment-functions-as-contexts is important in contemporary formal semantics; it has a long history and has been explored in a number of interesting directions (see the Notes at the end of the chapter for further discussion).

But let's return to the satisfaction definition. Suppose we've fixed

our vocabulary. (That is, from now on, when we talk of a model M , we mean a model of this vocabulary, and whenever we talk of formulas, we mean the formulas built from the symbols in that vocabulary.) We now give two further technical definitions which will enable us to state the satisfaction definition concisely.

First, let $M = (D, F)$ be a model, let g be an assignment in M , and let τ be a term. Then the *interpretation* of τ with respect to M and g is $F(\tau)$ if τ is a constant, and $g(\tau)$ if τ is a variable. We denote the interpretation of τ by $I_F^g(\tau)$.

The second idea we need is that of a *variant* of an assignment of values to variables. So, let g be an assignment in some model M , and let x be a variable. If g' is also an assignment in M , and for all variables y distinct from x we have that $g'(y) = g(y)$, then we say that g' is an x -*variant* of g . Variant assignments are the technical tool that allows us to try out new values for a given variable (say x) while keeping the values assigned to all other variables the same.

We are now ready for the satisfaction definition. Let ϕ be a formula, let $M = (D, F)$ be a model, and let g be an assignment in M . Then the relation $M, g \models \phi$ (ϕ is satisfied in M with respect to the assignment g) is defined inductively as follows:

$M, g \models R(\tau_1, \dots, \tau_n)$	iff	$(I_F^g(\tau_1), \dots, I_F^g(\tau_n)) \in F(R)$
$M, g \models \neg\phi$	iff	not $M, g \models \phi$
$M, g \models \phi \wedge \psi$	iff	$M, g \models \phi$ and $M, g \models \psi$
$M, g \models \phi \vee \psi$	iff	$M, g \models \phi$ or $M, g \models \psi$
$M, g \models \phi \rightarrow \psi$	iff	not $M, g \models \phi$ or $M, g \models \psi$
$M, g \models \exists x\phi$	iff	$M, g' \models \phi$, for some x -variant g' of g
$M, g \models \forall x\phi$	iff	$M, g' \models \phi$, for all x -variants g' of g .

(Here “iff” is shorthand for “if and only if”. That is, we are saying that the relationship on left-hand side holds precisely when the relationship on the right-hand side does too.) Note the crucial—and intuitive—role played by the x -variants in the clauses for the quantifiers. The clause for the existential quantifier boils down to this: $\exists x\phi$ is satisfied in a given model, with respect to an assignment g , if and only if there is some x -variant g' of g that satisfies ϕ in the model. That is, we have to try to find *some* value for x that satisfies ϕ in the model, while keeping the assignments to all other variables the same. Similarly, the clause for the universal quantifier says that $\forall x\phi$ is satisfied in a given model, with respect to an assignment g , if and only if *all* x -variants g' of g satisfy ϕ in the model. That is, no matter what x stands for, ϕ has to be satisfied in M .

We can now define what it means for a *sentence* to be true in a

model:

A sentence ϕ is true in a model M if and only if for *any* assignment g of values to variables in M , we have that $M, g \models \phi$. If ϕ is true in M we write $M \models \phi$

This is an elegant definition of truth that beautifully mirrors the special, self-contained nature of sentences. It hinges on the following observation: *it simply doesn't matter which variable assignment we use to compute the satisfaction of sentences*. Sentences contain no free variables, so the only free variables we will encounter when evaluating one are those produced when evaluating its quantified subformulas (if it has any). But the satisfaction definition tells us what to do with such free variables: simply try out variants of the current assignment and see whether they satisfy the matrix or not. In short, start with whatever assignment you like—the result will be the same. It is reasonably straightforward to make this informal argument precise, and the reader is asked to do so in Exercise 1.1.15.

Still, for all the elegance of the truth definition, satisfaction is the fundamental concept. Not only is satisfaction the technical engine powering the definition of truth, but from the perspective of natural language semantics it is the fundamental notion. By making *explicit* the role of variable assignments, it holds up a simple mirror to the process of evaluating descriptions in situations while making use of contextual information.

Exercise 1.1.10 Consider the model with $D = \{d_1, d_2, d_3, d_4, d_5\}$ and the following interpretation function F :

$$\begin{aligned} F(MIA) &= d_2 \\ F(HONEY-BUNNY) &= d_1 \\ F(VINCENT) &= d_4 \\ F(YOLANDA) &= d_1 \\ F(CUSTOMER) &= \{d_1, d_2, d_4\} \\ F(ROBBER) &= \{d_3, d_5\} \\ F(LOVE) &= \{(d_3, d_4)\}. \end{aligned}$$

Are the following sentences true or false in this model?

1. $\exists x \text{LOVE}(x, VINCENT)$
2. $\forall x (\text{ROBBER}(x) \rightarrow \neg \text{CUSTOMER}(x))$
3. $\exists x \exists y (\text{ROBBER}(x) \wedge \neg \text{ROBBER}(y) \wedge \text{LOVE}(x, y))$.

Exercise 1.1.11 Give a model that makes all the following sentences true:

1. $\text{HAS-GUN}(VINCENT)$
2. $\forall x (\text{HAS-GUN}(x) \rightarrow \text{AGGRESSIVE}(x))$
3. $\text{HAS-MOTORBIKE}(BUTCH)$
4. $\forall y (\text{HAS-MOTORBIKE}(y) \vee \text{AGGRESSIVE}(y))$

5. $\neg \text{HAS-MOTORBIKE}(\text{JULES})$.

Exercise 1.1.12 Our presentation of first-order logic did not introduce all the boolean connectives the reader is likely to encounter. For example, we did not introduce the symbol \leftrightarrow , the *bi-implication* (or *if and only if*) connective. A formula of the form $\phi \leftrightarrow \psi$ has the following satisfaction condition:

$$M, g \models \phi \leftrightarrow \psi \quad \text{iff} \quad \begin{aligned} &\text{either } M, g \models \phi \text{ and } M, g \models \psi, \\ &\text{or not } M, g \models \phi \text{ and not } M, g \models \psi. \end{aligned}$$

That is, a bi-implication holds whenever both ϕ and ψ are satisfied (in some model M with respect to g), or when neither ϕ nor ψ are satisfied (in some model M with respect to g).

But we did not lose out by not introducing the bi-implication connective, for anything we can say with its help we can say without: $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ means exactly the same thing as $\phi \leftrightarrow \psi$. To be more precise, $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ is satisfied in a model M with respect to an assignment g precisely when $\phi \leftrightarrow \psi$ is. Prove this.

Exercise 1.1.13 Two other booleans we did not introduce are \perp and \top . Syntactically, these expressions behave like atomic formulas. Semantically, \perp is never satisfied in any model with respect to any assignment whatsoever, and \top is satisfied in every model with respect to every assignment (that is, we can gloss \perp as “always false”, and \top as “always true”).

Show that $\phi \rightarrow \perp$ means the same thing as $\neg \phi$ (that is, show that two expressions are satisfied with respect to exactly the same models and assignments). Furthermore, show that \top means the same thing as $\perp \vee \neg \perp$.

Exercise 1.1.14 When we informally discussed the semantics of bound variables we claimed that in any model of appropriate vocabulary, $\forall x \text{ROBBER}(x)$ and $\forall y \text{ROBBER}(y)$ mean exactly the same thing. We can now be more precise: we claim that the first sentence is true in precisely the same models as the second sentence. Prove this.

Exercise 1.1.15 We claimed that when evaluating *sentence*s, it doesn't matter which variable assignment we start with. Formally, we are claiming that given any *sentence* ϕ and any model M (of the same vocabulary), and any variable assignments g and g' in M , then $M, g \models \phi$ iff $M, g' \models \phi$. We want the reader to do two things. First, show that the claim is *false* if ϕ is not a sentence but a formula containing free variables. Second, show that the claim is *true* if ϕ is a *sentence*.

Exercise 1.1.16 For any formula ϕ , given two assignments g and g' which differ only in what they assign to variables *not* in ϕ , and any model M (of appropriate vocabulary) then we have that $M, g \models \phi$ iff $M, g' \models \phi$. Prove this.

This result tells us that we don't need to worry about entire variable assignments, but only about the (finite) part of the assignment containing information about the (finitely many) variables that actually occur in the formulas being evaluated. Indeed, instead of mentioning entire assignment functions and writing things like $M, g \models \phi$, logicians often prefer to specify only what has been assigned to the free variables in the formula being evaluated. For example, if ϕ is a formula with only x and y free, a logician would be likely to write things like $M \models \phi[x \leftarrow d_1, y \leftarrow d_4]$ (assign d_1 to the free variable x , and d_4 to the free variable y).

Function Symbols, Equality, and Sorted First-Order Logic

We have now presented the core ideas of first-order logic, but before moving on we'll discuss three extensions of the basic formalism: first-order logic with function symbols, first-order logic with equality, and sorted first-order logic.

Let's first look at function symbols. Suppose we want to talk about Butch, Butch's father, Butch's grandfather, Butch's great grandfather, and so on. Now, if we know the names of all these people this is easy to do—but what if we don't? A natural solution is to add a 1-place function symbol **FATHER** to the language. Then if **BUTCH** is the constant that names Butch, **FATHER(BUTCH)** is a term that picks out Butch's father, **FATHER(FATHER(BUTCH))** picks out Butch's grandfather, and so on. That is, function symbols are a syntactic device that let us form recursively structured terms, thus letting us express many concepts in a natural way.

Let's make this precise. First, we shall suppose that it is the task of the vocabulary to tell us which function symbols we have at our disposal, and what the arity of each of these symbols is. Second, given this information, we say (as before) that a model M is a pair (D, F) where F interprets the constants and relation symbols as described earlier, and, in addition, F assigns to each function symbol f an appropriate semantic entity. What's an appropriate interpretation for an n -place function symbol? Simply a function that takes n -tuples of elements of D as input, and returns an element of D as output. Third, we need to say what terms we can form using these new symbols. Here's the definition we require:

1. All constants and variables are terms.
2. If f is a function symbol of arity n , and τ_1, \dots, τ_n are terms, then $f(\tau_1, \dots, \tau_n)$ is also a term.
3. Nothing else is a term.

A term is said to be *closed* if and only if it contains no variables.

Only one task remains: interpreting these new terms. In fact we need simply extend our earlier definition of I_F^g in the obvious way. Given a model M and an assignment g in M , we define (as before) $I_F^g(\tau)$ to be $F(\tau)$ if τ is a constant, and $g(\tau)$ if τ is a variable. On the other hand, if τ is a term of the form $f(\tau_1, \dots, \tau_n)$, then we define $I_F^g(\tau)$ to be $F(f)(I_F^g(\tau_1), \dots, I_F^g(\tau_n))$. That is, we apply the n -place function $F(f)$ —the function interpreting f —to the interpretation of the n argument terms.

Function symbols are a natural extension to first-order languages, as the fatherhood example should suggest. However, in this book we won't use them in our analyses of natural language semantics, we'll use them for more technical purposes. In particular, function symbols play an important role in Chapter 5, where (in the guise of Skolem functions) they will help us formulate inference systems for first-order logic suitable for computational implementation.

Let's now turn to equality. The first-order languages we have so far defined have a curious expressive shortcoming: we have no way to assert that two terms denote the same entity. This is real weakness as far as natural language semantics is concerned—for example, we may wish to assert that Marsellus's wife and Mia are the same person. What are we to do?

The solution is straightforward. Given any language of first-order logic (with or without function symbols) we can turn it into a first-order language *with equality* by adding the special two place relation symbol $=$. We use this relation symbol in the natural *infix* way: that is, if τ_1 and τ_2 are terms then we write $\tau_1 = \tau_2$ instead of the rather ugly $= (\tau_1, \tau_2)$. Beyond this notational convention, there's nothing to say about the syntax of $=$; it's just a two-place relation symbol. But what about its semantics?

Here matters are more interesting. Although, syntactically, $=$ is just a 2-place relation symbol, it is a very special one. In fact (unlike LOVE, or HATE, or any other two-place relation symbol) we are *not* free to interpret it how we please. In fact, given any model M , any assignment g in M , and any terms τ_1 and τ_2 , we shall insist that

$$M, g \models \tau_1 = \tau_2 \text{ iff } I_F^g(\tau_1) = I_F^g(\tau_2).$$

That is, the atomic formula $\tau_1 = \tau_2$ is satisfied if and only if τ_1 and τ_2 have exactly the same interpretation. In short, $=$ really means equality. In fact, $=$ is usually regarded as a *logical* symbol on a par with \neg or \forall , for like these symbols it has a fixed interpretation, and a semantically fundamental one at that.

So we can now assert that two names pick out the same individual.

For example, to say that Yolanda is Honey Bunny we use the formula

YOLANDA=HONEY-BUNNY,

and (if we make use of the 1-place function symbol **WIFE** to mean wife-of) we can say that Mia is Marsellus's wife as follows:

MIA=WIFE(MARSELLUS).

The final extension of the basic first-order formalism we shall consider is *sorted first-order logic*. Sometimes the things we want to talk about are divided into natural subclasses—for example, objects in the world can be categorised as either animate or inanimate. In such cases, it can be natural to work with first-order languages with special variables which can only be interpreted by the individuals in a particular subclass. Why? Because the use of such *sorted variables* enables us to make simple and direct statements about (say) animate and inanimate objects. For example, to say that All animate objects breath we could simply use the expression

$\forall a \text{BREATH}(a)$

where we are using the “a” as a variable which can only be interpreted as an animate object. The use of the special variable enables us to make a direct universal statement about all the individuals of a certain sort (here, animate objects). Similarly, if we wanted to say that no inanimate object talks we could use the expression

$\neg \exists i \text{TALK}(i).$

Here we are using “i” as a variable which can only be interpreted as an inanimate object, and once again the fact that the variable is restricted in its interpretation enables us to say what we want simply and directly.

So, sorted notation is convenient and compact, and indeed we shall briefly make use of it in Chapter 3 when we discuss hole semantics. Nonetheless, though sometimes convenient, anything that can be said in sorted first-order logic can also be said in ordinary first-order logic. For example, consider again the sentence All animate objects breath. To represent this sentence in ordinary first-order logic, simply use the following expression:

$\forall x (\text{ANIMATE}(x) \rightarrow \text{BREATH}(x)).$

Although the variable **x** can be interpreted by any object (animate or inanimate), we achieve the desired restriction to animate objects by making use of the unary relation symbol **ANIMATE**. Needless to say, we insist that this symbol be interpreted by the set of animate objects. Similarly, to say that no inanimate object talks we could use the ordinary first-order expression

$$\neg \exists x (\text{INANIMATE}(x) \wedge \text{TALK}(x)).$$

Once again, instead of making use of a special sort of variable that is restricted in its interpretation, we have made use of an extra unary relation symbol (here, `INANIMATE`) which we insist be interpreted by the individuals in the relevant sort. The strategy suggested by these examples is completely general: instead of making use of variables that are restricted in their interpretation to certain sorts of object, we can introduce special relation symbols to pick out the sorts we are interested in.

We mostly work with ordinary (unsorted) first-order logic in this book. For references on sorted first-order logic, see the Notes at the end of the chapter.

Exercise 1.1.17 Use function symbols and equality to say that Butch's mother is Mia's grandmother.

Exercise 1.1.18 There is a famous analysis, due to the philosopher Bertrand Russell, of the meaning of the determiner `the` in sentences like `The robber is screaming`. Russell claims that this sentence would be true in some situation if (a) there was at least one robber in the situation, (b) there was at most one robber in the situation, and (c) that robber was screaming. Write down a first-order sentence which expresses this analysis of `The robber is screaming`. Note: you will have to use the equality symbol.

1.2 Three Inference Tasks

Now that we know what first-order languages are, we have at our disposal a fundamental tool for representing the meaning of natural language expressions. But first-order logic can help with more than just representation: it also gives us a grip on inference, and this is the topic to which we now turn. We shall introduce three inference tasks: *querying*, *consistency checking*, and *informativity checking*. The three tasks are fundamental to computational semantics, and can be combined in various ways to deal with many interesting problems in the semantics of natural language.

By the end of the section the reader will have a good understanding of what these three tasks are. We shall learn that the querying task is relatively simple and can be handled by a piece of software called a *model checker*. We shall also learn that the consistency and informativity checking tasks are closely related, that both are extremely difficult (indeed, *undecidable*), and that developing computational tools to deal with them will force us to move on from the model-theoretic

perspective of this chapter to the *proof-theoretic* perspective developed in Chapters 4 and 5.

The Querying Task

The querying task is the simplest of the three inference tasks we shall consider. It is defined as follows:

The Querying Task Given a model M and a first-order formula ϕ , is ϕ satisfied in M or not?

And now we must consider two further questions: Why is this task interesting? And can we deal with it computationally?

Models are situations and first-order formulas are descriptions. Thus to ask whether a description holds or does not hold in a given situation is to ask a fundamental question. Moreover, it is a question that can be very useful; this may become clearer if we think in terms of databases rather than situations.

A database is a structured collection of facts; databases vary in how (and to what extent) they are structured, but if you think of a conventional database as a first-order model you will not go far wrong. Now, real databases are (often huge) repositories of content, and this content is typically accessed by posing queries in specialised languages called database query languages. The querying task we defined above is essentially a more abstract version of what goes on in conventional database querying, with first-order logic playing the role of the database query language, and models playing the role of databases.

But what does this have to do with natural language? Here's one link. Suppose we have represented some situation of interest as a model (maybe this model is some pre-existing database, or maybe it's something we have dynamically constructed to keep track of a dialogue). But however the model got there, it embodies content we are interested in, and it is natural to try and use this content to provide answers to questions. Now (as we shall learn in Chapter 6) it is possible to translate some kinds of natural language questions into first-order logic. And if we do this, we have a way of answering them: we see if their translations are satisfied in the model. In Chapter 6 we construct a simple question answering system based on this idea.

Now for the second question: is querying a task we can compute? The answer is basically *yes*, but there are two points we need to be careful about.

First, note that we defined the querying task for arbitrary formulas, not just for sentences. And if a formula contains free variables we can't simply evaluate it in a model—we also need to stipulate what the free

variables stand for. Second, we don't have a remotest chance of writing software for querying arbitrary models. Many (in fact, most) models are infinite, and while it is possible to give useful finite representations of some infinite models, most are too big and unruly to be worked with computationally. Hence we should confine our attention to finite models (and given our models-as-databases analogy, this is a reasonable restriction to make).

If we remember to pay attention to the free variables, and confine our attention to finite models, then it certainly *is* possible to write a program which performs the querying task. Such a program is called a *model checker*, and in the following section we shall write a first-order model checker in Prolog. We shall use this model checker when we discuss question answering in Chapter 6.

A final remark. People with traditional logical backgrounds may be surprised that we have defined querying as an inference task: traditional logic texts typically don't define this task, and many logicians wouldn't consider evaluating a formula in a model to be a form of inference. In our view, however, querying is a paradigmatic example of inference. Consider what it involves. On the one hand, we have the model, a repository (of a possibly vast amount) of low-level information about entities, relationships, and properties. On the other hand we have formulas, which may describe aspects of the situation in an abstract, logically complex, way. Given even a not-very-large model and a not-particularly-complex formula, it may be far from obvious whether or not the formula is satisfied in the model. Computing whether a formula is satisfied (or not satisfied) in a model is thus a beautiful example of a process which makes implicit information explicit. Hence querying can be viewed as a form of inference.

The Consistency Checking Task

Consistency is a commonly used concept in linguistics (especially in semantics) and its central meaning is something like this: a consistent description is one that "makes sense", or "is intelligible", or "describes something realisable". For example, *Mia is a robber* is obviously consistent, for it describes a possible state of affairs, namely a situation in which *Mia* is a robber. An inconsistent description, on the other hand "doesn't make sense", or "attempts to describe something impossible". For example, *Mia is a robber and Mia is not a robber* is clearly inconsistent. This description tears itself apart: it simultaneously affirms and denies that *Mia* is a robber, and hence fails to describe a possible situation.

Consistency and inconsistency are important in computational se-

mantics. Suppose we are analysing a discourse sentence by sentence. If at some stage we detect an inconsistency, this may be a sign that something has gone wrong with the communicative process. To give a blatant example, the discourse

Mia is happy. Mia is not happy.

is obviously strange. It is hard to know what to do with the ‘information’ it contains—but naively accepting it and attempting to carry on is probably not the best strategy. Thus we would like to be able to detect inconsistency when it arises, for it typically signals trouble.

But the ‘definitions’ given above of consistency and inconsistency are imprecise. If we are to work with these notions computationally, we need to pin them down clearly. Can the logical tools we have been discussing help us define precise analogs of these concepts, analogs which do justice to the key intuitions? They can: it is natural to identify the pre-theoretic concept of consistency with the model-theoretic concept of *satisfiability*, and to identify inconsistency with *unsatisfiability*.

A first-order formula is called satisfiable if it is satisfied in at least one model. As we have encouraged the reader to think of models as idealised situations, this means that satisfiable formulas are those which describe conceivable, or possible, or realisable situations. For example, ROBBER(MIA) describes a realisable situation, for any model in which Mia is a robber satisfies it. A formula that is not satisfiable in any model is called unsatisfiable. That is, unsatisfiable formulas describe inconceivable, or impossible, or unrealisable situations. For example, ROBBER(MIA) $\wedge \neg$ ROBBER(MIA) describes something that is unrealisable: there simply aren’t any models in which Mia both is and is not a robber.

It is useful to extend the concepts of satisfiability and unsatisfiability to finite sets of formulas. A finite set of formulas $\{\phi_1, \dots, \phi_n\}$ is satisfiable if $\phi_1 \wedge \dots \wedge \phi_n$ is satisfiable; that is, satisfiability for finite sets of formulas mean “lump together all the information in the set using conjunction and see if the result is satisfiable”. Similarly, a finite set of formulas $\{\phi_1, \dots, \phi_n\}$ is unsatisfiable if $\phi_1 \wedge \dots \wedge \phi_n$ is unsatisfiable.

Note that satisfiability (and unsatisfiability) are *model-theoretic* or (as it is sometimes put) *semantic* concepts. That is, both concepts are defined using the notion of satisfaction in a model, and nothing else. Furthermore, note that satisfiability (and unsatisfiability) are mathematically precise concepts: we know exactly what first-order languages and first-order models are, and we know exactly what it means when we claim that a formula is satisfied in a model. Finally, it seems reasonable to claim that the notion of a formula being satisfied in a model

is a good analog of the pre-theoretic notion of descriptions that describe realisable states of affairs. Hence for the remainder of the book we shall identify the mathematical notions “satisfiable” and “unsatisfiable” with the pre-theoretical notions “consistent” and “inconsistent” respectively, and accordingly, we define the consistency checking task as follows:

The Consistency Checking Task Given a first-order formula ϕ , is ϕ consistent (that is: satisfiable) or inconsistent (that is: unsatisfiable)?

Some logicians might prefer to call this the satisfiability checking task. But we prefer to talk about consistency checking, for it emphasises the pre-theoretic notion we are trying to capture. Incidentally, consistency checking for finite sets of formulas is done in the obvious way: we take the conjunction of the finite set, and test whether or not it is satisfiable.

But is consistency checking something we can compute? The answer is *no*, not fully. Consistency checking for first-order logic turns out to be a very hard problem indeed. In fact, a well-known theorem of mathematical logic tells us that there is no algorithm capable of solving this problem for all possible input formulas. It is a classic example of a *computationally undecidable task*.

We are not going to prove this undecidability result (see the Notes at the end of the chapter for pointers to some nice proofs), but the following remarks should help you appreciate why the problem is so hard. Note that the consistency checking task is *highly abstract* compared to the querying task. Whereas the querying task is about manipulating two concrete entities (a finite model and a formula), the consistency checking task is a search problem—and what a search problem! Given a formula, we have to determine if somewhere out there in the (vast) mathematical universe of models, a satisfying model exists. Now, even if a finite satisfying model exists, there are a lot of finite models; how do we find the one we’re looking for? And anyway, some satisfiable formulas only have infinite satisfying models (see Exercise 1.2.1); how on earth can we find such models computationally?

Hopefully these remarks have given you some sense of why first-order consistency checking is difficult. Indeed, given what we have just said, it may seem that we should simply give up and go home! Remarkably, however, all is not lost. As we shall learn in Chapters 4 and 5, it is possible to take a different perspective on consistency checking, a *proof-theoretic* (or *syntactic*) perspective rather than a model-theoretic perspective. That is, it turns out to be possible to re-analyse consistency checking from a perspective that emphasises symbol manipulation, not

model existence. The proof-theoretic perspective makes it possible to create software that offers a useful partial solution to the consistency checking problem, and (as we shall see in Chapter 6) such software is useful in computational semantics.

The Informativity Checking Task

The main goal of this section is to get to grips with the pre-theoretic concept of informativity (and uninformativity). We'll start by defining the model-theoretic concepts of validity and invalidity, and valid and invalid arguments. We'll then show that these concepts offer us a natural way of thinking about informativity.

A *valid* formula is a formula that is satisfied in all models (of the appropriate vocabulary) given any variable assignment. To put it the other way around: if ϕ is a valid formula, it is impossible to find a situation and a context in which ϕ is not satisfied. For example, consider $\text{ROBBER}(x) \vee \neg\text{ROBBER}(x)$. In any model, given any variable assignment, one (and indeed, only one) of the two disjuncts must be true, and hence the whole formula will be satisfied too. We indicate that a formula ϕ is valid by writing $\models \phi$. A formula that is not valid is called *invalid*. That is, invalid formulas are those which fail to be satisfied in at least one model. For example $\text{ROBBER}(x)$ is an invalid formula: it is not possible to satisfy it in any model where there are no robbers. We indicate that a formula ϕ is invalid by writing $\not\models \phi$.

There is a clear sense in which validities are logical: nothing can affect them, they carry a cast-iron guarantee of satisfiability. But logic is often thought of in terms of the more dynamic notion of *valid arguments*, a movement, or inference, from premises to conclusions. This notion can also be captured model-theoretically. Suppose ϕ_1, \dots, ϕ_n , and ψ are a finite collection of first-order formulas. Then we say that the argument with *premises* ϕ_1, \dots, ϕ_n and *conclusion* ψ is a *valid argument* if and only if whenever all the premises are satisfied in some model, using some variable assignment, then the conclusion is satisfied in the same model using the same variable assignment. The notation

$$\phi_1, \dots, \phi_n \models \psi$$

means that the argument with premises ϕ_1, \dots, ϕ_n and conclusion ψ is valid. Incidentally, there are many ways of speaking of valid arguments. For example, it is also common to say that ψ is a *valid inference* from the premises ϕ_1, \dots, ϕ_n , or that ψ is a *logical consequence* of ϕ_1, \dots, ϕ_n , or that ψ is a *semantic consequence* of ϕ_1, \dots, ϕ_n .

An argument that is not valid is called *invalid*. That is, an argument with premises ϕ_1, \dots, ϕ_n and conclusion ψ is an invalid argument if it

is possible to find a model and a variable assignment which satisfies all the premises but not the conclusion. We indicate that an argument is invalid by writing

$$\phi_1, \dots, \phi_n \not\models \psi.$$

Validity and valid arguments are closely related. For example, the argument with premises $\forall x(\text{ROBBER}(x) \rightarrow \text{CUSTOMER}(x))$ and $\text{ROBBER}(\text{MIA})$ and the conclusion $\text{CUSTOMER}(\text{MIA})$ is valid. That is:

$$\forall x(\text{ROBBER}(x) \rightarrow \text{CUSTOMER}(x)), \text{ROBBER}(\text{MIA}) \models \text{CUSTOMER}(\text{MIA}).$$

But now consider the following (valid) formula:

$$\models \forall x(\text{ROBBER}(x) \rightarrow \text{CUSTOMER}(x)) \wedge \text{ROBBER}(\text{MIA}) \rightarrow \text{CUSTOMER}(\text{MIA}).$$

Pretty clearly, the validity of the formula mirrors the validity of the argument. And indeed, as this example suggests, with the help of the boolean connectives \wedge and \rightarrow we can convert any valid argument into a validity. This is an example of the *Semantic Deduction Theorem* in action. This theorem is fully stated (and the reader is asked to prove it) in Exercise 1.2.2 below.

Validity and valid arguments are central concepts in model theory. Logicians regard validities as important because validities embody the abstract patterns that underly logical truth, and they regard valid arguments as important because of the light they throw on mathematical reasoning. (In a nutshell: for logicians, validities and valid arguments are the good guys.) However, it is possible to view validity and valid argumentation in a less positive way, and this brings us to the linguistically important concepts of informativity and uninformativity.

There is a clear sense in which valid formulas are *uninformative*. Precisely because they are satisfied in all models, valid formulas don't tell us anything at all about any particular model. That is, valid formulas don't rule out possibilities—they're boringly vanilla. Thus we shall introduce an alternative name for valid formulas: we shall often call them *uninformative* formulas, and we shall call invalid formulas *informative* formulas. (So the good guys have become the bad guys and vice-versa.) Moreover, as the concept of informativity turns out to be linguistically relevant, we shall define the following task:

The Informativity Checking Task Given a first-order formula ϕ , is ϕ informative (that is: invalid) or uninformative (that is: valid)?

Valid arguments can be accused of uninformativity too. If $\phi_1, \dots, \phi_n \models \psi$, and we already know that ϕ_1, \dots, ϕ_n , then there is a sense in which learning ψ doesn't tell us anything new. For this reason we shall introduce the following alternative terminology: if $\phi_1, \dots, \phi_n \models \psi$, then we shall say that ψ is *uninformative with respect to* ϕ_1, \dots, ϕ_n . On

the other hand, suppose that $\phi_1, \dots, \phi_n \not\models \psi$, and that we already know that ϕ_1, \dots, ϕ_n . Then if we are told ψ , we clearly *have learned* something new. Hence, if $\phi_1, \dots, \phi_n \not\models \psi$, then we shall say that ψ is *informative with respect to* ϕ_1, \dots, ϕ_n .

By appealing to the Semantic Deduction Theorem, we can reduce testing ψ for informativity (or uninformativity) with respect to ϕ_1, \dots, ϕ_n to ordinary informativity checking. For the theorem tells us that ψ is informative with respect to ϕ_1, \dots, ϕ_n if and only if $\phi_1 \wedge \dots \wedge \phi_n \rightarrow \psi$ is an informative formula. So we can always reduce informativity issues to a task involving a single formula, and this is the strategy we shall follow in Chapter 6.

But *why* should computational linguists care about informativity? The point is this: like inconsistency, uninformativity can be a sign that something is going wrong with the communicative process. If later sentences in a discourse are consequences of earlier ones, this should probably make us suspicious, not happy. To give a particularly blatant example, consider the discourse

Mia is married. Mia is married. Mia is married.

Obviously we should *not* clap our hands here and say “How elegant! The second sentence is a logical consequence of the first, and the third is a logical consequence of the second!” This is a clear example of malfunctioning discourse. It patently fails to convey any new information. If it was produced by a natural language generation system, we would suspect the system needed debugging. If it was uttered by a person, we would probably look for another conversational partner.

Now, it is important not to overstate the case here. In general, lack of informativity is not such a reliable indicator of communicative problems as inconsistency. For a start, sometimes we may be interested in discourses that embody valid argumentation (for example, if we are working with mathematical text). Furthermore, sometimes it is appropriate to rephrase the same information in different ways. For example, consider this little discourse:

Mia is married. She has a husband.

The second sentence is uninformative with respect to the first, but this discourse would be perfectly acceptable in many circumstances. Nonetheless, although uninformativity is not a failsafe indicator of trouble, it is often important to detect whether or not genuinely new information is being transmitted. So we need tools for carrying out informativity checking.

And this brings us to the next question: is informativity checking computable? And once again the answer is *no*, not fully. Like the con-

sistency checking task, the informativity checking task is undecidable. We're not going to prove this result, but given our previous discussion it is probably clear that informativity checking is likely to be tough. After all, informativity checking is a highly abstract task: validity means satisfiable in *all* models, and there are an awful lot of awfully big models out there.

This sounds like bad news, but once again there is light at the end of the tunnel. As we shall learn in Chapters 4 and 5, it is possible to take a proof-theoretic perspective on informativity checking. Instead of viewing informativity in terms of satisfaction in all models, it is possible to reanalyse it in terms of certain kinds of symbol manipulation. This proof-theoretic perspective makes it possible to create software that offers practical partial solutions to the informativity checking problem, and (as we shall see in Chapter 6) we can apply this software to linguistic issues.

Exercise 1.2.1 Consider the following formula:

$$\begin{aligned} & \forall x \exists y \text{BIGGERTHAN}(x,y) \wedge \forall x \neg \text{BIGGERTHAN}(x,x) \\ & \wedge \forall x \forall y \forall z (\text{BIGGERTHAN}(x,y) \wedge \text{BIGGERTHAN}(y,z) \rightarrow \text{BIGGERTHAN}(x,z)). \end{aligned}$$

Is this formula satisfiable? Is it satisfiable in a *finite* model?

Exercise 1.2.2 The *Semantic Deduction Theorem* for first-order logic says that $\phi_1, \dots, \phi_n \models \psi$ if and only if $\models (\phi_1 \wedge \dots \wedge \phi_n) \rightarrow \psi$. (That is, we can lump together the premises using \wedge , and then use \rightarrow to state that this information implies the conclusion.) Prove the Semantic Deduction Theorem.

Exercise 1.2.3 We say that two formulas ϕ and ψ are *logically equivalent* if and only if $\phi \models \psi$ and $\psi \models \phi$. For all formulas ϕ , ψ , and θ :

1. Show that $\phi \wedge \psi$ is logically equivalent to $\psi \wedge \phi$, and that $\phi \vee \psi$ is logically equivalent to $\psi \vee \phi$ (that is, show that \wedge and \vee are both *commutative*).
2. Show that $(\phi \wedge \psi) \wedge \theta$ is logically equivalent to $\phi \wedge (\psi \wedge \theta)$, and that $(\phi \vee \psi) \vee \theta$ is logically equivalent to $\phi \vee (\psi \vee \theta)$ (that is, show that \wedge and \vee are both *associative*).
3. Show that $\phi \rightarrow \psi$ is logically equivalent to $\neg \phi \vee \psi$ (that is, show that \rightarrow can be regarded as a symbol defined in terms of \neg and \vee). Furthermore, show that $\neg(\phi \rightarrow \psi)$ is logically equivalent to $\phi \wedge \neg \psi$.
4. Show that $\forall x \phi$ and $\neg \exists x \neg \phi$ are logically equivalent, and that so are $\exists x \phi$ and $\neg \forall x \neg \phi$ (that is, show that either quantifier can be defined in terms of the other with the help of \neg).

These equivalences come in useful in the course of the book. Equivalences 3 and 4 will enable us to take two handy shortcuts when implementing our model checker. And because \wedge and \vee are associative, it is natural to drop

brackets and write conjunctions such as $\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4$ and disjunctions such as $\psi_1 \vee \psi_2 \vee \psi_3 \vee \psi_4$. We often adopt this convention.

Exercise 1.2.4 Show that $\neg(\phi \vee \psi)$ is logically equivalent to $\neg\phi \wedge \neg\psi$, and that $\neg(\phi \wedge \psi)$ is logically equivalent to $\neg\phi \vee \neg\psi$. (These equivalences are called the *De Morgan laws*, and we will make use of them when we discuss resolution theorem proving in Chapters 4 and 5.)

Exercise 1.2.5 Show that $\theta \vee (\phi \wedge \psi)$ is logically equivalent to $(\theta \vee \phi) \wedge (\theta \vee \psi)$, that $(\phi \wedge \psi) \vee \theta$ is logically equivalent to $(\phi \vee \theta) \wedge (\psi \vee \theta)$, that $\theta \wedge (\phi \vee \psi)$ is logically equivalent to $(\theta \wedge \phi) \vee (\theta \wedge \psi)$ and that $(\phi \vee \psi) \wedge \theta$ is logically equivalent to $(\phi \wedge \theta) \vee (\psi \wedge \theta)$. (These equivalences are called the *distributive laws*, and we will make use of two of them when we discuss resolution theorem proving in Chapters 4 and 5.)

Relating Consistency and Informativity

Now for an important observation. As we've just learned, two of the inference tasks that interest us (namely the consistency checking task and the informativity checking task) are extremely difficult. Indeed, in order to make progress with them, we're going to have to rethink them from a symbol manipulation perspective (we'll do this in Chapters 4 and 5). But one piece of good news is at hand: the two tasks are intimately related, and knowing how to solve one helps us to solve the other.

Here are the key observations:

1. ϕ is consistent (that is, satisfiable) if and only if $\neg\phi$ is informative (that is, invalid).
2. ϕ is inconsistent (that is, unsatisfiable) if and only if $\neg\phi$ is uninformative (that is, valid).
3. ϕ is informative (that is, invalid) if and only if $\neg\phi$ is consistent (that is, satisfiable).
4. ϕ is uninformative (that is, valid) if and only if $\neg\phi$ is inconsistent (that is, unsatisfiable).

Why do these relationships hold? Consider, for example, the first. Suppose ϕ is consistent. This means it is satisfiable in at least one model. But this is the same as saying that there is at least one model where $\neg\phi$ is not satisfied. Which is precisely to say that $\neg\phi$ is informative. The remaining three equivalences are variations on this theme, and the reader is invited to explore them in Exercise 1.2.6.

These relationships are useful in practice. For example, in Chapters 4 and 5 we discuss *theorem provers*. A theorem prover is a piece of software whose primary task is to determine whether a first-order

formula is uninformative (that is, valid). But, by making use of the relationships just listed, it is clear that it can do something else as well. Suppose we want to know whether ϕ is inconsistent. Then, using the second of the relationships listed above, we can try to establish this by giving $\neg\phi$ to the theorem prover. If the theorem prover tells us that $\neg\phi$ is uninformative, then we know that ϕ is inconsistent.

Exercise 1.2.6 In the text we gave a simple argument establishing the first of the relationships listed above, namely that ϕ is consistent if and only if $\neg\phi$ is informative. Establish the truth of the remaining three relationships.

Exercise 1.2.7 The following terminology is useful: if $\phi_1, \dots, \phi_n \models \neg\psi$, then we shall say that ψ is *inconsistent with respect to* ϕ_1, \dots, ϕ_n . Show that ψ is inconsistent with respect to ϕ_1, \dots, ϕ_n if and only if $\neg\psi$ is uninformative with respect to ϕ_1, \dots, ϕ_n . Moreover, if $\phi_1, \dots, \phi_n \not\models \neg\psi$, then we shall say that ψ is *consistent with respect to* ϕ_1, \dots, ϕ_n . Show that ψ is consistent with respect to ϕ_1, \dots, ϕ_n if and only if $\neg\psi$ is informative with respect to ϕ_1, \dots, ϕ_n .

1.3 A First-Order Model Checker

In the previous section we learned that the querying task (for finite models) is a lot simpler computationally than the consistency and informativity checking tasks. In this section we build a tool for handling querying: we write a first-order model checker in Prolog. The model checker takes the Prolog representation of a (finite) model and the Prolog representation of a first-order formula and tests whether or not the formula is satisfied in the model (there is a mechanism for assigning values to any free variables the formula contains). Our checker won't handle function symbols (this extension is left as an exercise) but it will handle equality.

We are going to provide two versions of the model checker. The first version will be (so to speak) correct so long as you're not too nasty to it. That is, as long as you are sensible about the formulas you give it (taking care, for example, only to give it formulas built over the appropriate vocabulary) it will produce the right result. But we want a robust model checker, one that is faithful to the nuances implicit in the first-order satisfaction definition. So we shall explore the limitations of the first version, and then provide a more refined version that deals with the remaining problems.

How should we implement a model checker? We have three principal tasks. First, we must decide how to represent models in Prolog. Second, we must decide how to represent first-order formulas in Prolog. Third, we must specify how (Prolog representations of) first-order formulas

are to be evaluated in (Prolog representations of) models with respect to (Prolog representations of) variable assignments. Let's turn to these tasks straight away. The Prolog representations introduced here will be used throughout the book.

Representing Models in Prolog

Suppose we have fixed our vocabulary—for example, suppose we have decided to work with this one:

```
{ (LOVE,2),
  (CUSTOMER,1),
  (ROBBER,1),
  (JULES,0),
  (VINCENT,0),
  (PUMPKIN,0),
  (HONEY-BUNNY,0)
  (YOLANDA,0) }.
```

How should we represent models of this vocabulary in Prolog? Here is an example:

```
model([d1,d2,d3,d4,d5],
      [f(0,jules,d1),
       f(0,vincent,d2),
       f(0,pumpkin,d3),
       f(0,honey_bunny,d4),
       f(0,yolanda,d5),
       f(1,customer,[d1,d2]),
       f(1,robber,[d3,d4]),
       f(2,love,[(d3,d4)])]).
```

This represents a model with a domain D containing five elements. The domain elements (d_1-d_5) are explicitly given in the list which is the first argument of `model/2`. The second argument of `model/2` is also a list. This second list specifies the interpretation function F . In particular, it tells us that d_1 is Jules, that d_2 is Vincent, that d_3 is Pumpkin, that d_4 is Honey Bunny, and that d_5 is Yolanda. It also tells us that both Jules and Vincent are customers, that both Pumpkin and Honey Bunny are robbers, and that Pumpkin loves Honey Bunny. Recall that in Section 1.1 we formally defined a model M to be an ordered pair (D, F) . As this example makes clear, our Prolog representation mirrors the form of the set-theoretic definition.

Let's look at a second example, again for the same vocabulary.

```
model([d1,d2,d3,d4,d5,d6],
      [f(0,jules,d1),
       f(0,vincent,d2),
```

```
f(0,pumpkin,d3),
f(0,honey_bunny,d4),
f(0,yolanda,d4),
f(1,customer,[d1,d2,d5,d6]),
f(1,robber,[d3,d4]),
f(2,love,[])).
```

Note that although the domain contains six elements, only four of them are named by constants: both d_5 and d_6 are nameless. However, we do know something about the anonymous d_5 and d_6 : both of them are customers (so you might like to think of this model as a situation in which Jules and Vincent are the customers of interest, and d_5 and d_6 are playing some sort of supporting role). Next, note that d_4 has two names, namely Yolanda and Honey Bunny. Finally, observe that the 2-place LOVE relation is empty: the empty list in $f(2, \text{love}, [])$ signals this. As these observations make clear, our Prolog representation of first-order models correctly embodies the nuances of the set-theoretic definition: it permits us to handle nameless and multiply-named entities, and to explicitly state that a relation is empty. So we have taken a useful step towards our goal of faithfully implementing the first-order satisfaction definition.

Exercise 1.3.1 Give the set-theoretic description of the models that the two Prolog terms given above represent.

Exercise 1.3.2 Suppose we are working with the following vocabulary:

```
{(WORKS-FOR,2),
 (BOXER,1),
 (PROBLEM-SOLVER,1),
 (THE-WOLF,0),
 (MARSELLUS,0),
 (BUTCH,0)}.
```

Represent each of the following two models over this vocabulary as Prolog terms.

1. $D = \{d_1, d_2, d_3\}$,
 $F(\text{THE-WOLF}) = d_1$,
 $F(\text{MARSELLUS}) = d_2$,
 $F(\text{BUTCH}) = d_3$,
 $F(\text{BOXER}) = \{d_3\}$,
 $F(\text{PROBLEM-SOLVER}) = \{d_1\}$
 $F(\text{WORKS-FOR}) = \emptyset$.
2. $D = \{\text{entity-1}, \text{entity-2}, \text{entity-3}\}$
 $F(\text{THE-WOLF}) = \text{entity-3}$,
 $F(\text{MARSELLUS}) = \text{entity-1}$,
 $F(\text{BUTCH}) = \text{entity-2}$,

$$\begin{aligned}F(\text{BOXER}) &= \{\text{entity-2}, \text{entity-3}\}, \\F(\text{PROBLEM-SOLVER}) &= \{\text{entity-2}\}, \\F(\text{WORKS-FOR}) &= \{(\text{entity-3}, \text{entity-1}), (\text{entity-2}, \text{entity-1})\}.\end{aligned}$$

Exercise 1.3.3 Write a Prolog program which when given a term, determines whether or not the term represents a model. That is, your program should check that the term is of the `model/2` format, that the first argument is a list containing no multiple instances of symbols, that the second argument is a list whose members are all three-place predicates with functor `f`, and so on.

Representing Formulas in Prolog

Let us now decide how to represent first-order formulas (without function symbols, but with equality) in Prolog. The first (and most fundamental) decision is how to represent first-order variables. We make the following choice: *First-order variables will be represented by Prolog variables*. The advantage of this is that it allows us to use Prolog's in-built unification mechanism to handle variables: for example, we can assign a value to a variable simply by using unification. Its disadvantage is that occasionally we will need to exercise care to block unwanted unifications—but this is a price well worth paying.

Next, we must decide how to represent the non-logical symbols. We do so in the obvious way: a first-order constant `c` will be represented by the Prolog atom `c`, and a first-order relation symbol `R` will be represented by the Prolog atom `r`.

Given this convention, it is obvious how atomic formulas should be represented. For example, `LOVE(VINCENT,MIA)` would be represented by the Prolog term `love(vincent,mia)`, and `HATE(BUTCH,X)` would be represented by `hate(butch,X)`.

Recall that there is also a special two-place relation symbol, namely the equality symbol `=`. We shall use the Prolog term `eq/2` to represent it. For example, the Prolog representation of the first-order formula `YOLANDA=HONEY-BUNNY` is the term `eq(yolanda,honey_bunny)`.

Next the booleans. The Prolog terms

`and/2` `or/2` `imp/2` `not/1`

will be used to represent the connectives \wedge , \vee , \rightarrow , and \neg respectively.

Finally, we must decide how to represent the quantifiers. Suppose ϕ is a first-order formula, and `Phi` is its representation as a Prolog term. Then $\forall x\phi$ will be represented as

`all(X,Phi)`

and $\exists x\phi$ will be represented as

`some(X,Phi).`

Note that our Prolog notation is a *prefix* notation. For example, the first-order formula

`LOVE(VINCENT,MIA) ∧ HATE(BUTCH,X)`

would be represented by

`and(love(vincent,mia),hate(butch,X)).`

Prefix notation makes life simpler as far as Prolog programming is concerned, but it must be admitted that prefix notation can be pretty hard to decipher. So we have also provided some syntactic sugar: we have defined a Prolog *infix* notation, and in this notation the previous formula would be represented by

`love(vincent,mia) & hate(butch,X).`

Appendix D contains a table describing the standard logical, prefix Prolog, and infix Prolog representations of first-order formulas. When discussing Prolog programs we shall generally work with the prefix notation (as it is the fundamental one) but when using the programs you can switch to infix representations if you prefer; you can find out how to do so in Appendix A.

The Satisfaction Definition in Prolog

We turn to the final task: evaluating (Prolog representations of) formulas in (Prolog representations of) models with respect to (Prolog representations of) assignments. The predicate which carries out the task is called `satisfy/4`, and the clauses of `satisfy/4` mirror the first-order satisfaction definition in a fairly natural way. The four arguments that `satisfy/4` takes are: the formula to be tested, the model (in the representation just introduced), a list of assignments of members of the model's domain to any free variables the formula contains, and a polarity feature (`pos` or `neg`) that tells whether a formula should be positively or negatively evaluated. Two points require immediate clarification: how are assignments to be represented, and what is the purpose of that mysterious sounding polarity feature in the fourth argument of `satisfy/4`?

Assignments are easily dealt with. We shall use Prolog terms of the form `g(Variable,Value)` to indicate that a variable `Variable` has been assigned the element `Value` of the model's domain. When we evaluate a formula, the third argument of `satisfy/4` will be a list of terms of this form, one for each of the free variables the formula contains. (In effect, we're taking advantage of Exercise 1.1.16: we're only specifying what is assigned to the free variables actually occurring in

the formula we are evaluating.) If the formula being evaluated contains no free variables (that is, if it is a sentence) the list is empty.

But what about the polarity feature in the fourth argument? The point is this. When we evaluate a formula in a model, we use the satisfaction definition to break it down into smaller subformulas, and then check these smaller subformulas in the model (for example, the satisfaction definition tells us that to check a conjunction in a model, we should check both its conjuncts in the model). Easy? Well, yes—except that as we work through the formula we may well encounter negations, and a negation is a signal that what follows has to be checked as *false* in the model. And going deeper into the negated subformula we may well encounter another negation, which means that its argument has to be evaluated as *true*, and so on and so forth, flipping backwards and forwards between *true* and *false* as we work our way down towards the atomic formulas...

Quite simply, the polarity feature is a flag that records whether we are trying to see if a particular subformula is true or false in a model. If subformula is flagged *pos* it means we are trying to see if it is true, and if it is flagged *neg* it means that we are trying to see if it is false. (When we give the original formula to the model checker, the fourth argument will be *pos*; after all, we want to see if the formula is true in the model.) The heart of the model checker is a series of clauses that spell out recursively what we need to do to check the formula as true in the model, and what we need to do to check it as false.

Let's see how this works. The easiest place to start is with the clauses of *satisfy/4* for the boolean connectives. Here are the two clauses for negation (recall that the Prolog *: -* should be read as “if”):

```
satisfy(not(Formula),Model,G,pos):-
    satisfy(Formula,Model,G,neg).

satisfy(not(Formula),Model,G,neg):-
    satisfy(Formula,Model,G,pos).
```

Obviously these clauses spell out the required flip-flops between true and false.

Now for the two clauses that deal with conjunction:

```
satisfy(and(Formula1,Formula2),Model,G,pos):-
    satisfy(Formula1,Model,G,pos),
    satisfy(Formula2,Model,G,pos).

satisfy(and(Formula1,Formula2),Model,G,neg):-
    satisfy(Formula1,Model,G,neg),
    satisfy(Formula2,Model,G,neg).
```

The first clause tells us that for a conjunction to be true in a model, both its conjuncts need to be true there. On the other hand, as the second clause tells us, for a conjunction to be false in a model, at least one of its conjuncts need to be false there (note the use of the ; symbol, Prolog's built in “or”, in the second clause). We have simply turned what the satisfaction definition tells us about conjunctions into Prolog.

Now for disjunctions:

```
satisfy(or(Formula1,Formula2),Model,G,pos) :-
    satisfy(Formula1,Model,G,pos);
    satisfy(Formula2,Model,G,pos).

satisfy(or(Formula1,Formula2),Model,G,neg) :-
    satisfy(Formula1,Model,G,neg),
    satisfy(Formula2,Model,G,neg).
```

Again, these are a direct Prolog encoding of what the satisfaction definition tells us. Note the use of Prolog's built in “or” in the first clause.

Finally, implication. We'll handle this a little differently. As we asked the reader to show in Exercise 1.2.3, the \rightarrow connective can be defined in terms of \neg and \vee : for any formulas ϕ and ψ whatsoever, $\phi \rightarrow \psi$ is logically equivalent to $\neg\phi \vee \psi$. So, when asked to check an implication, why not simply check the equivalent formula instead? After all, we already have the clauses for \neg and \vee at our disposal. And that's exactly what we'll do:

```
satisfy(imp(Formula1,Formula2),Model,G,Pos) :-
    satisfy(or(not(Formula1),Formula2),Model,G,Pos).
```

Of course, it's straightforward to give a pair of clauses (analogous to those given above for \wedge and \vee) that handle \rightarrow directly. The reader is asked to do this in Exercise 1.3.7 below.

Let's press on and see how the quantifiers are handled. Here are the clauses for the existential quantifier:

```
satisfy(some(X,Formula),model(D,F),G,pos) :-
    memberList(V,D),
    satisfy(Formula,model(D,F),[g(X,V)|G],pos).

satisfy(some(X,Formula),model(D,F),G,neg) :-
    setof(V,
        (
            memberList(V,D),
            satisfy(Formula,model(D,F),[g(X,V)|G],neg)
        ),
        ).
```

```

Dom),
setof(V,memberList(V,D),Dom) .

```

This requires some thought. Before examining the code however, what's the `memberList/2` predicate it uses? It is one of the predicates in the library `comsemPredicates.pl`. It succeeds if its first argument, any Prolog term, is a member of its second argument, which has to be a list. That is, it does exactly the same thing as the predicate usually called `member/2`. (We have added it to our library to make our code self sufficient, and renamed it to avoid problems when our libraries are used with Prologs in which `member/2` is provided as a primitive.)

But now that we know that, the first clause for the existential quantifier should be understandable. The satisfaction definition tells us that a formula of the form $\exists x\phi$ is true in a model with respect to an assignment g if there is some variant assignment g' under which ϕ is true in the model. The call `memberList(V,D)` instantiates the variable `V` to some element of the domain `D`, and then in the following line, with the instruction `[g(X,V)|G]`, we try evaluating with respect to this variant assignment. If this fails, Prolog will backtrack, the call `memberList(V,D)` will provide another instantiation (if this is still possible), and we try again. In essence, we are using Prolog backtracking to try out different variant assignments.

And with the first clause clear, the second clause becomes comprehensible. First, note that the satisfaction definition tells us that a formula of the form $\exists x\phi$ is false in a model with respect to an assignment g if ϕ is false in the model with respect to all variant assignments g' . So, just as in the first clause, we use `memberList(V,D)` and backtracking to try out different variant assignments. However, we take care not to forget what we've tried out: we use Prolog's inbuilt `setof/3` predicate to collect all the instantiations of `V` that falsify the formula. But think about it: if *all* instantiations make the formula false, then our `setof/3` will simply be the domain `D` itself. In short, obtaining `D` as the result of our `setof/3` is the signal that we really have falsified the existential formula.

To deal with the universal quantifier, we take a shortcut. Recall that $\forall x\phi$ is logically equivalent to $\neg\exists x\neg\phi$ (the reader was asked to show this in Exercise 1.2.3). So let's make use of this equivalence in the model checker:

```

satisfy(all(X,Formula),Model,G,Pol):-
    satisfy(not(some(X,not(Formula))),Model,G,Pol).

```

Needless to say, a pair of clauses which directly handle the universal quantifier could be given. Exercise 1.3.8 below asks the reader to define

them.

Let's turn to the atomic formulas. Here are the clauses for one-place predicate symbols:

```
satisfy(Formula,model(D,F),G,pos) :-  
    compose(Formula,Symbol,[Argument]),  
    i(Argument,model(D,F),G,Value),  
    memberList(f(1,Symbol,Values),F),  
    memberList(Value,Values).  
  
satisfy(Formula,model(D,F),G,neg) :-  
    compose(Formula,Symbol,[Argument]),  
    i(Argument,model(D,F),G,Value),  
    memberList(f(1,Symbol,Values),F),  
    \+ memberList(Value,Values).
```

Before discussing this, two remarks. First, note that we have again made use of `memberList/2`. Second, we have also used `compose/3`, a predicate in the library file `comsemPredicates.pl` defined as follows:

```
compose(Term,Symbol,ArgList) :-  
    Term =.. [Symbol|ArgList].
```

That is, `compose/3` uses the built in Prolog `=..` functor to flatten a term into a list of which the first member is the functor and the other members the arguments of the term. This is a useful thing to do, as we can then get at the term's internal structure using list-processing techniques; we'll see a lot of this in various guises throughout the book. Note that we use `compose/3` here to *decompose* a formula.

With these preliminaries out of the way, we can turn to the heart of the matter. It's the predicate `i/4` that does the real work. This predicate is a Prolog version of the interpretation function I_F^g . Recall that when presented with a term, I_F^g interprets it using the variable assignment g if it is a variable, and with the interpretation function F if it is a constant. And this is exactly what `i/4` does:

```
i(X,model(_,F),G,Value) :-  
    (  
        var(X),  
        memberList(g(Y,Value),G),  
        Y==X, !  
    ;  
        atom(X),  
        memberList(f(0,X,Value),F)  
    ).
```

We can now put the pieces together to see how `satisfy/4` handles atomic formulas built using one-place predicate symbols. In both the

positive and negative clauses we use `compose/3` to turn the formula into a list, and then call `i/4` to interpret the term. We then use `memberList/2` twice. The first call looks up the meaning of the one-place predicate. As for the second call, this is the only place where the positive and negative clauses differ. In the positive clause we use the call `memberList(Value,Values)` to check that the interpretation of the term is one of the possible values for the predicate in the model (thus making the atomic formula true). In the negative clause we use the call `\+memberList(Value,Values)` to check that the interpretation of the term is *not* one of the possible values for the predicate in the model, thus making the atomic formula false (recall that `\+` is Prolog's inbuilt negation as failure predicate).

The clauses for two-place predicates work pretty much the same way. Of course, we make two calls to `i/4`, one for each of the two argument terms:

```
satisfy(Formula,model(D,F),G,pos) :-
    compose(Formula,Symbol,[Arg1,Arg2]),
    i(Arg1,model(D,F),G,Value1),
    i(Arg2,model(D,F),G,Value2),
    memberList(f(2,Symbol,Values),F),
    memberList((Value1,Value2),Values).

satisfy(Formula,model(D,F),G,neg) :-
    compose(Formula,Symbol,[Arg1,Arg2]),
    i(Arg1,model(D,F),G,Value1),
    i(Arg2,model(D,F),G,Value2),
    memberList(f(2,Symbol,Values),F),
    \+ memberList((Value1,Value2),Values).
```

It only remains to discuss the clauses for equality. But given our discussion of `i/4`, the code that follows should be transparent:

```
satisfy(eq(X,Y),Model,G,pos) :-
    i(X,Model,G,Value1),
    i(Y,Model,G,Value2),
    Value1=Value2.

satisfy(eq(X,Y),Model,G,neg) :-
    i(X,Model,G,Value1),
    i(Y,Model,G,Value2),
    \+ Value1=Value2.
```

Well, that's the heart of (the first version of) our model checker. Before playing with it, let's make it a little more user-friendly. For a start, it would be pretty painful to have to type in an entire model every

time we want to make a query. We find it convenient to have a separate file containing a number of example models, and in `exampleModels.pl` you will find four. Here's the third:

```
example(3,
    model([d1,d2,d3,d4,d5,d6,d7,d8],
        [f(0,mia,d1),
         f(0,jody,d2),
         f(0,jules,d3),
         f(0,vincent,d4),
         f(1,woman,[d1,d2]),
         f(1,man,[d3,d4]),
         f(1,joke,[d5,d6]),
         f(1,episode,[d7,d8]),
         f(2,in,[(d5,d7),(d5,d8)]),
         f(2,tell,[(d1,d5),(d2,d6)]))).
```

So, let's now write a predicate which takes a formula, a numbered example model, a list of assignments to free variables, and checks the formula in the example model (with respect to the assignment) and prints a result:

```
evaluate(Formula,Example,Assignment):-
    example(Example,Model),
    satisfy(Formula,Model,Assignment,Result),
    printStatus(Result).
```

Of course, we should also test our model checker. So we shall create a test suite file called `modelCheckerTestSuite.pl` containing entries of the following form:

```
test(some(X,robber(X)),1,[],pos).
```

The first argument of `test/4` is the formula to be evaluated, the second is the example model on which it has to be evaluated (here model 1), the third is a list of assignments (here empty) to free variables in the formula, and the fourth records whether the formula is satisfied or not (`pos` indicates it should be satisfied).

Giving the command

```
?- modelCheckerTestSuite.
```

will force Prolog to evaluate all the examples in the test suite, and print out the results. The output will be a series of entries of the following form:

```
Input formula:
1 some(A, robber(A))
Example Model: 1
Status: Satisfied in model.
```

Model Checker says: Satisfied in model.

The fourth line of output (`Status`) is the information (recorded in the test suite) that the formula should be satisfied. The fifth line (`Model Checker says`) shows that the model checker got this particular example right.

We won't discuss the code for `modelCheckerTestSuite/0`. It's essentially a `fail/0` driven iteration through the test suite file, that uses the `evaluate/3` predicate defined above to perform the actual model checking. You can find the code in `modelChecker1.pl`.

Exercise 1.3.4 Systematically test the model checker on these models. If you need inspiration, use `modelCheckerTestSuite/0` to run the test suite and examine the output carefully. As you will see, this version of the model checker does not handle all the examples in the way the test suite demands. Try to work out why not.

Exercise 1.3.5 If you did the previous exercise, you will have seen that the model checker sometimes prints out the message `Satisfied in model` more than once. Why is this? Is this a useful feature or not?

Exercise 1.3.6 Try the model checker on example model 1 with the examples:

```
and(some(X, customer(X)), some(Y, robber(Y)))
some(X, and(customer(X), some(Y, robber(Y))))
and(some(X, customer(X)), some(X, robber(X)))
some(X, and(customer(X), some(X, robber(X)))).
```

The model checker handles all four example correctly: that is, it says that all four examples are satisfied in model 1. Fine—but *why* is it handling them correctly? In particular, in the last two examples we reuse the variable `X`, binding different occurrences to different quantifiers. What is it about the code that lets the model checker know that the `X` in `customer(X)` is intended to play a different role than the `X` in `robber(X)`?

Exercise 1.3.7 Our model checker handles \rightarrow by rewriting $\phi \rightarrow \psi$ as $\neg\phi \vee \psi$. Provide clauses for \rightarrow (analogous to those given in the text for \wedge and \vee) that directly mirror what the satisfaction definition tells us about this connective.

Exercise 1.3.8 Our model checker handles \forall by rewriting $\forall x\phi$ as $\neg\exists x\neg\phi$. Provide clauses for \forall (analogous to those given in the text for \exists) that directly mirror what the satisfaction definition tells us about this quantifier.

Programs for the model checker

modelChecker1.pl

The main file containing the code for the model checker for first-order logic. Consult this file to run the model checker—it will load all other files it requires.

comsemPredicates.pl

Definitions of auxiliary predicates.

exampleModels.pl

Contains example models in various vocabularies.

modelCheckerTestSuite.pl

Contains formulas to be evaluated on the example models.

Refining the Model Checker

Our first model checker (`modelChecker1.pl`) is a reasonably well-behaved tool that is faithful to the main ideas of the first-order satisfaction definition. To put it another way: it is a piece of software for handling the querying task. Actually, it does rather more than handle the querying task: we can also use it to find elements in a model's domain that satisfy certain descriptions. Consider the following query:

```
?- satisfy(robber(X),
           model([d1,d2,d3],
                 [f(0,pumpkin,d1),
                  f(0,jules,d2),
                  f(1,custom,[d2]),
                  f(1,robber,[d1,d3])]),
           [g(X,Value)],
           pos).

Value = d1 ? ;

Value = d3 ? ;

no
```

Note the assignment list `[g(X,Value)]` used in the query: `X` is given the value `Value`, and this is a Prolog *variable*. So Prolog is forced to search for an instantiation when evaluating the query, and via backtracking

finds all instantiations for `Value` that satisfy the formula (namely d_1 and d_3). This is a useful technique, as we will shall see in Chapter 6 when we use it in a simple question answering system.

Nonetheless, although well behaved, this version of the model checker is not sufficiently robust, nor does it always return the correct answer. However, its weak points are rather subtle, so let's sit back and think our way through the following problems carefully.

First Problem Consider the following queries:

```
?- evaluate(X,1).  
?- evaluate(imp(X,Y),4).
```

Now, as the first query asks the model checker to evaluate a Prolog variable, and the second asks it to evaluate a ‘formula’, whose subformulas are Prolog variables, you may think that these examples are too silly to worry about. But let’s face it, these are the type of queries a Prolog programmer might be tempted to make (perhaps hoping to generate a formula that is satisfied in model 1). And (like any other) query, they should be handled gracefully—but they’re not. Instead they send Prolog into an infinite loop and you will probably get a stack overflow message (if you don’t understand why this happens, do Exercise 1.3.9 right away). This is unacceptable, and needs to be fixed.

Second Problem Here’s a closely related problem. Consider the following queries:

```
?- evaluate(mia,3).  
?- evaluate(all(mia,vincent),2).
```

Now, obviously these are not sensible queries: constants are not formulas, and cannot be checked in models. But we have the right to expect that our model checker responds correctly to these queries—and it *doesn’t*. Instead, our basic model checker returns the message `no` (that is, not satisfied in the given model) to both queries.

Why is this wrong? *Because neither expression is the sort of entity that can enter into a satisfaction relation with a model.* Neither is in the “satisfied” relation with any model, nor in the “not satisfied” relation either. They’re simply not formulas. So the model checker should come back with a different message that signals this, something like `Cannot be evaluated`. And of course, if the model checker is to produce such a message, it needs to be able to detect when its input is a legitimate formula. The next problem pins down what is required more precisely.

Third Problem We run into problems if we ask our model checker to verify formulas built from items that don't belong to the vocabulary. For example, suppose we try evaluating the atomic formula

```
tasty(royale_with_cheese)
```

in any of the example models. Then our basic model checker will say `Not satisfied in model`. This response is incorrect: the satisfaction relation is not defined between formulas and models of different vocabularies. Rather our model checker should throw out this formula and say something like "Hey, I don't know anything about these symbols!". So, not only does the model checker need to be able to verify that its input is a formula (as we know from the second problem), it also needs to be able to verify that it's a formula built over a vocabulary that is appropriate for the model.

Fourth Problem Suppose we make the following query:

```
?- evaluate(customer(X),1).
```

Our model checker will answer `Not satisfied in model`. This is wrong: `X` is a free variable, we have not assigned a value to it (recall that `evaluate/2` evaluates with respect to the empty list of assignments), and hence the satisfaction relation is not defined. So our model checker should answer `Cannot be evaluated`, or something similar.

Well, those are the main problems, and they can be fixed. In fact, `modelChecker2.pl` handles such examples correctly, and produces appropriate messages. We won't discuss the code of `modelChecker2.pl`, but we will ask the reader to do the following exercises. They will enable you better understand where `modelChecker1.pl` goes wrong, and how `modelChecker2.pl` fixes matters up. Happy hacking!

Exercise 1.3.9 Why does our basic model checker get in an infinite loop when given the following queries:

```
?- evaluate(X,1).
```

```
?- evaluate(imp(X,Y),4).
```

And what happens and why with examples like `some(X,or(customer(X),Z))` and `some(X,or(Z,customer(X)))`. If the answers are not apparent from the code, carry out traces (in most Prolog shells, the trace mode can be activated by "`?- trace.`" and deactivated by "`?- notrace.`").

Exercise 1.3.10 What happens if you use the basic model checker to evaluate constants as formulas, and why? If this is unclear, perform a trace.

Exercise 1.3.11 Modify the basic model checker so that it classifies the kinds of examples examined in the previous two exercises as ill-formed input.

Exercise 1.3.12 Try out the new model checker (`modelChecker2.pl`) on the formulas `customer(X)` and `not(customer(X))`. Why has problem 4 vanished?

Programs for the refined model checker

`modelChecker2.pl`

This file contains the code for the revised model checker for first-order logic. This version rejects ill-formed formulas and handles a number of other problems. Consult this file to run the model checker—it will load all other files it requires.

`comsemPredicates.pl`

Definitions of auxiliary predicates.

`exampleModels.pl`

Contains example models in various vocabularies.

`modelCheckerTestSuite.pl`

Contains formulas to be evaluated on the example models.

1.4 First-Order Logic and Natural Language

By this stage, the reader should have a reasonable grasp of the syntax and semantics of first-order logic, so it is time to raise a more basic question: just how good is first-order logic as a tool for exploring natural language semantics computationally? We shall approach this question from two directions. First we consider what first-order logic can offer the study of inference, and then we consider its representational capabilities.

Inferential Capabilities

First-order logic is sometimes called classical logic, and it deserves this name: it is the most widely studied and best understood of all logical formalisms. Moreover, it is understood from a wide range of perspectives. For example, the discipline called model theory (which takes as its starting point the satisfaction definition discussed in this chapter) has mapped the expressive power of first-order logic in extraordinary mathematical detail.

Nor have the inferential properties of first-order logic been neglected: the research areas of proof theory and automated reasoning both explore this topic. As we mentioned earlier (and as we shall further discuss in Chapters 4 and 5), the primary task that faces us when dealing with the consistency and informativity checking tasks (which we defined model-theoretically) is to reformulate them as concrete symbol manipulation tasks. Proof theorists and researchers in automated reasoning have devised many ways of doing this, and have explored the advantages, disadvantages, and interrelationships between the various methods in detail.

Some of the methods they have developed (notably the *tableau* and *resolution* methods discussed in Chapters 4 and 5) have proved useful computationally. Although no complete computational solution to the consistency checking task (or equivalently, the informativity checking task) exists, resolution and tableau-based theorem provers for first-order logic have proved effective in practice, and in recent years there have been dramatic improvements in their performance. Moreover, more recently, the automated reasoning community has started to develop tools called *model builders*, and (as we shall see in Chapters 5 and 6) these newer tools are also relevant to our inference tasks. So one important reason for computational semanticists to be interested in first-order logic is simply this: if you use first-order logic, a wide range of sophisticated inference tools lies at your disposal.

But there are other reasons for being interested in first-order logic. One of the most important is this: working with first-order logic makes it straightforward to incorporate *background knowledge* into the inference process.

Here's an example. When discussing informativity we gave the following example of an uninformative-but-sometimes-acceptable discourse:

Mia is married. She has a husband.

But *why* is this uninformative? Recall that by an uninformative formula we mean a valid formula (that is, one that is satisfied in every model). But

$\text{MARRIED}(\text{MIA}) \rightarrow \text{HAS-HUSBAND}(\text{MIA})$

is *not* valid. Why not? Because we are free to interpret MARRIED and HAS-HUSBAND so that they have no connection with each other, and in some of these interpretations the formula will be false.

But it is clear that such arbitrary interpretations of MARRIED and HAS-HUSBAND are somehow cheating. After all, as any speaker of English knows, the meanings of MARRIED and HAS-HUSBAND are inti-

mately linked. But can we capture this linkage, and can first-order logic help?

It can, and here's how. Speakers of English have the following knowledge, expressed here as a formula of first-order logic:

$$\forall x (\text{WOMAN}(x) \wedge \text{MARRIED}(x) \rightarrow \text{HAS-HUSBAND}(x)).$$

If we take this knowledge, and add to it the information that Mia is a woman, then we *do* have a valid inference: the two premises

$$\forall x (\text{WOMAN}(x) \wedge \text{MARRIED}(x) \rightarrow \text{HAS-HUSBAND}(x))$$

and

$$\text{WOMAN}(\text{MIA})$$

have as a logical consequence that

$$\text{MARRIED}(\text{MIA}) \rightarrow \text{HAS-HUSBAND}(\text{MIA}).$$

That is, in any model where the premises are true (and these are the models of interest, for they are the ones that reflect what we know) then the conclusion is true also. In short, the un informativity of our little discourse actually rests on an inference that draws on background knowledge. Modelling the discourse in first-order logic makes this inferential interplay explicit.

There are other logics besides first-order logic which are interesting from an inferential perspective, such as the family of logics variously known as modal, hybrid, and description logics. If you work with the simplest form of such logics (that is, the propositional versions of such logics) the consistency and informativity checking tasks typically *can* be fully computed. Moreover, the description logic community has produced an impressive range of efficient computational tools for dealing with these (and other) inferential tasks. And such logics have been successfully used to tackle problems in computational semantics (in fact, as various AI formalisms such as semantic nets are essentially forerunner of modern description logics, there is actually a very long history of applying such logics in computational semantics, albeit sometimes in disguised form).

But there is a price to pay. An interesting thing about first-order logic (as we shall shortly see) is that it has enough expressive power to handle a significant portion of natural language semantics. Propositional modal, hybrid and description logics don't have the expressive power for detailed semantic analyses. They are likely to play an increasingly important role in computational semantics, but they are probably best suited to giving efficient solutions to relatively specialised tasks. It is possible that stronger versions of such logics (in particular, logics which combine the resources of modal, hybrid, or description logic

with those of first-order logic) may turn out to be a good general setting for semantic analysis, but at present few computational tools for performing inference in such systems are available.

Summing up, if you are interested in exploring the role of inference in natural language semantics, then first-order logic is probably the most interesting starting point. Moreover, the key first-order inference techniques (such as tableaus and resolution) are not parochial. Although initially developed for first-order logic, they have been successfully adapted to many other logics. So studying first-order inference is a useful first step towards understanding inference in other logics.

But what is first-order logic like from a representational perspective? This is the question to which we now turn.

Representational Capabilities

Assessing the representational capabilities of first-order logic for natural language semantics is not straightforward: it would be easy to write a whole chapter (indeed, a whole book) on the topic. But, roughly speaking, our views are as follows. First-order logic does have its shortcomings when it comes to representing the meaning of natural language expressions, and we shall draw attention to an important example. Nonetheless, first-order logic is capable of doing an awful lot of work for us. It's not called classical logic for nothing: it is an extremely flexible tool.

It is common to come across complaints about first-order logic of the following kind: first-order logic is inadequate for natural language semantics because it cannot handle times (or intervals, or events, or modal phenomena, or epistemic states, or ...). Take such complaints with a (large) grain of salt. They are often wrong, and it is important to understand why.

The key lies in the notion of the model. We have encouraged the reader to think of models as mathematical idealisations of situations, but while this is a useful intuition pump, it is less than the whole truth. The full story is far simpler: models can provide abstract pictures of just about *anything*. Let's look at a small example—a model which represents the way someone's emotional state evolves over three days. Let $D = \{d_1, d_2, d_3, d_4\}$ and let F be as follows:

$$\begin{aligned} F(\text{MIA}) &= d_1 \\ F(\text{MONDAY}) &= d_2 \\ F(\text{TUESDAY}) &= d_3 \\ F(\text{WEDNESDAY}) &= d_4 \\ F(\text{PERSON}) &= \{d_1\} \\ F(\text{DAY}) &= \{d_2, d_3, d_4\} \end{aligned}$$

$$F(\text{PRECEDE}) = \{(d_2, d_3), (d_3, d_4), (d_2, d_4)\}$$

$$F(\text{HAPPY}) = \{(d_1, d_2), (d_1, d_4)\}.$$

That is, there are four entities in this model: one of them (d_1) is a person called Mia, and three of them (d_2 , d_3 , and d_4) are the days of the week Monday, Tuesday and Wednesday. The model also tells us that (as we would expect) Monday precedes Tuesday, Tuesday precedes Wednesday, and Monday precedes Wednesday. Finally, it tells us that Mia was happy on both Monday and Wednesday (presumably something unpleasant happened on Tuesday). In short, the last clause of F spells out how Mia's emotional state evolves over time.

Though simple, this example illustrates something crucial: models can provide pictures of virtually anything we find interesting. Do you feel that the analysis of tense and aspect in natural language requires time to be thought of as a collection of intervals, linked by such relations as overlap, inclusion, and precedence? Very well then—work with models containing intervals related in these ways. Moreover, though you could use some specialised interval-based temporal logic to talk about these structures, you are also free (should you so desire) to talk about these models using a first-order language of appropriate vocabulary. Or perhaps you feel that temporal semantics requires the use of events? Very well then—work with models containing interrelated events. And once again, you can (if you wish) talk about these models using a first-order language of appropriate vocabulary. Or perhaps you feel that the possible world semantics introduced by philosophical logicians to analyse such concepts as necessity and belief is a promising way of handling these aspects of natural language. Very well then—work with models containing possible worlds linked in the ways you find useful. And while you *could* talk about such models with some sort of modal language, you are also free to talk about these models using a first-order language of appropriate vocabulary. Models and their associated first-order languages are a playground, not a jail. They provide a space where we are free to experiment with our ideas and ontological assumptions as we attempt to come to grips with natural language semantics. The major limitations are those imposed by our imagination.

Moreover, clever use of models sometimes enables first-order logic to provide useful approximate solutions to demands for expressive power that, strictly speaking, it cannot handle. A classic example is second-order (and more generally, higher-order) quantification. First-order logic is called “first-order” because it only permits us to quantify over *first-order entities*; that is, the elements of the model’s domain. It does not let us quantify over *second-order entities* such as sets of domain

elements (properties), or sets of pairs of domain elements (two-place relations), sets of triples of domain elements (three-place relations), and so on. Second-order logic allows us to do all this. For example, in second-order logic we can express such sentences as Butch has every property that a good boxer has by means of the following expression:

$$\forall P \forall x ((\text{GOOD-BOXER}(x) \rightarrow P(x)) \rightarrow P(\text{BUTCH})).$$

Here P is a second-order variable ranging over properties (that is, subsets of the domain) while x is an ordinary first-order variable. Hence this formula says: “for any property P and any individual x , if x is a good boxer implies that x has property P , then Butch has property P too”.

So, have we finally found an unambiguous example of something that first-order logic cannot do? Surprisingly, no. There *is* a way in which first-order logic can come to grips with second-order entities—and once again, it’s models that come to the rescue. Quite simply, if we want to quantify over properties, why not introduce models containing domain elements that play the role of properties? That is, why not introduce first-order entities whose job it is to mimic second-order entities? After all, we’ve just seen that domain elements can be thought of as times (and intervals, events, possible worlds, and so on); maybe we can view them as properties (and two-place relations, and three-place relations, and so on) as well?

In fact, it has been known since the 1950s that this can be done in an elegant way: it is possible to give a first-order interpretation to second-order logic (and more generally, to higher-order logic). Moreover, in some respects this first-order treatment of second-order logic is better behaved than the standard interpretation. In particular, both model-theoretically and proof-theoretically, the first-order perspective on second-order quantification leads to a simpler and better behaved logic than the standard perspective does. Now, it may be that the first-order analysis of second-order quantification isn’t as strong as it should be (on the other hand, it could also be argued that the standard analysis of second-order quantification is too strong). But the fact remains that the first-order perspective allows us to get to grips with some aspects of second-order logic. It may be an approximation, but it is an interesting one. For further information on this topic, consult the references cited in the Notes at the end of the chapter.

It’s not hard to give further examples of how clever use of models enables us to handle demands for expressive power which at first glance seem to lie beyond the grasp of first-order logic. For example, first-order logic offers an elegant handle on *partiality*; that is, sentences which are

neither true nor false but rather *undefined* in some situation (see the Notes at the end of the chapter for further information). But instead of multiplying examples of what first-order logic can do, let's look at something it can't.

Natural languages are rich in quantifiers: as well as being able to say things like **some robbers** and **every robber**, we can also say things like **two robbers**, **three robbers**, **most robbers**, **many robbers** and **few robbers**. Some of these quantifiers (notably counting quantifiers such as **two** and **three**) can be handled by first-order logic; others, however, provably can't. For example, if we take **most** to mean “more than half the entities in the model”, which seems a reasonable interpretation, then even if we restrict our attention to finite models, it is impossible to express this idea in first-order logic (see the Notes for references). Thus a full analysis of natural language semantics will require richer logics capable of handling such *generalized quantifiers*. But while a lot is known about the theory of such quantifiers (see the Notes), as yet few computational tools are available, so we won't consider generalized quantifiers further in this book.

So, first-order logic isn't capable of expressing everything of interest to natural language semantics: generalized quantifiers are an important counterexample. Nonetheless, a surprisingly wide range of other important phenomena can be given a first-order treatment. All in all, it's an excellent starting point for computational semantics.

Notes

There are many good introductions to first-order logic, and the best advice we can give the reader is to spend some time browsing in a library to see what's on offer. That said, there are three references we particularly recommend. For an unhurried introduction that motivates the subject linguistically, try the first volume of Gamut (1991a). For a wide-ranging discursive overview, try Hodges (1983). This survey article covers a lot of ground—from propositional logic to Lindström's celebrated characterization of first-order logic. But although parts of the article cover advanced material, the clarity of the writing should enable most readers to follow. The reader who wants a more focused, traditional approach could try Enderton (2001). This is a solid and well-written introduction to first-order logic; among other things it will give you a firm grasp of the mathematical underpinnings of inductive definitions, and a taste of model theory.

As we said in the text, logicians traditionally wouldn't regard the querying task as an inference task. But there's another group of re-

searchers who certainly would view it this way, namely computer scientists. Model checking has become important in computer science in recent years. The classic application is hardware verification; that is, checking whether a piece of hardware, such as a computer chip, really works the way it is intended to. The chip design is represented as a (very large) model. (Remember what we said in the text? Models can be used to represent just about *anything!*) The properties the chip is supposed to possess is represented as a formula. Then the formula is evaluated in the model. If the formula is false, there is a problem with the chip design, which hopefully can be tracked down and rectified. However, computer scientist don't use first-order logic in such applications; specialised temporal logics, or modal logics equipped with fixed-point operators, are used instead. For an informal introduction to model checking, with lots of interesting motivation, try Halpern and Vardi (1991). For a more detailed introduction, try Chapter 3 of Huth and Ryan (2000).

Another group of computer scientists who would probably agree that querying is best viewed as a form of inference are researchers in *database theory*. As we said in the text, if you think of a model as a database, and a first-order language as a database query language, you will not go far wrong. For an excellent introduction to database theory that adopts this logical perspective, see Abiteboul et al. (1995). The basic difference between work in databases and the more traditional perspective adopted in the text is that database researchers are typically interested in working with restricted fragments of first-order logic in which querying can be efficiently implemented.

The mention of efficiency brings us to an important point. We remarked that the querying task was far easier than the informativity or consistency checking tasks. This is true, but the querying task is by no means computationally straightforward: in fact it is a PSPACE-complete problem (see Vardi (1982)). This means that on some input it may take time exponential in the size of the formula (that is, the number of symbols the formula contains) to verify whether or not it is satisfied. Complexity theorists believe that PSPACE-complete problems are even harder than NP-complete problems, so even our easiest inference task turns out to be highly non-trivial. (Complexity theorists classify problems as being in NP if they can be solved in polynomial time on a non-deterministic machine. NP-complete problems are the most difficult problems in NP. See Papadimitriou (1994) for formal definitions and discussion of these complexity classes.)

To our surprise, while preparing these Notes we were only able to find one other implementation of model checking for full first-order

logic, namely the checker due to Jan Jaspars which you can find at

<http://staff.science.uva.nl/~jaspars/animations/>

Many highly sophisticated model checkers exist, but such checkers are typically for various forms of modal or temporal logic, or are database query languages (in effect, model checkers for various fragments of first-order logic). It would be interesting to know of other implementations of the full first-order satisfaction definition.

In the text we remarked that the informativity and consistency checking tasks are inter-definable (that is, if you can solve one, you can solve the other) and that both tasks are in fact *undecidable*. You can find proofs of this fundamental result in Enderton (2001) and Boolos and Jeffrey (1989).

Readers with a background in philosophy of language or pragmatics will recognise that our account of informativity echoes ideas of Grice (1975). It was Grice who first emphasized the importance of informativity in communication, and his ideas have influenced our discussion. But a caveat should be made. What is informative is highly dependent on context. In the account given in the text, in essence we identified “informative” with “being a logical consequence of the information at an agent’s disposal”. But we really need to be more subtle here. Conversational agents are not usually aware of all the consequences of their beliefs, so our approach will for many purposes be too strong. A more detailed theory, which gives a fine-grained account of how agents update their beliefs, is required; for discussion of some of the issues involved see Cherniak (1986), Harman (1986) and Wassermann (1999).

The querying task, and the consistency and informativity checking tasks, rest on different assumptions about the way that semantic content is to be stored computationally: the querying task assumes that content is stored as a model, whereas the consistency and informativity checking tasks assume that it is stored as a (collection of) formula(s). In a real application it might well be useful to use both methods. We haven’t discussed this possibility here, but that is simply because we wanted to introduce the individual tasks as clearly as possible in their own terms. In Chapter 6, when we discuss the Helpful Curt program, we’ll see that such mixed methods can be useful: Helpful Curt makes use of querying and consistency checking to provide answers to questions posed by the user.

While our account has emphasized first-order logic, we also mentioned that logic from the modal/hybrid/description family of logics may turn out to be useful to computational semantics. For an introduction to modal logic, see Blackburn et al. (2001b). For hybrid logic,

see Blackburn (2000). For description logic, see the excellent *Handbook of Description Logic* (Baader et al., 2003). One of the articles in this collection, Franconi (2002), surveys applications of description logic in natural language. Recent papers not covered by this survey include Gardent and Jacquay (2003), Koller et al. (2004), and Gardent and Striegnitz (2003).

We mentioned that nowadays it is uncontroversial to view modal and temporal phenomena from a first-order perspective. For a classic statement of the first-order perspective on temporal phenomena, see Van Benthem (1991). The essence of the first-order perspective is that many so-called ‘non-classical’ logics (such as modal logic and temporal logic) are often just a different notation for a restricted fragment of first-order logic. And other fragments of first-order logic may well turn out to be important to computational semantics. For example, Pratt-Hartmann (2003) shows that a surprising amount can be done with the two-variable fragment of first-order logic. This is the fragment of first-order logic consisting of all formulas containing not more than two variables; its informativity and consistency checking problems are both decidable.

The quasi-reduction of second-order logic to first-order logic has been known for more than 50 years: it was introduced in Henkin (1950), a classic of mathematical logic. For a clear introduction to this topic, see Chapter 4 of Enderton (2001). For more advanced material, see Doets and Van Benthem (1983) and Shapiro (1999). For information on partial logic, and reductions of partial logic to first-order logic, see Langholm (1988) and Muskens (1996).

As an example of a phenomenon in natural language that lies beyond the scope of first-order logic, we mentioned the generalized quantifier most. The proof that the meaning of most cannot be captured in first-order logic can be found in Barwise and Cooper (1981); this classic paper contains other examples in a similar vein, and is well worth reading. For an overview of modern generalized quantifier theory, see Keenan and Westerståhl (1997). Another natural language phenomenon which is typically viewed as requiring more than the capabilities of first-order logic is the semantics of plural noun phrases. Actually, the trade-offs involved here between first- and second-order expressivity are subtle, and lie well beyond the scope of this book; we refer the reader to Lønning (1997) for a detailed account.

As we mentioned in the text, the ideas that variable assignment functions can be viewed as a mechanism for representing context has proved important to natural language semantics. The pioneering work was carried out by Hans Kamp in his Discourse Representation Theory

(see Kamp (1984) and Kamp and Reyle (1993)) and Irene Heim in her File Change Semantics (see Heim (1982)). These theories rethought the way that quantification (and in particular, existential quantification) could be logically modelled. This work inspired the development of Dynamic Predicate Logic (see Groenendijk and Stokhof (1991)) in which first-order formulas were viewed as programs that modify assignment functions. For a detailed discussion of these developments, see Van Eijck and Kamp (1997) and Muskens et al. (1997).

One final comment may be helpful for some readers. In the text we carefully stated that we had a countably infinite collection of first-order variables at our disposal, and we also made a number of statements to the effect that most models were infinite. Lying behind these comments is the fundamental set-theoretical result that there is an infinite hierarchy of ever bigger infinities. If you have never encountered set-theory before, this claim may well strike you as bizarre, but in fact the ideas involved are not particularly difficult, and any decent text book on set-theory will explain the concepts involved (Enderton (1977) is a good choice). Incidentally, if you haven't previously encountered the idea of thinking of properties as subsets, and binary relations as ordered pairs, and so on, then this reference is a good place to find out more.

2

Lambda Calculus

Now that we know something of first-order logic and how to work with it in Prolog, it is time to turn to the first major theme of the book, namely:

How can we automate the process of associating semantic representations with natural language expressions?

In this chapter we explore the issue concretely. We proceed by trial and error. We first write a simple Prolog program that performs the task in a limited way. We note where it goes wrong, and why, and develop a more sophisticated alternative. These experiments lead us, swiftly and directly, to formulate a version of the *lambda calculus*. The lambda calculus is a tool for controlling the process of making substitutions. With its help, we will be able to describe, neatly and concisely, how semantic representations should be built. The lambda calculus is one of the main tools used in this book, and by the end of the chapter the reader should have a reasonable grasp of why it is useful to computational semantics and how to work with it in Prolog.

2.1 Compositionality

Given a sentence of English, is there a systematic way of constructing its semantic representation? This question is far too general, so let's ask a more specific one: is there a systematic way of translating simple sentences such as **Vincent loves Mia** and **A woman snorts** into first-order logic?

The key to answering this is to be more precise about what we mean by “systematic”. Consider **Vincent loves Mia**. Its semantic content is at least partially captured by the first-order formula $\text{LOVE}(\text{VINCENT}, \text{MIA})$. Now, the most basic observation we can make about systematicity is the following: the proper name **Vincent** contributes the constant **VINCENT** to the representation, the transitive verb **loves** contributes the relation

symbol LOVE, and Mia contributes MIA.

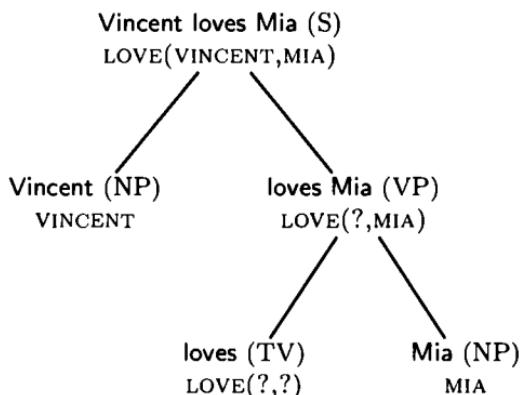
This first observation is important, and by no means as trivial as it may seem. If we generalise it to the claim that the words making up a sentence contribute *all* the bits and pieces needed to build the sentence's semantic representation, we have formulated a principle that is a valuable guide to the complexities of natural language semantics. The principle is certainly plausible. Moreover, it has the advantage of forcing us to face up to a number of non-trivial issues sooner rather than later (for example, what exactly does the determiner every contribute to the representation of Every woman loves a boxer?).

Nonetheless, though important, this principle doesn't tell us everything we need to know about systematicity. For example, from the symbols LOVE, MIA and VINCENT we can also form LOVE(MIA,VINCENT). Why don't we get this (incorrect) representation when we translate Vincent loves Mia? What exactly is it about the sentence Vincent loves Mia that forces us to translate it as LOVE(VINCENT,MIA)? Note that the answer "But Vincent loves Mia means LOVE(VINCENT,MIA), stupid!", which in many circumstances would be appropriate, isn't particularly helpful here. Computers *are* stupid. We can't appeal to their semantic insight, because they don't have any. If we are to have any hope of automating semantic construction, we must find another kind of answer.

The missing ingredient is a notion of *syntactic structure*. Vincent loves Mia isn't just a string of words: it has a hierarchical structure. In particular, Vincent loves Mia is a sentence (S) that is composed of the subject noun phrase (NP) Vincent and the verb phrase (VP) loves Mia. This VP is in turn composed of the transitive verb (TV) loves and the direct object NP Mia.

Given this hierarchy, it is easy to tell a coherent story—and indeed (see below) draw a convincing picture—about why we should get the representation LOVE(VINCENT,MIA), and not anything else. Why is MIA in the second argument slot of LOVE? Because, when we combine a TV with an NP to form a VP, we have to put the semantic representation associated with the NP (in this case, MIA) in the *second* argument slot of the VP's semantic representation (in this case, LOVE(?,{?})). Why does VINCENT have to go in the *first* argument slot? Because this is the slot reserved for the semantic representations of NPs that we combine with VPs to form an S. More generally, given that we have some reasonable syntactic story about what the pieces of the sentences are, and which pieces combine with which other pieces, we can try to use this information to explain how the various semantic contributions have to be combined. In short, one reasonable explication of "systematicity" is that it amounts to using the additional information provided by syn-

tactic structure to spell out exactly how the semantic contributions are to be glued together.



Our discussion has led us to one of the key concepts of contemporary semantic theory: *compositionality*. Suppose we have some sort of theory of syntactic structure. It doesn't matter too much what sort of theory it is, just so long as it is hierarchical in a way that eventually leads to the lexical items. (That is, our notion of syntactic structure should allow us to classify the sentence into subparts, sub-subparts, and sub-sub-subparts, . . . , and so on—ultimately into the individual words making up the sentence.) Such structures make it possible to tell an elegant story about where semantic representations come from. Ultimately, semantic information flows from the lexicon, thus each lexical item is associated with a representation. How is this information combined? By making use of the hierarchy provided by the syntactic analysis. Suppose the syntax tells us that some kind of sentential subpart (a VP, say) is decomposable into two sub-subparts (a TV and an NP, say). Then our task is to describe how the semantic representation of the VP subpart is to be built out of the representation of its two sub-subparts. If we succeed in doing this for all the grammatical constructions covered by the syntax, we will have given a *compositional semantics* for the language under discussion (or at least, for that fragment of the language covered by our syntactic analysis).

Natural Language Syntax via Definite Clause Grammars

So, is there a systematic way of translating simple sentences such as *Vincent loves Mia* and *A woman snorts* into first-order logic? We don't yet have a method, but at least we now have a plausible strategy for finding one. We need to:

Task 1 Specify a reasonable syntax for the fragment of natural language of interest.

Task 2 Specify semantic representations for the lexical items.

Task 3 Specify the translation compositionally. That is, we should specify the translation of all expressions in terms of the translation of their parts, where “parts” refers to the substructure given to us by the syntax.

Moreover, all three tasks need to be carried out in a way that leads naturally to computational implementation.

As this is a book on computational semantics, tasks 2 and 3 are where our real interests lie, and most of our energy will be devoted to them. But since compositional semantics presupposes syntax, we need a way of handling task 1. What should we do?

We have opted for a particularly simple solution: in this book the syntactic analysis of a sentence will be a tree whose non-leaf nodes represent *complex syntactic categories* (such as sentence, noun phrase and verb phrase) and whose leaves represent *lexical items* (these are associated with *basic syntactic categories* such as noun, transitive verb, determiner, proper name and intransitive verb). The tree the reader has just seen is a typical example. This approach has an obvious drawback (namely, the reader won’t learn anything interesting about syntax) but it also has an important advantage: we will be able to make use of Definite Clause Grammars (DCGs), the in-built Prolog mechanism for grammar specification and parsing.

Here is a DCG for the fragment of English we shall use in our initial semantic construction experiments. (This DCG, decorated with Prolog code for semantic operations, can be found in the files `experiment1.pl`, `experiment2.pl`, and `experiment3.pl`.)

<code>s --> np, vp.</code>	<code>noun --> [woman].</code>
<code>np --> pn.</code>	<code>noun --> [foot, massage].</code>
<code>np --> det, noun.</code>	<code>vp --> iv.</code>
<code>pn --> [vincent].</code>	<code>vp --> tv, np.</code>
<code>pn --> [mia].</code>	<code>iv --> [snorts].</code>
<code>det --> [a].</code>	<code>iv --> [walks].</code>
<code>det --> [every].</code>	<code>tv --> [loves].</code>
	<code>tv --> [likes].</code>

This grammar tells us how to build certain kinds of sentences (`s`) out of noun phrases (`np`), verb phrases (`vp`), proper names (`pn`), determiners (`det`), nouns (`noun`), intransitive verbs (`iv`), and transitive verbs (`tv`), and gives us a tiny lexicon to play with. For example, the grammar accepts the simple sentence

`Vincent walks.`

because `Vincent` is declared as a proper name, and proper names are noun phrases according to this grammar; `walks` is an intransitive verb, and hence a verb phrase; and sentences can consist of a noun phrase followed by a verb phrase.

But the real joy of DCGs is that they provide us with a lot more than a natural notation for specifying grammars. Because they are part and parcel of Prolog, we can actually compute with them. For example, by posing the query

```
s([mia,likes,a,foot,massage],[])
```

we can test whether `Mia likes a foot massage` is accepted by the grammar, and the query

```
s(X,[])
```

generates all grammatical sentences.

With a little effort, we can do a lot more. In particular, by making use of the DCG extra argument mechanism (consult an introductory text on Prolog if you're unsure what this means) we can associate semantic representations with lexical items very straightforwardly. The normal Prolog unification mechanism then gives us the basic tool needed to combine semantic representations, and to pass them up towards sentence level. In short, working with DCGs both frees us from having to implement parsers, and makes available a powerful mechanism for combining representations, so we'll be able to devote our attention to semantic construction.

It is worth emphasising, however, that the semantic construction methods discussed in this book are compatible with a wide range of theories of natural language syntax, and with a wide range of programming languages. In essence, we exploit the recursive structure of trees to build representations compositionally: where the trees actually come from is relatively unimportant, as is the programming language used to encode the construction process. We have chosen to fill in the syntactical component using Prolog DCGs—but a wide range of options is available and we urge our readers to experiment.

Exercise 2.1.1 How many sentences are accepted by this grammar? How many noun phrases? How many verb phrases? Check your answers by generating the relevant items.

2.2 Two Experiments

How can we systematically associate first-order formulas with the sentences produced by our little grammar? Let's just plunge in and try, and see how far our knowledge of DCGs and Prolog will take us.

Experiment 1

First the lexical items. We need to associate **Vincent** with the constant **VINCENT**, **Mia** with the constant **MIA**, **walks** with the unary relation symbol **WALK**, and **loves** with the binary relation symbol **LOVE**. The following piece of DCG code makes these associations. Note that the arity of **walk** and **love** are explicitly included as part of the Prolog representation.

```
pn(vincent) --> [vincent].
pn(mia) --> [mia].
iv(snort(_)) --> [snorts].
tv(love(_, _)) --> [loves].
```

How do we build semantic representations for sentences? Let's first consider how to build representations for quantifier-free sentences such as **Mia loves Vincent**. The main problem is to steer the constants into the correct slots of the relation symbol. (Remember, we want **Vincent loves Mia** to be represented by **LOVE(VINCENT,MIA)**, not **LOVE(MIA,VINCENT)**.) Here's a first (rather naive) attempt. Let's directly encode the idea that when we combine a TV with an NP to form a VP, we have to put the semantic representation associated with the NP in the second argument slot of the VP's semantic representation, and that we use the first argument slot for the semantic representations of NPs that we combine with VPs to form Ss.

Prolog has a built in predicate **arg/3** such that **arg(N,P,I)** is true if I is the Nth argument of P. This is a useful tool for manipulating pieces of syntax, and with its help we can cope with simple quantifier free sentences rather easily. Here's the needed extension of the DCG:

```
s(Sem) --> np(SemNP), vp(Sem),
  {
    arg(1, Sem, SemNP)
  }.

np(Sem) --> pn(Sem).

vp(Sem) --> tv(Sem), np(SemNP),
  {
    arg(2, Sem, SemNP)
  }.

vp(Sem) --> iv(Sem).
```

These clauses work by adding an extra argument to the DCG (here, the position filled by the variables **Sem** and **SemNP**) to percolate up the required semantic information using Prolog unification. Note that while

the second and the fourth clauses perform only this percolation task, the first and third clauses, which deal with branching rules, have more work to do: they use `arg/3` to steer the arguments into the correct slots. This is done by associating extra pieces of code with the DCG rules, namely `arg(1,Sem,SemNP)` and `arg(2,Sem,SemNP)`. (These are normal Prolog goals, and are added to the DCG rules in curly brackets to make them distinguishable from the grammar symbols.) This program captures, in a brutally direct way, the idea that the semantic contribution of the object NP goes into the second argument slot of TVs, while the semantic contribution of subject NPs belongs in the first argument slot.

It works. For example, if we pose the query:

```
?- s(Sem,[mia,snorts],[]).
```

we obtain the (correct) response:

```
Sem = snort(mia).
```

But this is far too easy—let's try to extend our fragment with the determiners `a` and `every`. First, we need to extend the lexical entries for these words, and the entries for the common nouns they combine with:

```
det(some(_,and(_,_)))--> [a].
det(all(_,imp(_,_)))--> [every].
noun(woman(_))--> [woman].
noun(footmassage(_))--> [foot,massage].
```

NPs formed by combining a determiner with a noun are called *quantified noun phrases*.

Next, we need to say how the semantic contributions of determiners and noun phrases should be combined. We can do this by using `arg/3` four times to get the instantiation of the different argument positions correct:

```
np(Sem)--> det(Sem), noun(SemNoun),
{
  arg(1,SemNoun,X),
  arg(1,Sem,X),
  arg(2,Sem,Matrix),
  arg(1,Matrix,SemNoun)
}.
```

The key idea is that the representation associated with the NP will be the representation associated with the determiner (note that the `Sem` variable is shared between `np` and `det`), but with this representation fleshed out with additional information from the noun. The Prolog variable `X` is a name for the existentially quantified variable

the determiner introduces into the semantic representation; the code `arg(1, SemNoun, X)` and `arg(1, Sem, X)` unifies the argument place of the noun with this variable. The code `arg(2, Sem, Matrix)` simply says that the second argument of `Sem` will be the matrix of the NP semantic representation, and more detail is added by `arg(1, Matrix, SemNoun)`: it says that the first slot of the matrix will be filled in by the semantic representation of the noun. So if we pose the query

```
?- np(Sem, [a,woman], []).
```

we obtain the response

```
Sem = some(X, and(woman(X), Y)).
```

Note that the output is an *incomplete* first-order formula. We don't yet have a full first-order formula (the Prolog variable `Y` has yet to be instantiated) but we do know that we are existentially quantifying over the set of women.

Given that such incomplete first-order formulas are the semantic representations associated with quantified NPs, it is fairly clear what must happen when we combine a quantified NP with a VP to form an S: the VP must provide the missing piece of information. (That is, it must provide an instantiation for `Y`.) The following clause does this:

```
s(Sem) --> np(Sem), vp(SemVP),
{
    arg(1, SemVP, X),
    arg(1, Sem, X),
    arg(2, Sem, Matrix),
    arg(2, Matrix, SemVP)
}.
```

Unfortunately, while the underlying idea is essentially correct, things have just started going badly wrong. Until now, we've simply been extending the rules of our original DCG with semantic information—and we've already dealt with `s --> np, vp`. If we add this second version of `s --> np, vp` (and it seems we need to) we are duplicating syntactic information. This is uneconomical and inelegant. Worse, this second sentential rule interacts in an unintended way with the rule

```
np(Sem) --> pn(Sem).
```

As the reader should check, as well as assigning the correct semantic representation to `A woman snorts`, our DCG also assigns the splendidly useless string of symbols

```
snort(some(X, and(woman(X), Y))).
```

But this isn't the end of our troubles. We already have a rule for forming VPs out of TVs and NPs, but we will need a second rule to

cope with quantified NPs in object position, namely:

```
vp(Sem) --> tv(SemTV), np(Sem),
{
    arg(2,SemTV,X),
    arg(1,Sem,X),
    arg(2,Sem,Matrix),
    arg(2,Matrix,SemTV)
}.
```

If we add this rule, we can assign correct representations to all the sentences in our fragment. However we will also produce a lot of nonsense (for example, *A woman loves a foot massage* is assigned four representations, three of which are just jumbles of symbols) and we are being systematically forced into syntactically unmotivated duplication of rules. This doesn't look promising. Let's try something else.

Exercise 2.2.1 The code for experiment 1 is in `experiment1.pl`. Generate the semantic representations of the sentences and noun phrases yielded by the grammar.

Experiment 2

Although our first experiment was ultimately unsuccessful, it did teach us something useful: to build representations, we need to work with *incomplete* first-order formulas, and we need a way of manipulating the missing information. Consider the representations associated with determiners. In experiment 1 we associated `a` with `some(_,and(_,_,_))`. That is, the determiner contributes the skeleton of a first-order formula whose first slot needs to be instantiated with a variable, whose second slot needs to be filled with the semantic representation of a noun, and whose third slot needs to be filled by the semantic representation of a VP. However in experiment 1 we didn't manipulate this missing information directly. Instead we took a shortcut: we thought in terms of argument *position* so that we could make use of `arg/3`. Let's avoid plausible looking shortcuts. The idea of missing information is evidently important, so let's take care to always associate it with an explicit Prolog variable. Perhaps this direct approach will make semantic construction easier.

Let's first apply this idea to the determiners. We shall need three extra arguments: one for the bound variable, one for the contribution made by the noun, and one for the contribution made by the VP. Incidentally, these last two contributions have a standard name: the contribution made by the noun is called the *restriction* and the contribution made by the VP is called the *nuclear scope*. We reflect this terminology

in our choice of variable names:

```
det(X,Restr,Scope,some(X, and(Restr,Scope)))--> [a].  
det(X,Restr,Scope,all(X, imp(Restr,Scope)))--> [every].
```

But the same idea applies to common nouns, intransitive verbs, and transitive verbs too. For example, instead of associating `woman` with `WOMAN(_)`, we should state that the translation of `woman` is `WOMAN(y)` for some particular choice of variable `y`—and we should *explicitly* keep track of which variable we choose. (That is, although the `y` appears free in `WOMAN(y)`, we actually want to have some sort of hold on it.) Similarly, we want to associate a transitive verb like `loves` with `LOVE(y,z)` for some particular choice of variables `y` and `z`, and again, we should keep track of the choices we made. So the following lexical entries are called for:

```
noun(X,woman(X))--> [woman].  
iv(Y,snort(Y))--> [snorts].  
tv(Y,Z,love(Y,Z))--> [loves].
```

Given these changes, we need to redefine the rules for producing sentences and verb phrases.

```
s(Sem)--> np(X,SemVP,Sem), vp(X,SemVP).  
vp(X,Sem)--> tv(X,Y,SemTV), np(Y,SemTV,Sem).  
vp(X,Sem)--> iv(X,Sem).
```

The semantic construction rule associated with the sentential rule, for example, tells us that `Sem`, the semantic representation of the sentence, is essentially going to be that of the noun phrase (that's where the value of the `Sem` variable will trickle up from) but that, in addition, the bound variable `X` used in `Sem` must be the same as the variable used in the verb phrase semantic representation `SemVP`. Moreover, it tells us that `SemVP` will be used to fill in the information missing from the semantic representation of the noun phrase.

So far so good, but now we need a little trickery. Experiment 1 failed because there was no obvious way of making use of the semantic representations supplied by quantified noun phrases and proper names in a single sentential rule. Here we only have a single sentential rule, so evidently the methods of experiment 2 avoid this problem. Here's how it's done:

```
np(X,Scope,Sem)--> det(X,Restr,Scope,Sem), noun(X,Restr).  
np(SemPN,Sem,Sem)--> pn(SemPN).
```

Note that we have given all noun phrases—regardless of whether they are quantifying phrases or proper names—the same arity in the grammar rules. (That is, there really is only *one* `np` predicate in this

grammar, not two predicates that happen to make use of the same atom as their functor name.)

It should be clear how the first rule works. The skeleton of a quantified noun phrase is provided by the determiner. The restriction of this determiner is filled by the noun. The resultant noun phrase representation is thus a skeleton with two marked slots: X marks the bound variable, while Scope marks the missing nuclear scope information. This Scope variable will be instantiated by the verb phrase representation when the sentential rule is applied.

The second rule is trickier. The easiest way to understand it is to consider what happens when we combine a proper name noun phrase with a verb phrase to form a sentence. Recall that this is our sentential rule:

$$\text{s}(\text{Sem}) \rightarrow \text{np}(X, \text{SemVP}, \text{Sem}), \quad \text{vp}(X, \text{SemVP}).$$

Hence when we have a noun phrase of the form $\text{np}(\text{SemPN}, \text{Sem}, \text{Sem})$, two things happen. First, because of the doubled Sem variable, the semantic representation of the verb phrase (that is, SemVP) becomes the semantic representation associated with the sentence. Secondly, because X is unified with SemPN , the semantics of the proper name (a first-order constant) is substituted into the verb phrase representation. So the doubled Sem variable performs a sort of role reversal. Intuitively, whereas the representation for sentences that have a quantified noun phrase as subject is essentially the subject's representation filled out by the verb phrase representation, the reverse is the case when we have a proper name as subject. In such cases, the verb phrase is boss. The sentence representation is essentially the verb phrase representation, and the role of the proper name is simply to obediently instantiate the empty slot in the verb phrase representation.

Our second experiment has fared far better than our first. It is clearly a good idea to explicitly mark missing information; this gives us the control required to fill it in and manoeuvre it into place. Nonetheless, experiment 2 uses the idea clumsily. Much of the work is done by the rules. These state how semantic information is to be combined, and (as our NP rule for proper names shows) this may require rule-specific Prolog tricks such as variable doubling. Moreover, it is hard to think about the resulting grammar in a modular way. For example, when we explained why the NP rules took the form they do, we did so by explaining what was eventually going to happen when the S rule was used. Now, perhaps we weren't *forced* to do this—nonetheless, we find it difficult to give an intuitive explanation of this rule any other way.

This suggests we are missing something. Maybe a more disciplined

approach to missing information would reduce—or even eliminate—the need for rule-specific combination methods? Indeed, this exactly what happens if we make use of the *lambda calculus*.

Exercise 2.2.2 Using either pen and paper or a tracer, compare the sequence of variable instantiations this program performs when building representations for *Vincent snorts* and *A woman snorts*, and *Vincent loves Mia* and *Vincent loves a woman*.

Programs for the first two experiments

`experiment1.pl`

The code of our first experiment in semantic construction for a small fragment of English.

`experiment2.pl`

The second experiment in semantic construction.

2.3 The Lambda Calculus

We shall view lambda calculus as a notational extension of first-order logic that allows us to bind variables using a new variable binding operator λ . Occurrences of variables bound by λ should be thought of as placeholders for missing information: they *explicitly* mark where we should substitute the various bits and pieces obtained in the course of semantic construction. An operation called β -conversion performs the required substitutions. We suggest that the reader thinks of the lambda calculus as a special programming language dedicated to a single task: gluing together the items needed to build semantic representations.

Here is a simple lambda expression:

$\lambda x. \text{MAN}(x)$.

The prefix $\lambda x.$ binds the occurrence of x in $\text{MAN}(x)$. We often say that the prefix $\lambda x.$ *abstracts over* the variable x . We call expressions with such prefixes *lambda abstractions* (or, more simply, *abstractions*). In our example, the binding of the free x variable in $\text{MAN}(x)$ explicitly indicates that MAN has an argument slot where we may perform substitutions. More generally, the purpose of abstracting over variables is to mark the slots where we want substitutions to be made.

We use the symbol @ to indicate the substitutions we wish to carry out. Consider the following example:

$\lambda x.\text{MAN}(x)@\text{VINCENT}.$

This compound expression consists of the abstraction $\lambda x.\text{MAN}(x)$ and the constant VINCENT glued together by the @ symbol (that is, we use infix notation for the @-operator). Such expressions are called *functional applications*; the left-hand expression is called the *functor*, and the right-hand expression the *argument*. We often say that the functor is *applied* to its argument: for example, the expression $\lambda x.\text{MAN}(x)@\text{VINCENT}$ is the application of the functor $\lambda x.\text{MAN}(x)$ to the argument VINCENT.

But what is such an expression good for? It is an instruction to throw away the $\lambda x.$ prefix of the functor, and to replace every occurrence of x that was bound by this prefix by the argument; we shall call this substitution process β -conversion (other common names include β -reduction and λ -conversion). Performing the β -conversion specified by the previous expression yields:

$\text{MAN}(\text{VINCENT}).$

Abstraction, functional application, and β -conversion underly much of our subsequent work. In fact, as we shall soon see, the business of specifying semantic representations for lexical items is essentially going to boil down to devising lambda abstractions that specify the missing information, while functional application coupled with β -conversion will be the engine used to combine semantic representations compositionally.

The previous example was rather simple, and in some respects rather misleading. For a start, the argument was simply the constant VINCENT, but (as we shall soon see) arguments can be complex expressions containing occurrences of λ and @. Moreover, the $\lambda x.$ in $\lambda x.\text{MAN}(x)$ is simply used to mark the missing term in the predicate MAN, but as experiments 1 and 2 made clear, to deal with noun phrases and determiners (and indeed, many other things) we need to mark more complex kinds of missing information. However, λ will be used for such tasks too. For example, our semantic representation of the noun phrase a woman will be: $\lambda y.\exists x(\text{WOMAN}(x) \wedge y@x).$ Here we are using the variable y to indicate that some information is missing (namely, the nuclear scope, to use the linguistic terminology mentioned earlier) and to show exactly where this information has to be plugged in when it is found (it will be applied to the argument x and conjoined to the formula WOMAN(x)).

We are almost ready to examine some linguistic examples, but before doing so an important point needs to be made: the lambda expressions

$\lambda x.\text{MAN}(x)$, $\lambda u.\text{MAN}(u)$, and $\lambda v.\text{MAN}(v)$ are intended to be equivalent, and so are $\lambda u.\exists x(\text{WOMAN}(x) \wedge u@x)$ and $\lambda v.\exists x(\text{WOMAN}(x) \wedge v@x)$. All these expressions are functors which when applied to an argument A , replace the bound variable by the argument. No matter which argument A we choose, the result of applying any of the first three expressions to A and then β -converting should be $\text{MAN}(A)$, and the result of applying either of the last two expressions to A should be $\exists x(\text{WOMAN}(x) \wedge A@x)$. To put it another way, replacing the bound variables in a lambda expression (for example, replacing the variable x in $\lambda x.\text{MAN}(x)$ by u to obtain $\lambda u.\text{MAN}(u)$) is a process which yields a lambda expression which is capable of performing exactly the same gluing tasks.

This shouldn't be surprising—it's the way bound variables always work. For example, we saw in Chapter 1 that $\forall x\text{ROBBER}(x)$ and $\forall y\text{ROBBER}(y)$ mean exactly the same thing, and mathematics students will be used to relabelling variables when calculating integrals. A bound variable is essentially a placeholder: the particular variable used is not intended to have any significance. For this reason bound variables are sometimes called dummy variables.

The process of relabelling bound variables (any bound variable: it doesn't matter whether it is bound by \forall or \exists or λ) is called α -conversion. If a lambda expression E can be obtained from a lambda expression E' by α -conversion then we say that E and E' are α -equivalent (or that they are *alphabetic variants*). Thus $\lambda x.\text{MAN}(x)$, $\lambda y.\text{MAN}(y)$, and $\lambda z.\text{MAN}(z)$ are all α -equivalent, and so are the expressions $\lambda y.\exists x(\text{WOMAN}(x) \wedge y@x)$ and $\lambda z.\exists y(\text{WOMAN}(y) \wedge z@y)$. In what follows we often treat α -equivalent expressions as if they were identical. For example, we will sometimes say that the lexical entry for some word is a lambda expression E , but when we actually work out some semantic construction, we might use an α -equivalent expression E' instead of E itself. As λ -bound variables are merely placeholders, this clearly should be allowed. But the reader needs to understand that it's not merely *permissible* to work like this, it can be *vital* to do so if β -conversion is to work as intended.

Why? Well, suppose that the expression F in $\lambda x.F$ is a complex expression containing many occurrences of λ , \forall and \exists . It could happen that when we apply $\lambda x.F$ to an argument A , some occurrence of a variable that is free in A becomes bound by a lambda operator or a quantifier when we substitute it into F . Later in the chapter we'll see a concrete example—for now we'll simply say that *we don't want this to happen*. Such accidental bindings (as they are usually called) defeat the purpose of working with the lambda calculus. The whole point of developing the lambda calculus was to gain control over the process

of performing substitutions. We don't want to lose control by foolishly allowing unintended interactions.

And such interactions need never be a problem. We don't need to use $\lambda x.F$ as the functor: any α -equivalent expression will do. By relabelling all the bound variables in $\lambda x.F$ we can always obtain an α -equivalent functor that doesn't bind any of the variables that occur free in A , and accidental binding is prevented. Thus, strictly speaking, it is not merely functional application coupled with β -conversion that drives the process of semantic construction in this book, but functional application and β -conversion coupled with α -conversion.

That's all we need to know about the lambda calculus for the time being—though it is worth mentioning that the lambda calculus can be introduced from a different, more logically oriented, perspective. The logical perspective is important (we discuss it briefly in the Notes at the end of the chapter) nonetheless it is *not* the only legitimate perspective on lambda calculus. Indeed, as far as computational semantics is concerned, it is the computational perspective we have adopted here—*lambda calculus as glue language*—that makes the lambda calculus interesting. So let's ignore the logical perspective for now and try putting our glue language to work. Here's a good place to start: does lambda calculus solve the problem we started with? That is, does it get rid of the difficulties we encountered in experiments 1 and 2?

Let's see. What is involved in building the semantic representation for every boxer walks using lambdas? The first step is to assign lambda expressions to the different basic syntactic categories. We assign the determiner **every**, the common noun **boxer**, and the intransitive verb **walks** the following lambda expressions:

every: $\lambda u.\lambda v.\forall x(u@x \rightarrow v@x)$

boxer: $\lambda y.\text{BOXER}(y)$

walks: $\lambda z.\text{WALK}(z)$.

Before going further, pause a moment. These expressions should remind you of the representations we used in our experiments. For example, in experiment 2 we gave the determiner **every** the representation

`det(X, Restr, Scope, all(X, imp(Restr, Scope))).`

If we use the Prolog variable **U** instead of **Restr**, and **V** instead of **Scope** this becomes

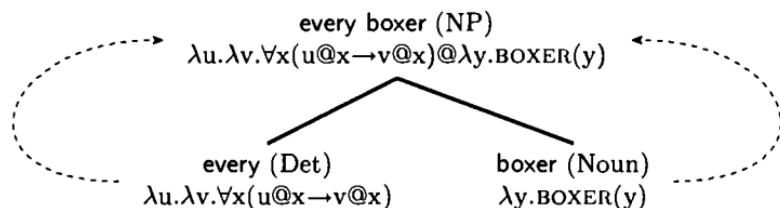
`det(X, U, V, all(X, imp(U, V)))`

which is quite similar to $\lambda u.\lambda v.\forall x(u@x \rightarrow v@x)$.

But there are also important differences. The experiment 2 representation is a Prolog-specific encoding of missing information. In contrast,

lambda expressions are programming language independent (we could work with them in Java, C++ or Haskell, for example). Moreover, experiment 2 ‘solved’ the problem of combining missing information on a rule-by-rule basis. As will soon be clear, functional application and β -conversion provide a completely general solution to this problem.

Let’s return to every boxer walks. According to our grammar, a determiner and a common noun can combine to form a noun phrase. Our semantic analysis couldn’t be simpler: we will associate the NP node with the functional application that has the determiner representation as functor and the noun representation as argument.



Now, applications are instructions to carry out β -conversion, so let’s do what is required. (Note that as there are no free occurrences of variables in the argument expression, there is no risk of accidental variable capture, so we don’t need to α -convert the functor.) Performing the demanded substitution yields:

every boxer: $\lambda v. \forall x(\lambda y.BOXER(y) @ x \rightarrow v @ x)$.

But this expression contains the subexpression $\lambda y.BOXER(y) @ x$. Now, the argument x is a free variable, but x does not occur bound in the functor $\lambda y.BOXER(y)$, so we don’t need to α -convert the functor. So we can simply go ahead and perform the required β -conversion, and when we do so we obtain:

every boxer: $\lambda v. \forall x(BXER(x) \rightarrow v @ x)$.

We can’t perform any more β -conversions, so let’s carry on with the analysis of the sentence. Now, we know that we want the S node to be associated with $\forall x(BXER(x) \rightarrow WALK(x))$. How do we get this representation?

We obtain it by a procedure analogous to the one just performed at the NP node. First, we associate the S node with the application that has the NP representation just obtained as functor, and the VP representation as argument:

every boxer walks: $\lambda v. \forall x(BXER(x) \rightarrow v @ x) @ \lambda z.WALK(z)$.

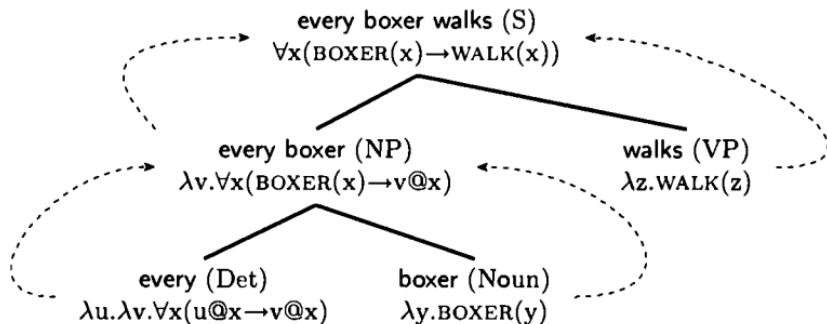
Performing β -conversion yields:

every boxer walks: $\forall x(BXER(x) \rightarrow \lambda z.WALK(z) @ x)$.

We can then perform β -conversion on the expression $\lambda z.\text{WALK}(z)@x$, and when we do so we obtain the desired representation:

every boxer walks: $\forall x(\text{BOXER}(x) \rightarrow \text{WALK}(x))$,

which is what we wanted. Here is the entire semantic construction process in diagrammatic format (with the β -conversions already carried out):



It is worth reflecting on this example, for it shows that in two important respects semantic construction is getting simpler. First, the process of combining two representations is now uniform: we simply say which of the representations is the functor and which the argument, whereupon combination is carried out by applying functor to argument and β -converting. Second, more of the load of semantic analysis is now carried by the lexicon: it is here that we use the lambda calculus to make the missing information stipulations.

Are there clouds on the horizon? For example, while the semantic representation of a quantifying noun phrase such as *a woman* can be used as a functor, surely the semantic representation of an NP like *Vincent* will have to be used as an argument? We avoided this problem in experiment 2 by the variable doubling trick used in the NP rule for proper names—but that was a Prolog specific approach, incompatible with the use of lambda calculus. Maybe—horrible thought!—we’re going to be forced to duplicate syntactic rules again, just as we did in experiment 1.

In fact, there’s no problem at all. The lambda calculus offers a delightfully simple functorial representation for proper names, as the following examples show:

Mia: $\lambda u.(u@\text{MIA})$

Vincent: $\lambda u.(u@\text{VINCENT})$.

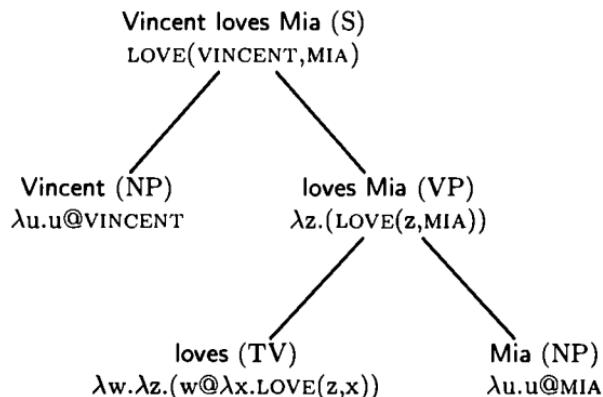
These representations are abstractions, thus they can be used as functors. However note what such functors do: they are instructions

to substitute their argument in the slot marked u , which amounts to applying their own arguments to themselves! Because the lambda calculus offers us the means to specify such role-reversing functors, the spectre of syntactic rule duplication vanishes.

As an example of these new representations in action, let us build a representation for *Vincent loves Mia*. We shall assume that TV semantic representations take their object NP's semantic representation as argument, so we assign loves the following lambda expression:

$$\lambda w. \lambda z. (w @ \lambda x. \text{LOVE}(z, x)).$$

And as in the previous example, the subject NP's semantic representation takes the VP's semantic representation as argument, so we can build the following tree:



Let's sum up what we have achieved. Our decision to move beyond the approach of experiment 2 to the more disciplined approach of the lambda calculus was sensible. For a start, we don't need to spend any more time thinking about how to combine two semantic representations—functional application and β -conversion give us a general mechanism for doing so. Moreover, much of the real work is now being done at the lexical level; indeed, even the bothersome problem of finding a decent way of handling NP representations uniformly now has a simple lexical solution.

In fact, for the remainder of this book, the following version of the three tasks listed earlier will underly our approach to semantic construction:

Task 1 Specify a DCG for the fragment of natural language of interest.

Task 2 Specify semantic representations for the lexical items with the help of the lambda calculus.

Task 3 Specify the semantic representation \mathcal{R}' of a syntactic item \mathcal{R} whose parts are \mathcal{F} and \mathcal{A} with the help of functional application.

That is, specify which of the subparts is to be thought of as functor (here it's \mathcal{F}), which as argument (here it's \mathcal{A}) and then define \mathcal{R}' to be $\mathcal{F}' @ \mathcal{A}'$, where \mathcal{F}' is the semantic representation of \mathcal{F} and \mathcal{A}' is the semantic representation of \mathcal{A} .

We must now show that the second and third tasks lend themselves naturally to computational implementation.

Exercise 2.3.1 Work in detail through the functional applications and β -conversions required to build the semantic representations for the VP and the S in the tree for *Vincent loves Mia*. Make sure you fully understand the semantic representation used for TVs.

2.4 Implementing Lambda Calculus

Our decision to perform semantic construction with the aid of an abstract glue language (the lambda calculus) has pleasant consequences for grammar writing. But how can we make the approach computational?

The answer is clear: we should wrap the key combinatorial mechanisms (functional application, β -conversion, and α -conversion) into a simple black box. When thinking about semantics we should be free to concentrate on the interesting issues—we shouldn't have to worry about the mechanics of gluing semantic representations together.

In this section we build such a black box. We first show how to represent lambda abstractions and functional applications in Prolog, we then write (a first version of) a β -conversion predicate, and finally we write an α -conversion predicate. The required black box is simply the β -conversion predicate modified to call the α -conversion predicate before carrying out reductions.

Lambda expressions in Prolog

Let's get to work. First we have to decide how to represent a lambda abstraction $\lambda x. \mathcal{E}$ in Prolog. We do so as follows:

`lam(X,E).`

That is, we use the Prolog functor `lam/2` instead of λ , the Prolog variable `X` instead of `x`, and write `E` (the Prolog representation of \mathcal{E}) in the second argument place of `lam`.

Next we have to decide how to represent functional application $\mathcal{F} @ \mathcal{A}$ in Prolog. We do so as follows:

`app(F,A).`

That is, we use the Prolog functor `app/2` instead of $@$, write `F` (the Prolog representation of \mathcal{F}) as its first argument, and `A` (the Prolog

representation of \mathcal{A}) as its second argument. (So our fundamental Prolog notation for functional application is a prefix notation. But, as with first-order notation, we have provided a Prolog infix notation too. Appendix D gives the details. When using the programs you can toggle to infix notation if you find it more readable; Appendix A tells you how.)

With these notational decisions taken, we have all we need to start writing DCGs that build lambda expressions. Let's first see what the syntactic rules will look like. (This following DCG is part of the file `experiment3.pl`.) Actually, there's practically nothing that needs to be said here. If we use the syntactic rules in the manner suggested by (our new version of) task 3, all we need is the following:

```
s(app(NP,VP))--> np(NP), vp(VP).
np(PN)--> pn(PN).
np(app(Det,Noun))--> det(Det), noun(Noun).
vp(IV)--> iv(IV).
vp(app(TV,NP))--> tv(TV), np(NP).
```

Note that the unary branching rules just percolate up their semantic representation (here coded as Prolog variables PN, IV and so on), while the binary branching rules use the `app/2` term to build semantic representations out of the component semantic representations in the manner suggested by task 3. Compared with experiments 1 and 2, this is completely transparent: we simply apply function to argument to get the desired result.

The real work is done at the lexical level. The entries for nouns and intransitive verbs practically write themselves:

```
noun(lam(X,footmassage(X)))--> [foot,massage].
noun(lam(X,woman(X)))--> [woman].
iv(lam(X,walk(X)))--> [walks].
```

And here's the code stating that $\lambda x.(x@\text{VINCENT})$ is the translation of Vincent, and $\lambda x.(x@\text{MIA})$ the translation of Mia:

```
pn(lam(X,app(X,vincent)))--> [vincent].
pn(lam(X,app(X,mia)))--> [mia].
```

Next, recall that the lambda expressions for the determiners `every` and `a` are $\lambda u.\lambda v.\forall x(u@x \rightarrow v@x)$ and $\lambda u.\lambda v.\exists x(u@x \wedge v@x)$. We express these in Prolog as follows.

```
det(lam(U, lam(V, all(X, imp(app(U,X), app(V,X))))))--> [every].
det(lam(U, lam(V, some(X, and(app(U,X), app(V,X))))))--> [a].
```

And now we have enough to start experimenting with semantic construction: we simply pass the semantic information up the tree from the lexicon, and `app/2` explicitly records how it all fits together. Here

is an example query:

```
?- s(Sem,[mia,snorts],[]).
Sem = app(lam(_A,app(_A,mia)),lam(_B,snort(_B)))
```

So far so good—but we’re certainly not finished yet. While the output is correct, we don’t want to produce representations crammed full of lambdas and applications: we want genuine first-order formulas. So we need a way of getting rid of the glue after it has served its purpose. Let’s start developing the tools needed to do this.

Implementing β -conversion

The crucial tool we need is a predicate that carries out β -conversion. For example, when given

```
app(lam(U,app(U,mia)),lam(X,snort(X)))
```

as input (this is the rather messy expression just produced by our DCG), it should carry out the required β -conversions to produce

```
snort(mia),
```

which is the first-order expression we really want. But how can we define such a predicate?

The crucial idea is to make use of a *stack* that keeps track of all expressions that need to be used as arguments at some point. (A stack is a data structure that stores information in a last-in/first-out manner: the last item pushed onto the stack will be the first item popped off it. Prolog lists can naturally be thought of as stacks, the head of the list being the top of the stack). Let’s look at some examples. Here’s what should happen when we β -convert `app(lam(X,smoke(X)),mia)`:

Expression	Stack
app(lam(X,smoke(X)),mia)	[]
lam(X,smoke(X))	[mia]
smoke(mia)	[]

As this example shows, at the start of the β -conversion process the stack is empty. When the lambda expression we are working with is an application (as in the first line), its argument is pushed onto the stack and the outermost occurrence of `app/2` is discarded. When the formula we are working with is an abstraction (as in the second line) the initial lambda is thrown away, and the item at the top of the stack (here `mia`) is popped and substituted for the newly-freed variable (here `X`). If the expression we are working with is neither an application nor an abstraction (as in the third line) we halt.

Now, the previous example obviously works correctly, but it is so simple you might be tempted to think that it is overkill to use a stack. But it's not. Consider this example:

Expression	Stack
app(app(lam(Z, lam(Y, invite(Y, Z))), mia), vincent)	[]
app(lam(Z, lam(Y, invite(Y, Z))), mia)	[vincent]
lam(Z, lam(Y, invite(Y, Z)))	[mia, vincent]
lam(Y, invite(Y, mia))	[vincent]
invite(vincent, mia)	[]

Note the way the last-in/first-out discipline of the stack ensures that the arguments are correctly substituted. In this example, `vincent` is the argument of the outermost functional application. But we *can't* substitute `vincent` at this stage as the functor expression

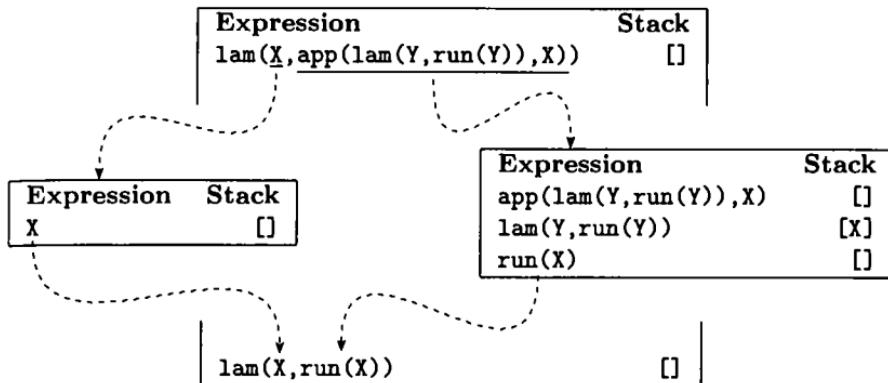
```
app(lam(Z, lam(Y, invite(Y, Z))), mia)
```

is not an abstraction. Instead, we must first β -convert this functor (this involves substituting `mia` into the `Z` slot). As we see in line four, once this has been done the functor is in the correct syntactic form (that is, its outermost operator is now a lambda) and so we can substitute `vincent` for `Y`. In short, the stack keeps track of which functor corresponds to which argument.

Here's another example. This shows what happens when the semantic representation of a proper name is applied to a complex argument:

Expression	Stack
app(lam(U, app(U, mia)), lam(X, smoke(X)))	[]
lam(U, app(U, mia))	[lam(X, smoke(X))]
app(lam(X, smoke(X)), mia)	[]
lam(X, smoke(X))	[mia]
smoke(mia)	[]

Finally, what are we to do when the expression we need to β -convert is *not* an application? This means we cannot push anything onto the stack, and thus we are blocked—but there may well be subexpressions that can be β -converted. The answer is clear: we simply split the expression into its component parts, and recursively apply our stack-based algorithm to both components. When the recursion bottoms out, we put the pieces back together. Here's an example:



Well, that's the idea—let's now turn it into Prolog. Here's the base clause:

```
betaConvert(X, Y, []):-  
    var(X),  
    Y=X.
```

Why is this correct? As we have just seen, functors are not always in the required syntactic form (they don't always have λ as the outermost operator), so we sometimes need to β -convert the functor first. Bearing this in mind, consider the base clause. Variables cannot be further reduced, so the result of β -converting a variable is the variable itself. Moreover, the stack must be empty at this point, for a non-empty stack would only make sense if the expression to be reduced was a lambda-abstraction, which a variable simply isn't.

Now for the second clause. If we're not dealing with a variable, and if the expression is an application (that is, of the form `app(Fun, Arg)`), then we push the argument onto the stack and go on with reducing the functor. (Of course, we only do this if the functor is not a variable, as variables cannot be further reduced.) The following code does this:

```
betaConvert(Expression, Result, Stack):-  
    nonvar(Expression),  
    Expression = app(Functor, Argument),  
    nonvar(Functor),  
    betaConvert(Functor, Result, [Argument|Stack]).
```

Now for the third clause. Whereas an application pushes an element on the stack, an abstraction `lam(X, Formula)` pops an element off the stack and substitutes it for `X`. We implement the substitution process by unifying `X` with the element at the top of the stack. Note that this clause can only be entered if the stack is non-empty: the use of `[X|Stack]` as the third argument guarantees that there is at least one element on the

stack (and the choice of the Prolog variable X to mark the top of the stack forces the desired unification):

```
betaConvert(Expression,Result,[X|Stack]) :-  
    nonvar(Expression),  
    Expression = lam(X,Formula),  
    betaConvert(Formula,Result,Stack).
```

Finally, we need a clause that deals with other kinds of complex expression (such as conjunctions) and with lambda-abstractions when the stack is empty. In such cases we use `compose/3` to break down the complex expression into a list of subexpressions, carry out β -conversion on all elements of this list using `betaConvertList/3`, and then re-assemble using `compose/3`:

```
betaConvert(Expression,Result,[]):-  
    nonvar(Expression),  
    \+ (Expression = app(X,_), nonvar(X)),  
    compose(Expression,Functor,SubExpressions),  
    betaConvertList(SubExpressions,ResultSubExpressions),  
    compose(Result,Functor,ResultSubExpressions).
```

The definition of `betaConvertList/3` is the obvious recursion:

```
betaConvertList([],[]).
```

```
betaConvertList([Expression|Others],[Result|Results]) :-  
    betaConvert(Expression,Result),  
    betaConvertList(Others,Results).
```

Last of all, here's a predicate which initialises the β -conversion process with an empty stack:

```
betaConvert(X,Y) :-  
    betaConvert(X,Y,[]).
```

Let's try out the `s(Sem,[mia,snorts],[])` query again—except this time, we'll feed the output into our new β -conversion predicate:

```
?- s(Sem,[mia,snorts],[]), betaConvert(Sem,Reduced).
```

```
Sem = app(lam(A,app(A,mia)),lam(B,snort(B)))  
Reduced = snort(mia)
```

```
yes
```

This, of course, is exactly what we wanted.

Implementing α -conversion

The stack-based approach to β -conversion just discussed is the heart of the black box we are constructing. Nonetheless, as it stands, our

implementation of β -conversion is *not* fully correct. Why not? Because we have ignored the need to perform α -conversion.

As we mentioned earlier, when performing β -conversion we have to take care that none of the free variables in the argument becomes accidentally bound by lambdas or quantifiers in the functor. There is a very easy way to prevent such problems: before carrying out β -conversion we should change all the bound variables in the functor (both those bound by lambdas, and those bound by quantifiers) to variables not used in the argument. Doing so guarantees that accidental bindings simply cannot occur—and *not* doing so can have disastrous consequences for semantic construction.

Here's a typical example of what can go wrong. Let's start with the following representations for the NP **every man** and the VP **loves a woman**:

$$\begin{aligned} \text{every man (NP): } & \lambda u. \forall y (\text{MAN}(y) \rightarrow u @ y), \\ \text{loves a woman (VP): } & \lambda x. \exists y (\text{WOMAN}(y) \wedge \text{LOVE}(x, y)). \end{aligned}$$

Applying the NP representation to the VP representation gives us the following representation for the sentence:

$$\lambda u. \forall y (\text{MAN}(y) \rightarrow u @ y) @ \lambda x. \exists y (\text{WOMAN}(y) \wedge \text{LOVE}(x, y)).$$

Let's reduce this to something more readable. Substituting the argument into u yields:

$$\forall y (\text{MAN}(y) \rightarrow \lambda x. \exists y (\text{WOMAN}(y) \wedge \text{LOVE}(x, y)) @ y).$$

So far so good—but now things can go badly wrong if we are careless. We need to perform β -conversion on the following subexpression:

$$\lambda x. \exists y (\text{WOMAN}(y) \wedge \text{LOVE}(x, y)) @ y.$$

At this point, warning bells should start to clang. In this subexpression, the argument is the free variable y . But now look at the functor: y occurs existentially quantified, so if we substitute x for its argument y we end up with:

$$\forall y (\text{MAN}(y) \rightarrow \exists y (\text{WOMAN}(y) \wedge \text{LOVE}(y, y))).$$

This certainly does *not* express the intended meaning of the sentence **every man loves a woman** (it says something along the lines of “everything is not a man or some woman loves herself”). The existential quantifier has ‘stolen’ the y , which really should have been bound by the outermost universal quantifier.

But we don't have to fall into this trap. Indeed, avoiding it is simplicity itself: all we have to do is α -convert the functor of the subexpression so that it does not bind the variable y . For example, if we decided to replace y by z in the functor, instead of reducing the expression

$$\lambda x. \exists y (\text{WOMAN}(y) \wedge \text{LOVE}(x,y)) @ y$$

(which is what leads to trouble) then our task is instead to reduce the following α -equivalent expression

$$\lambda x. \exists z (\text{WOMAN}(z) \wedge \text{LOVE}(x,z)) @ y.$$

Doing it this way yields

$$\forall y (\text{MAN}(y) \rightarrow \exists z (\text{WOMAN}(z) \wedge \text{LOVE}(y,z))),$$

which is a sensible semantic representation for every man loves a woman.

The consequences for our implementation are clear. Two tasks remain before our desired black box is ready: we must implement a predicate which carries out α -conversion, and we must alter the β -conversion predicate so that it uses the α -conversion predicate before carrying out reductions.

So the first question is: how do we implement α -conversion? In essence we recursively strip down the expression that needs to be α -converted and relabel bound variables as we encounter them. Along the way we will need to do some administration to keep track of free and bound variables (remember we only want to rename bound variables). We shall do this with the help of a list of substitutions and a (difference-) list of free variables. These lists will be empty at the beginning of the conversion process (they will be initialised by the main predicate for α -conversion, `alphaConvert/2`):

```
alphaConvert(Expression, Converted) :-  
    alphaConvert(Expression, [], [], Converted).
```

And now we simply carry on and define clauses covering all possible kinds of expressions that could be encountered. First, what happens if we encounter a variable? Well, if the variable is part of the list of substitutions, we simply substitute it. Otherwise, it must be a free variable, and we leave it as it is and add it to the list of free variables. This is coded as follows:

```
alphaConvert(X, Sub, Free1-Free2, Y) :-  
    var(X),  
    (  
        memberList(sub(Z,Y), Sub),  
        X==Z, !,  
        Free2=Free1  
    ;  
        Y=X,  
        Free2=[X|Free1]  
    ).
```

Let's now deal with the crucial case of the variable binders (the ex-

istential quantifier, the universal quantifier, and the lambda-operator). As we would expect, these are the expressions that introduce substitutions: the new variable Y will be substituted for X.

```

alphaConvert(Expression,Subs,Free1-Free2,some(Y,F2)):-  

    nonvar(Expression),  

    Expression = some(X,F1),  

    alphaConvert(F1,[sub(X,Y)|Subs],Free1-Free2,F2).  
  

alphaConvert(Expression,Subs,Free1-Free2,all(Y,F2)):-  

    nonvar(Expression),  

    Expression = all(X,F1),  

    alphaConvert(F1,[sub(X,Y)|Subs],Free1-Free2,F2).  
  

alphaConvert(Expression,Subs,Free1-Free2,lambda(Y,F2)):-  

    nonvar(Expression),  

    Expression = lambda(X,F1),  

    alphaConvert(F1,[sub(X,Y)|Subs],Free1-Free2,F2).

```

Other complex expressions (such as conjunctions) are broken down into their parts (using `compose/3`), and α -converted as well, using a predicated called `alphaConvertList/4`. This part of the code is very much like that used in the analogous part of our β -conversion predicate:

```

alphaConvert(F1,Subs,Free1-Free2,F2):-  

    nonvar(F1),  

    \+ F1 = some(_,_),  

    \+ F1 = all(_,_),  

    \+ F1 = lambda(_,_),  

    compose(F1,Symbol,Args1),  

    alphaConvertList(Args1,Subs,Free1-Free2,Args2),  

    compose(F2,Symbol,Args2).

```

And `alphaConvertList/4` has the expected recursive definition:

```
alphaConvertList([],_,Free-Free,[]).
```

```

alphaConvertList([X|L1],Subs,Free1-Free3,[Y|L2]):-  

    alphaConvert(X,Subs,Free1-Free2,Y),  

    alphaConvertList(L1,Subs,Free2-Free3,L2).

```

And that's all there is to it. As this predicate creates brand new symbols every time it encounters a binding, its output is exactly what we want. For example, in the following query the `some/2` and the `all/2` bind the same variable X. In the output they each bind a distinct, brand new, variable:

```
?- alphaConvert(some(X, and(man(X), all(X,woman(X)))),R).
```

```
R = some(_A, and(man(_A), all(_B, woman(_B))))  
yes
```

The Black Box

Fine—but now that we are able to perform α -conversion, how do we use it to make our β -conversion predicate correct? In the obvious way. Here is the clause of our β -conversion predicate that handles the case for applications:

```
betaConvert(Expression,Result,Stack):-  
    nonvar(Expression),  
    Expression = app(Functor,Argument),  
    nonvar(Functor),  
    betaConvert(Functor,Result,[Argument|Stack]).
```

We simply have to add a line which uses `alphaConvert/2` to relabel all bound variables in the functor using fresh new symbols:

```
betaConvert(Expression,Result,Stack):-  
    nonvar(Expression),  
    Expression = app(Functor,Argument),  
    nonvar(Functor),  
    alphaConvert(Functor,Converted),  
    betaConvert(Converted,Result,[Argument|Stack]).
```

And that's our black box completed.

Of course, we should test that it really works. There is a test suite containing entries of the following form:

```
expression(app(lam(A, lam(B, like(B,A))), mia),  
         lam(C, like(C, mia))).
```

The first argument of `expression/2` is the lambda expression to be β -converted, the second is the result (or more accurately, one of the possible results). If you load the file `betaConversion.pl` and then issue the command

```
?- betaConvertTestSuite.
```

Prolog will evaluate all the examples in the test suite, and the output will be a series of entries of the following form:

```
Expression: app(lam(_716,app(_716,mia)),lam(_722,walk(_722)))  
Expected: walk(mia)  
Converted: walk(mia)  
Result: ok
```

This tells us that our `betaConvert/2` predicate has been applied to `Expression` (that is, the first argument of `expression/2`) to produce

Converted, and because this matches what we Expected (that is, the second argument of `expression/2`) the Result is ok.

Fine—but why did we say that the second argument of `expression/2` was “one of the possible results”? Well, consider the following example:

```
Expression: app(lam(_716, lam(_719, like(_719, _716))), mia)
Expected: lam(_725, like(_725, mia))
Converted: lam(_939, like(_939, mia))
Result: ok
```

Here Expected and Converted are not identical—but the result is clearly right as the two expressions are α -equivalent. So when we say that Expected and Converted should “match”, what we really mean is that they have to be α -equivalent.

How do we test that two expressions are α -equivalent? It’s a two-step process. We first check that any free variables used in the expressions are identical, and occur at the same position. We then check that (if we ignore the differences in the variables that are bound) the two expressions have the same syntactic structure. The following code does this:

```
alphabeticVariants(Term1, Term2):-
    alphaConvert(Term1, [], []-Free1, Term3),
    alphaConvert(Term2, [], []-Free2, Term4),
    Free1==Free2,
    numbervars(Term3, 0, _),
    numbervars(Term4, 0, _),
    Term3=Term4.
```

The first three lines of this code check that the free variables in `Term1` and `Term2` (the two terms we want to prove are α -equivalent) are identical and occur in same position; this is done with the help of our `alphaConvert/4` predicate. Note that this part of the process gives us two new terms (`Term3` and `Term4`) and the next step is to check whether these new terms have the same syntactic structure. We do this by making use of the built-in predicate `numbervars/3`. This instantiates all Prolog variables with unique non-variables, and once that’s done, we use unification to check whether the instantiated terms `Term3` and `Term4` are identical.

One thing may be unclear. Why do we use `numbervars/3` to substitute non-variables for variables? Why not simply try unifying the terms after we have checked that the free variables are the same? Because this won’t work. Consider the formulas $\exists x \forall y R(x,y)$ and $\exists x \forall y R(y,x)$. These are not α -equivalents, but Prolog would cheerfully unify the variable `x`

and y and declare that they were! By first substituting non-variables for variables we prevent such problems.

This concludes our discussion of the λ -calculus. As should now be clear, it is a tool that every computational semanticist should be familiar with. It gives us a transparent and concise notation for specifying semantic representations, and comes with simple and well-defined mechanisms for gluing them together. Moreover, as we have seen, the key mechanisms can be wrapped up into a black box. To be sure, we have spent the last few pages looking under the hood at how it all works, but having done that we can now slam the hood firmly shut again. We don't need to worry anymore about whether variables unify, or how we can guarantee that there won't be clashes that lead to errors in the semantic construction process. It's plug-and-play time. We're free to start thinking about computational semantics.

Exercise 2.4.1 The β -conversion code is in file `betaConversion.pl`. It contains a commented-out sequence of print instructions that displays the contents of the stack. Comment in these print instructions and then try out the examples in file `betaConversionTestSuite.pl` (which was designed by David Milward). Incidentally, don't just run the test suite—read it as well. Some of the examples are very instructive, and the file contains comments on many of them.

Exercise 2.4.2 In the text we learned that failing to apply α -conversion before reducing β -expressions can yield the incorrect representations. In fact, if we try to work with our first version of `betaConvert/2` (the one that doesn't call on `alphaConvert/2`) we also encounter a host of additional Prolog-specific problems.

The β -conversion code is in file `betaConversion.pl`. By commenting out a single marked line in this file, α -conversion can be turned off. Do the following experiments.

1. Enrich the DCG with a rule for `and`. Use $\lambda u.\lambda v.\lambda x.(u@x \wedge v@x)$ as its semantic representation. Show that with α -conversion switched on, your DCG gives the correct representation for Vincent and Mia dance. Then explain why Prolog fails to build any representation at all when α -conversion is turned off.
2. Run `betaConversionTestSuite/0` with α -conversion turned off. You will encounter some surprising behaviour (for example, in one case Prolog fails to terminate). Explain these failures.

Exercise 2.4.3 Give the Prolog code for lexical entries of ditransitive verbs such as `offer` in `Vincent offers Mia a drink`.

Exercise 2.4.4 Find a suitable lambda expression for the lexical entry of the determiner `no`, and then give the corresponding Prolog code.

Exercise 2.4.5 Extend the DCG so that it covers sentences such as everyone dances and somebody snorts.

Exercise 2.4.6 Pereira and Shieber (see Notes at the end of the chapter) provide a simpler approach to the semantics for transitive verbs. Firstly, `lam(X, lam(Y, love(X, Y)))` is their semantic representation for the verb `love`. Secondly, they use the following grammar rule to deal with transitive verbs:

```
vp(lam(X,app(NP,TV)))--> tv(lam(X,TV)), np(NP).
```

Explain how this works. Is it really unproblematic?

Exercise 2.4.7 In the text, when we used the black box with our DCG we carried out β -conversion only after the representation had been built for the entire sentence. Change the DCG (which you can find in `experiment3.pl`) so that β -conversion is interleaved with semantic construction. This can be done by making β -conversion part and parcel of the grammar rules. For instance, change the rule for sentences to

```
s(S)--> np(NP), vp(VP), {betaConvert(app(NP,VP),S)}.
```

Exercise 2.4.8 Food for thought. What happens when you functionally apply $\lambda x.(x@x)$ to itself and β -convert? And what happens when you functionally apply $\lambda x.((x@x)x)$ to itself and β -convert?

Programs for implementing the lambda calculus

`experiment3.pl`

DCG with lambda calculus for a small fragment of English.

`betaConversion.pl`

Implementation of β -conversion.

`alphaConversion.pl`

Implementation of α -conversion.

`betaConversionTestSuite.pl`

Examples to test our implementation of β -conversion.

`comsemPredicates.pl`

Definitions of some auxiliary predicates.

2.5 Grammar Engineering

With our Prolog notation for lambda abstraction and functional application, and our black box for β -conversion, we have the basic semantic construction tools we need. So it is time to move on from the baby DCGs we have been playing with and define a more interesting grammar fragment.

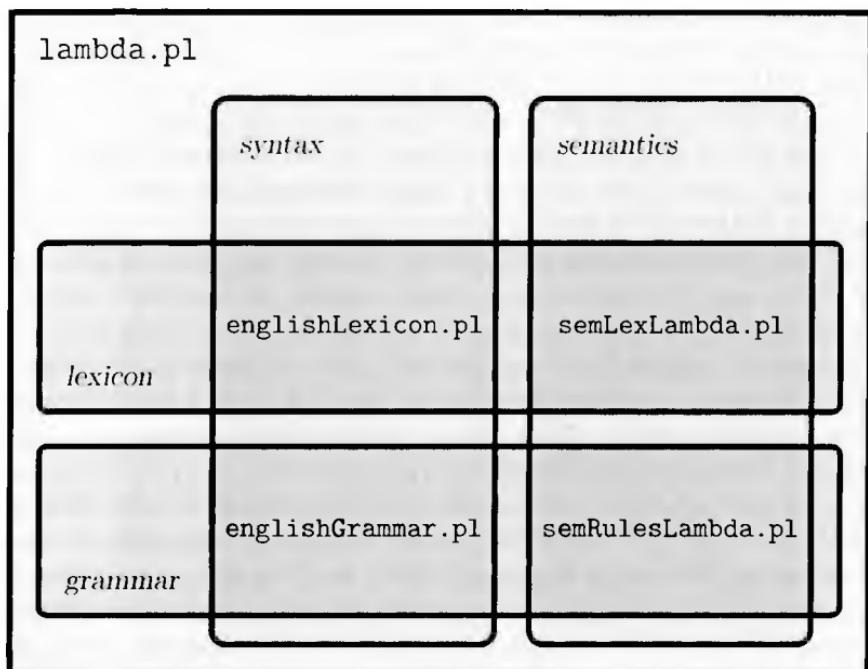
But let's do it right—we should try to observe some basic principles of grammar engineering. That is, we should strive for a grammar that is *modular* (each component should have a clear role to play and a clean interface with the other components), *extendible* (it should be straightforward to enrich the grammar should the need arise) and *reusable* (we should be able to reuse a significant portion of the grammar, even when we change the underlying semantic representation language).

Grammar engineering principles have strongly influenced the design of the larger grammar we shall now discuss—and this is *not* purely for pedagogical reasons. In the course of writing this book we have experimented with many ways of computing semantic representations (for example, in the following chapter we will consider three different techniques for coping with scope ambiguities). Moreover, we have checked that our approach is compatible with other semantic representation languages (notably the Discourse Representation Structures, used in DRT, though this is not a topic we shall explore in this book). As we have learned (often the hard way) incorporating such extensions, and keeping track of what is going on, requires a disciplined approach towards grammar design. In the end we opted for a fairly simple two-dimensional grammar architecture consisting of a collection of *syntax rules* and corresponding *semantic rules*, and a *lexicon* with corresponding *lexical semantics*.

The syntax rules are DCG rules annotated with additional grammatical information. These rules will *not* change in the course of the book. The lexicon lists information about words belonging to various syntactic categories in an easily extractable form; this component will also stay fixed throughout the book. The syntax rules together with the lexical entries constitute the *syntactic* part of the grammar.

The *semantic* part of the grammar consists of the semantic rules and the lambda definitions that define the lexical semantics. The semantic rules state how the semantic representation of a complex category is computed out of the semantic representations of its parts. The semantic lexicon is where we state what we have previously called “lexical entries”. It is here that we will do most of our semantic work, and our modifications will largely be confined to this level.

The diagram below illustrates the modular design of the syntax-semantics interface.



As the diagram shows, the information guiding the semantic construction process is spread across five files. The top level file `lambda.pl`, contains the implementation of the lambda calculus we have discussed, and makes use of the linguistic information provided by the four lower level files. We'll first discuss how these lower level files represent the various kinds of linguistic information, and then we'll see how the top level file wraps everything together.

The Syntax Rules

Let's first examine the core DCG rules that we shall use (we won't show all the rules, as many of them are fairly similar; consult the file `englishGrammar.pl` for the complete set). The rules license a number of semantically important constructions, such as proper names, determiners, pronouns, relative clauses, the copula construction, and coordination. These DCG rules resemble those we used earlier in this chapter, but there are some important differences. First of all, in order to deal with agreement, morphology, and coordination, we will decorate syntactic categories with a list of attribute-value pairs expressing this grammatical information. And secondly, all syntax rules come with a

hook to a semantic rule, in the form of the `combine/2` notation. Here is an example illustrating both these extensions:

```
s([coord:yes,sem:Sem])-->
  np([coord:_,num:Num,sem:NP]),
  vp([coord:_,inf:fin,num:Num,sem:VP]),
  {combine(s:Sem,[np:NP,vp:VP])}.
```

This rule is of course the familiar $S \rightarrow NP\ VP$ rule, but it tells us slightly more: that this S is a non-coordinating sentence, that the number feature of NP and VP are constrained to be of the same value, and that the inflectional form of the VP must be finite. It also tells us (here is where `combine/2` is used) that the S semantics is formed by combining the NP and the VP semantics. We will see how this combination is actually to be carried out when we examine the contents of the file `semRulesLambda.pl`; in the meantime, we'll simply remark that factoring out the definition of the `combine/2` predicate in this manner helps keep things modular.

Let's now consider some of the coordination rules. Implementing coordination in DCGs requires special attention because left-recursive syntax rules (that is, rules of the form $X \rightarrow X\ Y$) will cause Prolog to fall into an infinite loop. The way we solve this is simple and effective: we introduce a feature `coord` that can either have the value `yes` or `no`.

```
np([coord:yes,num:p1,sem:NP])-->
  np([coord:no,num:sg,sem:NP1]),
  coord([type:conj,sem:C]),
  np([coord:_,num:_,sem:NP2]),
  {combine(np:NP,[np:NP1,coord:C,np:NP2])}.
```

```
np([coord:yes,num:sg,sem:NP])-->
  np([coord:no,num:sg,sem:NP1]),
  coord([type:disj,sem:C]),
  np([coord:_,num:sg,sem:NP2]),
  {combine(np:NP,[np:NP1,coord:C,np:NP2])}.
```

Note that these rules show a further interesting point about English noun phrases in coordination, namely that their grammatical number is determined by the kind of coordinating particle that is used. So, the NP `Mia` and `Vincent` will receive the value `p1` for the feature `num`, whereas `Mia` or `Vincent` gets the value `sg`.

Let's look at some lexical rules. These are rules that apply to terminal symbols, the actual strings that the DCG is trying to analyse:

```
noun([sem:Sem])-->
  {lexEntry(noun,[symbol:Symbol,syntax:Word])},
  Word,
```

```
{semLex(noun, [symbol:Symbol, sem:Sem])}.
```

First we call the lexicon to check if a string belongs to the syntactic category being searched for (this gives us `Word`). What does the `semLex/2` predicate do? Think of it as a *semantic macro* which is used to construct the actual semantic representation for the noun. In fact, each lexical category is associated with such a macro; there are semantic macros for intransitive verbs, transitive verbs, determiners, and so on.

Why do things this way? Because it helps us to keep things modular. If we want to change the semantic representation used for (say) transitive verbs, we simply change the appropriate semantic macro (and if there are a lot of transitive verbs in our lexicon, this saves a lot of work). Indeed, because of our use of semantic macros, it is possible to make far more substantial changes to the underlying semantic representation formalism without much difficulty. For example, in the following chapter we will see that a richer notion of semantic representation is needed if we are to deal with the problems raised by scope ambiguities. We'll be able to define the richer representations we shall require, and make them work with our semantic construction machinery simply by changing the semantic macros (and the way `combine/2` is defined).

What are the shortcomings of our grammar? One in particular should be mentioned. We have implemented only a limited amount of inflectional morphology—all our examples are relentlessly third-person present-tense. This is a shame (tense and its interaction with temporal reference is a particularly rich source of semantic examples). Nonetheless, we shall not be short of interesting things to do. Even this small set of rules assigns tree structures to an interesting range of English sentences:

- Mia knows every owner of a hash bar.
- Vincent or Mia dances.
- Every boxer that kills a criminal loves a woman.
- Vincent does not love a boxer or criminal that snorts.
- Vincent does not love a boxer or a criminal that snorts.
- If a boxer snorts then a woman collapses.

The Semantic Rules

Let's turn to the semantic rules, that is, the contents of the file `semRulesLambda.pl`. This is where the `combine/2` predicate is defined. Here the news is extremely pleasant. For a start, the required semantic annotations are utterly straightforward; they are simply the obvious “apply the function to the argument statements” expressed with the help of `app/2`.

```
combine(s:app(A,B),[np:A, vp:B]).
```

That is, we simply apply the NP semantics to the VP semantics. None of the rules are much more complex than this. The rules for coordination are the most complex, but even these are straightforward as the following example shows:

```
combine(np:app(app(B,A),C),[np:A,coord:B,np:C]).
```

The unary rules, of course, are even simpler for they merely pass the representation up to the mother node. For example:

```
combine(np:A,[pn:A]).
```

Moreover, because the definition of `combine/2` pins down the fundamentals of the semantic construction process, it is easy to experiment with other forms of semantic construction. To give a simple example, we can experiment with carrying out β -conversion directly after we have combined two representations simply by redefining `combine/2`. Here, for example, is the new clause that would be used for building sentence representations:

```
combine(s:Converted,[np:A, vp:B]):-  
    betaConvert(app(A,B), Converted).
```

In short, the architecture is modular. In the following chapter we will exploit this modularity in a more far-reaching fashion to deal with quantifier scope.

The Lexicon

Let's now turn to the lexicon (that is, `englishLexicon.pl`). The general format of a lexical entry is `lexEntry(Cat, Features)` where `Cat` is the syntactic category, and `Features` is a list of attribute-value pairs. This list contains further information about the lexical entry, such as the symbols that will be used in the semantic representation, the surface form of the lexical entry, and other grammatical information specific to the type of lexical entry.

We will go through a few examples to give an idea of how the lexicon is set up. Let's first consider nouns, proper names, and adjectives:

```
lexEntry(noun,[symbol:burger,syntax:[burger]]).
```

```
lexEntry(noun,[symbol:boxer,syntax:[boxer]]).
```

```
lexEntry(noun,[symbol:car,syntax:[car]]).
```

```
lexEntry(pn,[symbol:mia,syntax:[mia]]).
```

```
lexEntry(pn,[symbol:yolanda,syntax:[yolanda]]).
```

```
lexEntry(pn,[symbol:vincent,syntax:[vincent,vega]]).
```

```
lexEntry(adj, [symbol:big,syntax:[big]]).
lexEntry(adj, [symbol:kahuna,syntax:[kahuna]]).
lexEntry(adj, [symbol:married,syntax:[married]]).
```

As these examples show, nouns, proper names, and adjectives in our lexicon come with two features: `symbol`, the relation symbol used to compute the semantic representation), and `syntax`, a list of words describing the appearance of the lexical entry in a phrase. Note that we use a list for the `syntax` feature in order to deal with multi-word lexemes—for example, for the compound name `Vincent Vega`.

Other types of lexical entries make use of different features. For example, the entries for determiners have information about the syntactic mood and the type of quantification that they introduce:

```
lexEntry(det, [syntax:[every],mood:decl,type:uni]).
lexEntry(det, [syntax:[a],mood:decl,type:indef]).
lexEntry(det, [syntax:[the],mood:decl,type:def]).
lexEntry(det, [syntax:[which],mood:int,type:wh]).
```

Moreover, note that there is no feature `symbol` in the entries for determiners. Why is that? Well, the semantic contribution of determiners is not simply a constant or relation symbol, but rather a relatively complex expression that is dependent on the underlying representation language. Hence we shall specify the semantics of these categories in the semantic lexicon.

Let's now have a look at the verbs. First, the transitive verbs. These entries contain information about morphological inflection (the feature `inf` with possible values `inf` for infinite forms and `fin` for finite forms) and number (the feature `num` with possible values `sg` for singular and `pl` for plural). Consider for instance the verb `to clean` in the lexicon:

```
lexEntry(tv, [symbol:clean,syntax:[clean],inf:inf,num:sg]).
lexEntry(tv, [symbol:clean,syntax:[cleans],inf:fin,num:sg]).
lexEntry(tv, [symbol:clean,syntax:[clean],inf:fin,num:pl]).
```

As a final example consider the lexical entries for the auxiliary verbs:

```
lexEntry(av, [syntax:[does],inf:fin,num:sg,pol:pos]).
lexEntry(av, [syntax:[does,not],inf:fin,num:sg,pol:neg]).
lexEntry(av, [syntax:[did],inf:fin,num:sg,pol:pos]).
lexEntry(av, [syntax:[did,not],inf:fin,num:sg,pol:neg]).
```

The auxiliary verbs come with the additional feature `pol`, expressing the polarity, whose value is either `pos` or `neg`. This information is used for constructing the appropriate semantic representations for expressions with auxiliary verbs.

This way of setting up a lexicon offers natural expansion options. For example, if we decided to develop a grammar that dealt with both

singular and plural forms of nouns (which we won't do in this book, though see Exercise 2.5.5), it is just a matter of extending the general format of entries with one or more fields.

The Semantic Lexicon

We now come to the most important part of the grammar, the semantic lexicon. As the introduction of `combine/2` has reduced the process of combining semantic representations to an elegant triviality, and as the only semantic information the lexicon supplies is the relevant constant and relation symbols, the semantic lexicon is where the real semantic work will be done. And, as we have already discussed, this work boils down to devising suitable semantic macros. Let's consider some examples right away.

```
semLex(pn,M) :-
    M = [symbol:Sym,
         sem:lam(U,app(U,Sym))].

semLex(noun,M) :-
    M = [symbol:Sym,
         sem:lam(X,Formula)],
    compose(Formula,Sym,[X]).

semLex(tv,M) :-
    M = [symbol:Sym,
         sem:lam(K, lam(Y, app(K, lam(X,Formula))))],
    compose(Formula,Sym,[Y,X]).
```

The semantic macro for proper names is straightforward. For instance, for `Mia`, the value `mia` will be associated with the feature `symbol`, and hence the representation `lam(U,app(U,mia))` will be associated with the feature `sem`. The second macro builds a semantic representation for any noun given the predicate symbol `Sym`, turning this into a formula lambda-abstracted with respect to a single variable. The representation is built using the `compose/3` predicate to coerce it into the required lambda expression. For example, the symbol `boxer` would give the representation `lam(X,boxer(X))`.

As we've already mentioned, the semantic lexicon also has self-contained entries for the determiners. Here they are:

```
semLex(det,M) :-
    M = [type:uni,
         sem:lam(U, lam(V, all(X, imp(app(U,X), app(V,X)))))].
```



```
semLex(det,M) :-
```

```
M = [type:indef,
      sem:lam(P, lam(Q, some(X, and(app(P,X), app(Q,X)))))].
```

These, of course, are just the old-style lexical entries we are used to. For a complete listing of the semantic lexicon we have just been discussing, see the file `semLexLambda.pl`. We shall see other types of lexical semantics as we work our way through the book.

Wrapping Everything Together

Finally we turn to the main level program.

```
lambda:-  
    readLine(Sentence),  
    lambda(Sentence, Sems),  
    setof(Sem, t([sem:Sem], Sentence, []), Sems),  
    printRepresentations(Sems).
```

This uses `readLine/1` to read in a sentence, computes all semantic representations, and finally prints out all semantic representations in a nicely ordered way. The `readLine/1` predicate allows us to type in sentences in a natural way (rather than the lists used with DCGs in the earlier experiments). For example:

```
?- lambda.  
  
> Mia knows a boxer.  
  
1 some(A, and(boxer(A), know(mia, A)))
```

The sentence test suite is in file `sentenceTestSuite.pl`, which contains entries of the form:

```
sentence([a,man,walks], 1).
```

The test suite can be started with `lambdaTestSuite/0`.

And that's the architecture. From now on, the syntax rules and the lexicon will be used unchanged. To put it another way: *from now on, the locus of change will be the semantic lexicon and the semantic rules*. In particular, it is here that we will develop our treatments of quantifier scope in the following chapter.

Exercise 2.5.1 Find out how copula verbs are handled in the lexicon and grammar, and how the semantic representations for sentences like *Mia is a boxer* and *Mia is not Vincent* are generated.

Exercise 2.5.2 Extend the grammar so that it handles expressions of the form *Vincent is male*, *Mia and Vincent are cool*, and *Marsellus or Butch is big*.

Exercise 2.5.3 Extend the grammar so that it handles expressions of the form **Vincent and Jules are in a restaurant** and **Butch is on a motorbike**.

Exercise 2.5.4 Add the preposition **without** to the lexicon, and define a new semantic macro that takes the implicit negation of this preposition into account. For instance, **a man without a weapon** should receive a representation such as $\exists x(\text{MAN}(x) \wedge \neg \exists y(\text{WEAPON}(y) \wedge \text{WITH}(x,y)))$.

Exercise 2.5.5 Extend the grammar (the syntax rules and the lexicon) to cover plural forms of nouns. Introduce a new feature in the lexical entries to express number information. Then add entries for determiners and classify them as combining with singular nouns only (for instance the determiner **a**), combining with plural nouns only (for instance **both**, **or** **all**), or combining with either (for example **the**).

Notes

Compositionality is a simple and natural idea—and one capable of arousing an enormous amount of controversy. Traditionally attributed to Gottlob Frege (the formulation “the meaning of the whole is a function of the meaning of its parts” is often called Frege’s principle) it received a precise mathematical formulation in Richard Montague’s paper “Universal Grammar” (Montague, 1970c). For a detailed and accessible overview of the compositionality concept, see Janssen (1997). For evidence that there are still interesting issues to be resolved about this venerable concept, see Pagin and Westerståhl (2001), a special issue of the *Journal of Logic, Language and Information* devoted to compositionality.

Compositional approaches to semantic analysis use syntactic structure to guide the semantic construction process. In this book we have opted for a relatively simple notion of syntactic structure (finite trees whose nodes are labelled with categories) and an easy-to-use (but not particularly sophisticated) mechanism for grammar specification and parsing (namely, DCGs), but we did not discuss why these were (or were not) good choices, nor consider alternatives. To find out more about what syntactic structure is, and how to determine what sort of syntactic structure a sentence has, consult either Radford (1997) or Burton-Roberts (1986); the first is a thorough introduction to syntax from the perspective of contemporary Chomskyan theorising, whereas the second is a hands-on introduction to determining the kinds of trees (and other structures) that constitute the syntactic structure of English. Both books are clearly written and should be accessible even if you have little or no linguistic background. For more on processing syntax,

Programs for the full grammar fragment

`lambda.pl`

This is the main file for the implementation of the lambda calculus using the extended grammar

`readLine.pl`

Module for converting strings into lists.

`sentenceTestSuite.pl`

Test suite with sentences.

`semLexLambda.pl`

Semantic lexicon for `lambda.pl`.

`semRulesLambda.pl`

Semantic rules for `lambda.pl`.

`englishLexicon.pl`

Our standard English lexical entries. Contains entries for nouns, proper names, intransitive and transitive verbs, prepositions, and pronouns.

`englishGrammar.pl`

Our standard grammatical rules for a fragment of English. Rules cover basic sentences, noun phrases, relative clauses and modification of prepositional phrases, verb phrases, and a limited form of coordination.

Gazdar and Mellish (1989) is a good choice. It is Prolog based, and discusses DCGs, more sophisticated alternatives to DCGs, attribute-value structures (an important generalization of the attribute-value pairs we made use of in our grammar architecture) and also contains chapters on semantic and pragmatic processing. Finally, for a clear and up-to-date introduction to all aspects of speech and language processing, see Jurafsky and Martin (2000). As we remarked in the Introduction, this is the standard textbook on speech and language processing, and the approach it takes to semantic construction is based on first-order logic and lambda calculus (see in particular Chapters 14 and 15). Chapter 15 also contains brief discussions of how semantic construction can be integrated with Earley Parsers (an efficient approach to natural language parsing), and of how compositional approaches to semantic analysis can be adapted for use in practical systems.

The idea of using the lambda calculus to specify the meanings of lexical entries, and using functional application and β -conversion as the basic mechanism for combining representations, is due to Richard Montague; it is used in both “Universal Grammar” (Montague, 1970c) and “The Proper Treatment of Quantification in Ordinary English” (Montague, 1973) (as we said in the Introduction, what we presented in this chapter is essentially the computational heart of Montague semantics). Montague’s original papers are well worth reading (they are collected in Montague (1974)). Nonetheless, they are densely written and many readers will be better off approaching Montague semantics via either Dowty et al. (1981) or Gamut (1991b); both contain good, careful, textbook level expositions of Montague’s key ideas. For more detailed discussion, try Janssen (1986a) and Janssen (1986b), or the more up-to-date and wide ranging Carpenter (1997). For an overview of Montague’s work, and an account of the major directions in which his work has been developed, see Partee (1997a).

As we shall shortly discuss, much of the literature on λ -calculus is logical or mathematical in nature. In this book we have tried to emphasise that there is a down-to-earth computational perspective on λ -calculus too. Not only is the lambda calculus a useful tool for gluing representations together, but the basic ideas emerge, with seeming inevitability, when one sits down and actually tries to do semantic construction in Prolog—or at least, that is what we have tried to suggest by approaching lambda calculus via experiments 1 and 2. We hasten to add that this “seeming inevitability” is clear only with the benefit of hindsight. The links between the ideas of logic programming and Montague semantics seem to have first been explicitly drawn in Pereira and Shieber (1987).

However our account hasn't discussed one interesting issue: what do lambda expressions actually *mean*? Hopefully the reader now has a pretty firm grasp of what one can *do* with lambda expressions—but are we forced to think of lambda expressions purely procedurally? As we are associating lambda expressions with expressions of natural language, it would be nice if we could give them some kind of model-theoretic interpretation.

Actually, there's something even more basic we haven't done: we haven't been particularly precise about what counts as a λ -expression! Moreover—as readers who did Exercise 2.4.8 will be aware—if one takes an “anything goes” attitude, it is possible to form some pretty wild (and indeed, wildly pretty) expressions, such as $\lambda x.(x@x)$ which when applied to itself and β -converted yields itself, and $\lambda x.((x@x)@x)$ which when applied to itself and β -converted yields ever longer lambda expressions.

Let's briefly discuss such issues. The main point we wish to make is that there are two major variants of the lambda calculus, namely the *untyped* lambda calculus, and the *typed* lambda calculus. Both can be given model-theoretic interpretations in terms of functions and arguments, and both can be regarded as programming languages—but the version of typed lambda calculus most widely used in natural language semantics also has a simple logical interpretation. Both the untyped and typed lambda calculus were developed in the 1930s by Alonzo Church and his then students Barkley Rosser and Stephen Kleene; Church (1941) is the first expository account.

The untyped lambda calculus adopts an “anything goes” attitude to functional application and β -conversion. For example, $\lambda x.((x@x)@x)$ is a perfectly reasonable expression in untyped lambda calculus, and it is fine to apply it to itself and β -convert. Doing so leads to a non-terminating computation—but this merely reflects the fact that the untyped lambda calculus is a full-blown programming language (and as every programmer knows, non-terminating programs are a fact of life). Indeed, when you get down to it, the untyped lambda calculus is what lies at the heart of the functional programming language Lisp. Writing a (pure) Lisp program is all about defining functions, and the core mechanism Lisp offers for this purpose is lambda abstraction (for a classic introduction to functional programming from a Lisp-as-lambda-calculus perspective, see Abelson and Sussman (1985)).

Much to everyone's surprise, the untyped lambda calculus turned out to have a model-theoretic semantics, and a very beautiful one at that. Clearly, any reasonable semantics should treat abstractions as functions—the difficulty is to find suitable collections of functions in

which the idea of self-application can be captured (after all, we want to be able to model the self-applicative behaviour of expressions like $\lambda x.((x@x)@x)$ correctly). In standard set theory, functions *cannot* be applied to themselves, so constructing function spaces with the structure necessary to model self-application is no easy task. However Dana Scott showed the way forwards with his D_∞ model (see Scott (1970) and Scott (1973)) and since then a number of other model construction techniques have been developed.

The idea of giving model-theoretic interpretations to the untyped lambda calculus (and other programming languages) gave rise to a new branch of theoretical computer science called *denotational semantics of programming languages*. This emerged around the same time that Montague was doing his pioneering work on natural language semantics (that is, in the late 1960s and early 1970s) and the two fields, though different, have a lot in common. The classic introduction to denotational semantics is Stoy (1977). Though dated, this is well worth looking at, and has an interesting Foreword by Dana Scott. For a more recent (and more wide ranging) text, try Nielson and Nielson (1992).

But the style of lambda calculus that has had the most impact on natural language semantics is typed lambda calculus. Actually, typed lambda calculi come in many flavours; the one we shall discuss is called the *simply typed lambda calculus with explicit (or Church-style) typing*, for ever since the work Richard Montague, this has generally been the tool of choice for natural language semanticists. For an important early paper on this style of typed lambda calculus, see Church (1940).

The key feature of this style of typed lambda calculus is that it adopts a very restrictive approach to functional application. Instead of “Anything goes”, the motto is “If it don’t fit, don’t force it”, and typed systems have exacting notions about what fits. Let’s discuss the idea of (simple explicit) typing in a little more detail.

To build the lambda expressions we have used in this book in such a system of typed lambda calculus, we would proceed as follows. First we would specify the set of types. There would be infinitely many of these, namely (1) the type e of individuals, (2) the type t of truth values, and (3) for any types τ_1 and τ_2 , the function type $\langle\tau_1, \tau_2\rangle$. Our logical language would contain all the familiar first-order symbols, but in addition it would contain an infinite collection of variables of each type (the ordinary first-order variables, which range over individuals, would now be thought of as the set of type e variables), and (crucially) it would also contain the λ and $@$ operators.

In the typed lambda calculus, every expression receives a *unique* type. The key clauses that ensure this are the definitions of abstraction

and functional application. First, if \mathcal{E} is a lambda expression of type τ_2 , and v is a variable of type τ_1 then $\lambda v.\mathcal{E}$ is a lambda expression of type $\langle \tau_1, \tau_2 \rangle$. Therefore, it matters which type of variable we abstract over: abstracting with respect to different types of variables results in abstractions with different types. Second, if \mathcal{E} is a lambda expression of type $\langle \tau_1, \tau_2 \rangle$, and \mathcal{E}' is a lambda expression of type τ_1 then (and only then!) are we permitted to functionally apply \mathcal{E} to \mathcal{E}' . If we do this, the application has type τ_2 . *In short, we are only allowed to perform functional application when the types of \mathcal{E} and \mathcal{E}' fit together correctly.* The intuition is that the types tell us what the domain and range of each expression is, and if these don't match, application is *not* permitted. Note, moreover, that the type of the expression that results from the application is determined by the types of \mathcal{E} and \mathcal{E}' . In effect, we have taken the (wild and crazy) untyped lambda calculus and tamed it with a strict typing discipline.

This has a number of consequences. For a start, self-application is impossible. (This is obvious. After all, to use \mathcal{E} as a functor, it must have a function type, say $\langle \tau_1, \tau_2 \rangle$. But then its argument must have type τ_1 . So \mathcal{E} can't be used as one of its own arguments, as every expression has a unique type, and we know that \mathcal{E} has type $\langle \tau_1, \tau_2 \rangle$.) Moreover, it can be shown that unending sequences of β -conversions of the type we noted in the untyped lambda calculus simply aren't possible in the typed lambda calculus (that is, all computations in simply typed lambda calculus terminate, or to put it another way, typed lambda calculus is a weaker programming language than the untyped lambda calculus).

But for natural language semantics perhaps the most interesting consequence of (simple explicit) typing is that it becomes extremely straightforward to give a model-theoretic semantics to such systems. To do so, simply take any first-order model M . The denotation D_e of type e expressions are the elements of the model, the permitted denotations D_t of the type t expressions are TRUE and FALSE, and the permitted denotations $D_{\langle \tau_1, \tau_2 \rangle}$ of type $\langle \tau_1, \tau_2 \rangle$ expressions are functions whose domain is D_{τ_1} and whose range is D_{τ_2} . In short, expressions of the simply typed lambda calculus are interpreted using an inductively defined function hierarchy.

Which particular functions are actually used in this hierarchy? Consider the expression $\lambda x.\text{MAN}(x)$, where x is an ordinary first-order variable. Now, $\text{MAN}(x)$ is a formula, something that (given a variable assignment) can be TRUE or FALSE, so this has type t . As was already mentioned, first-order variables are viewed as type e variables, hence it follows that the abstraction $\lambda x.\text{MAN}(x)$ has type $\langle e, t \rangle$. That is, it must

be interpreted by a function from the set of individuals to the set of truth values. But which one? In fact, it would standardly be interpreted by the function which, when given an individual from the domain of quantification as argument, returns TRUE if that individual is a man, and FALSE otherwise. That is, it is interpreted using the function which exactly *characterises* the subset of the model consisting of all men, or to put it another way, it is interpreted by the *property of manhood*. And this, of course, is precisely what the standard first-order semantics uses to interpret `MAN`. In short, the functional interpretation of lambda expressions is set up so that, via the mechanism of such *characteristic functions*, it meshes perfectly with the ordinary first-order interpretation.

By building over this base in a fairly straightforward way, the interpretation can be extended to cover *all* lambda expressions. For example, the expression

$$\lambda u. \exists x (\text{WOMAN}(x) \wedge u @ x)$$

(that is, the kind of expression associated with NPs) would be interpreted as a function which takes as argument the type of function that u denotes (and u denotes type $\langle e, t \rangle$ functions, that is, properties such as `run`) and returns a type t value (that is, either TRUE or FALSE). For example, if we combine this expression with $\lambda y. \text{RUN}(y)$, we get $\exists x (\text{WOMAN}(x) \wedge \text{RUN}(x))$, which is either true or false in a model. Admittedly, thinking in terms of functions that take other functions as arguments and return functions as values can get rather involved, but the basic idea is straightforward, and for applications in natural language semantics, only a very small part of the function hierarchy tends to be used.

And something else interesting happens. Recall that we allowed ourselves variables over *all* types. Well, as we've already said, quantification over entities of type e corresponds to first-order quantification. But what about quantification over entities of type $\langle e, t \rangle$ for example? As we just discussed, these are (characteristic functions of) properties—so if we quantify across such entities (and we are allowed to) we are carrying out what is known as *monadic second-order* quantification. Similarly, if we quantify across entities of type $\langle e, \langle e, t \rangle \rangle$ we are quantifying across two-place relations (after all, entities of this type combine with two entities to give a truth value, and that is what a two-place relation like `love` does), so we are carrying out what is known as *dyadic second-order* quantification. And in fact, as we can quantify across entities of arbitrary types, we're not merely able to perform various kinds of second-order quantification, we're actually at the controls of a system

of *higher-order logic* (or ω -*order logic*, or *classical type theory*, as it is often called).

Now, arguably natural language semantics never requires types much above order three or so—nonetheless the ability to take a logical perspective on higher-order types really is useful. Most importantly of all, it means that the consistency and informativity checking tasks (and indeed the querying task) that we introduced in the previous chapter can be extended to all types. Why? Because now an expression such as $\lambda x.\text{MAN}(x)$ is not only a useful piece of glue, it can also be regarded as a well-formed expression of higher-order logic with a model-theoretic interpretation (and indeed, Richard Montague treated lambda expressions in exactly this way). Moreover, given background knowledge, it makes complete sense to say (for example) that $\lambda x.\text{PERSON}(x)$ is a logical consequence of $\lambda x.\text{MAN}(x)$. The significance of this is that (theoretically at least) it clears the way for systematic use of inference at the *subsentential* level. The example just given shows an entailment relation between the nouns **MAN** and **PERSON**, and in a similar way implications can hold between NPs, VPs, PPs and so on.

To sum up our discussion: throughout this chapter we have talked about the lambda calculus as a mechanism for marking missing information, and we have exploited the mechanisms of functional application and β -conversion as a way of moving missing information to where we want it. But there is nothing mysterious or ill-defined about this metaphor. It is possible to give precise mathematical models of missing information in terms of functions and arguments: an abstraction is interpreted as a function, and the missing information is simply the argument we will later supply it with. Indeed, a variety of models are possible, depending on whether one wants to work with typed or untyped versions of the lambda calculus. Furthermore, if we take the typed route, we end up in a wonderful playground called higher-order logic.

Now for an important question: have we been working with typed or untyped lambda calculus in this book? In a nutshell, we've been working with untyped lambda calculus (in particular, our implementation of β -conversion handles arbitrary untyped expressions) but we've been using it as if it were a typed system (that is, we made use of the standard Montague-style lambda expressions for the lexical entries). Why did we do this? Why didn't we define a system of typed lambda calculus right from the start?

There are two main reasons. For a start, we wanted to show the reader how naturally lambda calculus emerges as a computational solution to the semantic construction task—and we don't think that talking

about types adds much of interest to this story. But there is a more fundamental reason. As will become increasingly clear, logic in this book is viewed as a tool to be *used*, and in particular, to be used *computationally*. Now it is true that expressions such as $\lambda x.\text{MAN}(x)$ can be viewed as logical expressions, and that in theory this opens the door to applying our inference tasks at the subsentential level. Unfortunately, higher-order theorem proving is not as well developed as first-order theorem proving, so this theoretical possibility is of somewhat limited practical import at present. Hence, as we can't do much useful computation with higher-order logic, at present we prefer to downplay the logical perspective on lambda expressions and present them simply as useful glue. However we shall say a little more about higher-order theorem proving and its relevance to computational semantics in the Notes to Chapter 6.

The mathematical and logical literature on the lambda calculus is vast. Here are some of the more obvious points of entry. The bible of untyped lambda calculus is Barendregt (1984), and Barendregt (1992) contains a wealth of material on typed systems. However these are highly technical accounts, perhaps best suited for occasional reference. For a more approachable account of both typed and untyped systems, try Hindley and Seldin (1986). This is an extremely well-written book, and if you would like to learn more about the topics we have just mentioned (and related topics such as implicitly typed systems and combinatoric logic) there is probably no better introduction. Another useful source is Turner (1997); it is fairly technical, but broad in scope, and discusses why various sorts of typing are useful in linguistics. For logical work on the systems Montague himself used, and a number of interesting variants, the classic source is Gallin (1975). Another readable source of information here (with the added attraction of linguistic motivation) is Muskens (1996), while Fitting (2002) contains a superb introduction to higher-order tableaus. An excellent, wide ranging, discussion of higher-order logic can be found in Doets and Van Benthem (1983). Finally, it's worth remarking that some semanticists find the discipline imposed by the simply typed lambda calculus too rigid to deal properly with all aspects of natural language semantics; for more flexible approaches, see Chierchia et al. (1989).

Lambda calculus is the classic tool for semantic construction, but it's *not* the only tool available. One interesting alternative is the use of *attribute-value structures* (or *feature structures*) and *attribute-value structure unification* (or *feature structure unification*). The reader who wants to go further in computational semantics really should be acquainted with this approach. Gazdar and Mellish (1989) contains a

good text book level introduction to its use in building syntactic and semantic representations, and indeed, on its uses in pragmatics too. Another readable paper on the topic is Moore (1989). This paper compares the approach with the use of lambda calculus, and argues in favour of unification (for our views on lambda calculus versus unification-based approaches, see Blackburn and Bos (2003)). More recently, linear logic has been proposed as a glue language for semantic construction; see Dalrymple et al. (1997).

Finally, we want to remark that the grammar architectures of the type described in this chapter have proved extremely flexible in practice; in particular, they have been used to build Discourse Representation Structures (DRSs), the representation formalism used in Discourse Representation Theory, instead of standard first-order representations. This is important because DRSs make it easier to capture discourse level effects (such as anaphoric links) and certain pragmatic phenomenon (such as presupposition). For more information, see Blackburn et al. (2001a) and Bos (2004).

Underspecified Representations

This chapter develops methods for dealing with an important semantic phenomenon: *scope ambiguities*. Sentences with scope ambiguities are often semantically ambiguous (that is, they have at least two non-equivalent first-order representations) but fail to exhibit any syntactic ambiguity (that is, they have only one syntactic analysis). As our approach to semantic construction is based on the idea of using syntactic structure to guide semantic construction, we face an obvious problem here: if there is no syntactic ambiguity, we will only be able to build one of the possible representations. As scope ambiguities are common, we need to develop ways of coping with them right away.

We are going to investigate four different approaches to scope ambiguities: Montague's original method, two storage based methods, and a modern underspecification based approach called hole semantics. We will implement the storage based approaches and hole semantics.

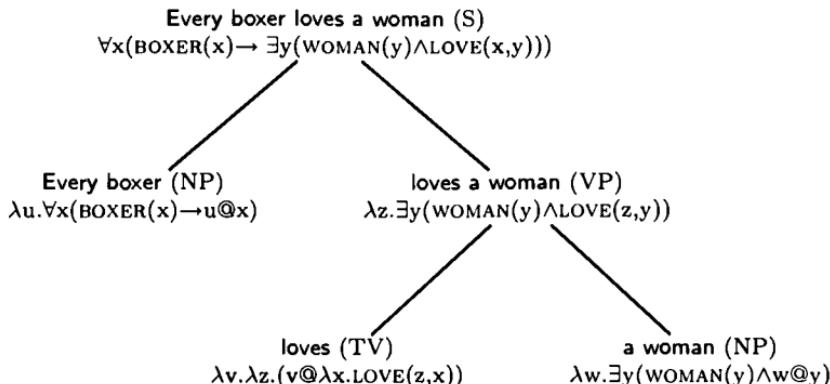
But as well as developing practical solutions to a pressing problem, this chapter tells an important story. Computational semanticists are adopting an increasingly abstract perspective on what representations are and how we should work with them. Once we have studied the evolutionary line leading from Montague's method to contemporary underspecification-based methods, we will be in a better position to appreciate why.

3.1 Scope Ambiguities

Scope ambiguity is a common phenomenon and can arise from many sources. In this chapter we will mostly be concerned with *quantifier scope ambiguities*. These are ambiguities that arise in sentences containing more than one quantifying noun phrase; for example, Every boxer loves a woman.

The methods of the previous chapter allow us to assign a represen-

tation to this sentence as follows:



The first-order formula we have constructed states that for each boxer there is a woman that he loves:

- (1) $\forall x(\text{BOXER}(x) \rightarrow \exists y(\text{WOMAN}(y) \wedge \text{LOVE}(x,y))).$

This representation permits different women to be loved by different boxers.

However, **Every boxer loves a woman** has a second meaning (or to use the linguistic terminology, a second *reading*) that is captured by the following formula:

- (2) $\exists y(\text{WOMAN}(y) \wedge \forall x(\text{BOXER}(x) \rightarrow \text{LOVE}(x,y))).$

This says that there is one woman who is loved by all boxers.

It is clear that these readings are somehow related, and that this relation has something to do with the relative scopes of the quantifiers: both first-order representations have the same components, but somehow the parts contributed by the two quantifying noun phrases have been shuffled around. In the first representation the existential quantifier contributed by **a woman** has ended up inside the scope of the universal quantifier contributed by **Every boxer** and in the second representation the nesting is reversed. It is common to say that in the first reading **Every boxer** *has scope over* (or *out-scopes*) **a woman**, while in reading (2) **a woman** *has scope over* **Every boxer**. Another common way of expressing this is to say that in the first reading **Every boxer** *has wide scope* and **a woman** *has narrow scope*; their roles are reversed in the second reading.

Unfortunately, these scoping possibilities are not reflected syntactically: the only plausible parse tree for this sentence is the one just shown. Thus while it makes good semantic sense to say that in reading (2) **a woman** out-scopes **Every boxer**, we can't point to any syntactic

structure that would explain why this scoping possibility exists. And as each word in the sentence is associated with a fixed lambda expression, and as semantic construction is simply functional application guided by the parse tree, this means there is no way for us to produce this second reading. This difficulty strikes at the very heart of our semantic construction methodology.

In this chapter we examine four increasingly sophisticated (and increasingly abstract) approaches to the problem. The first of these, Montague's original method, introduces some important ideas, but as it relies on the use of additional syntax rules it isn't compatible with the approach to grammar engineering adopted in this book. However, by introducing a more abstract form of representation—the *store*—we will be able to capture Montague's key ideas in a computationally attractive way. Stores were introduced by Robin Cooper and are arguably the earliest example of *underspecified representations*. Nonetheless, stores are relatively concrete. By simultaneously moving to more abstract underspecified representations, and replacing the essentially generative perspective underlying storage methods with a constraint-based perspective, we arrive at *hole semantics*. As we shall see, this more abstract view of what representations are, and how we should work with them, has advantages. For example, in hole semantics not only can we handle quantifier scope ambiguities, we can also handle the scope ambiguities created by constructs such as negation, and do so in a uniform way.

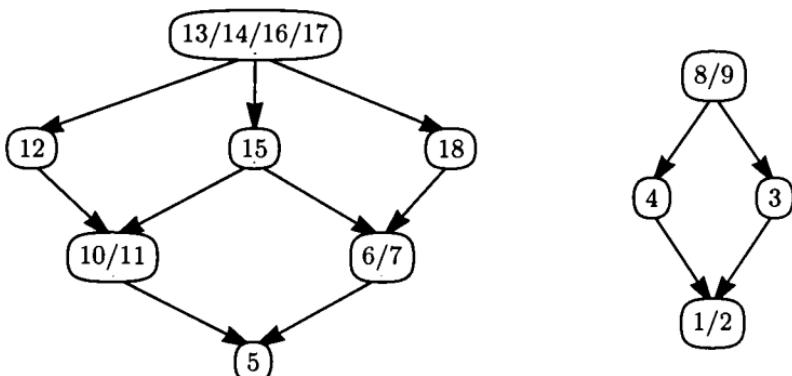
But before examining these solutions let's discuss the problem a little further. Some readers may be having doubts: are scope ambiguities *really* such a problem? Consider again *Every boxer loves a woman*. In a sense, representation (1) is sufficient to cover both readings of our example: it is 'weaker', since it is entailed by representation (2). Couldn't we argue that this weaker reading is the 'real' representation of the sentence, and that the stronger reading is pragmatically inferred from it with the help of contextual knowledge? Perhaps we don't need novel techniques for coping with quantifier ambiguity after all.

But while this idea is just about plausible for *Every boxer loves a woman*, it doesn't withstand closer scrutiny. Consider the sentence *Every owner of a hash bar gives every criminal a big kahuna burger*. This has 18 readings. Here they are:

- 1 $\forall x((\exists y(\text{HBAR}(y) \wedge \text{OF}(x,y)) \wedge \text{OWNER}(x)) \rightarrow \forall z(\text{CRIM}(z) \rightarrow \exists u(\text{BKB}(u) \wedge \text{GIVE}(x,z,u))))$
- 2 $\forall x(\text{CRIM}(x) \rightarrow \forall y((\exists z(\text{HBAR}(z) \wedge \text{OF}(y,z)) \wedge \text{OWNER}(y)) \rightarrow \exists u(\text{BKB}(u) \wedge \text{GIVE}(y,x,u))))$
- 3 $\forall x((\exists y(\text{HBAR}(y) \wedge \text{OF}(x,y)) \wedge \text{OWNER}(x)) \rightarrow \exists z(\text{BKB}(z) \wedge \forall u(\text{CRIM}(u) \rightarrow \text{GIVE}(x,u,z))))$
- 4 $\forall x(\text{CRIM}(x) \rightarrow \exists y(\text{BKB}(y) \wedge \forall z((\exists u(\text{HBAR}(u) \wedge \text{OF}(z,u)) \wedge \text{OWNER}(z)) \rightarrow \text{GIVE}(z,x,y))))$
- 5 $\forall x(\text{CRIM}(x) \rightarrow \exists y(\text{HBAR}(y) \wedge \forall z((\text{OF}(z,y) \wedge \text{OWNER}(z)) \rightarrow \exists u(\text{BKB}(u) \wedge \text{GIVE}(z,x,u))))$

- 6 $\forall x(\text{CRIM}(x) \rightarrow \exists y(\text{HBAR}(y) \wedge \exists z(\text{BKB}(z) \wedge \forall u((\text{OF}(u,y) \wedge \text{OWNER}(u)) \rightarrow \text{GIVE}(u,x,z))))$
- 7 $\forall x(\text{CRIM}(x) \rightarrow \exists y(\text{BKB}(y) \wedge \exists z(\text{HBAR}(z) \wedge \forall u((\text{OF}(u,z) \wedge \text{OWNER}(u)) \rightarrow \text{GIVE}(u,x,y))))$
- 8 $\exists x(\text{BKB}(x) \wedge \forall y((\exists z(\text{HBAR}(z) \wedge \text{OF}(y,z)) \wedge \text{OWNER}(y)) \rightarrow \forall u(\text{CRIM}(u) \rightarrow \text{GIVE}(y,u,x)))$
- 9 $\exists x(\text{BKB}(x) \wedge \forall y(\text{CRIM}(y) \rightarrow \forall z((\exists u(\text{HBAR}(u) \wedge \text{OF}(z,u)) \wedge \text{OWNER}(z)) \rightarrow \text{GIVE}(z,y,x)))$
- 10 $\exists x(\text{HBAR}(x) \wedge \forall y((\text{OF}(y,x) \wedge \text{OWNER}(y)) \rightarrow \forall z(\text{CRIM}(z) \rightarrow \exists u(\text{BKB}(u) \wedge \text{GIVE}(y,z,u))))$
- 11 $\exists x(\text{HBAR}(x) \wedge \forall y(\text{CRIM}(y) \rightarrow \forall z((\text{OF}(z,x) \wedge \text{OWNER}(z)) \rightarrow \exists u(\text{BKB}(u) \wedge \text{GIVE}(z,y,u))))$
- 12 $\exists x(\text{HBAR}(x) \wedge \forall y((\text{OF}(y,x) \wedge \text{OWNER}(y)) \rightarrow \exists z(\text{BKB}(z) \wedge \forall u(\text{CRIM}(u) \rightarrow \text{GIVE}(y,u,z))))$
- 13 $\exists x(\text{HBAR}(x) \wedge \exists y(\text{BKB}(y) \wedge \forall z((\text{OF}(z,x) \wedge \text{OWNER}(z)) \rightarrow \forall u(\text{CRIM}(u) \rightarrow \text{GIVE}(z,u,y))))$
- 14 $\exists x(\text{BKB}(x) \wedge \exists y(\text{HBAR}(y) \wedge \forall z((\text{OF}(z,y) \wedge \text{OWNER}(z)) \rightarrow \forall u(\text{CRIM}(u) \rightarrow \text{GIVE}(z,u,x))))$
- 15 $\exists x(\text{HBAR}(x) \wedge \forall y(\text{CRIM}(y) \rightarrow \exists z(\text{BKB}(z) \wedge \forall u((\text{OF}(u,x) \wedge \text{OWNER}(u)) \rightarrow \text{GIVE}(u,y,z))))$
- 16 $\exists x(\text{HBAR}(x) \wedge \exists y(\text{BKB}(y) \wedge \forall z(\text{CRIM}(z) \rightarrow \forall u((\text{OF}(u,x) \wedge \text{OWNER}(u)) \rightarrow \text{GIVE}(u,z,y))))$
- 17 $\exists x(\text{BKB}(x) \wedge \exists y(\text{HBAR}(y) \wedge \forall z(\text{CRIM}(z) \rightarrow \forall u((\text{OF}(u,y) \wedge \text{OWNER}(u)) \rightarrow \text{GIVE}(u,z,x))))$
- 18 $\exists x(\text{BKB}(x) \wedge \forall y(\text{CRIM}(y) \rightarrow \exists z(\text{HBAR}(z) \wedge \forall u((\text{OF}(u,z) \wedge \text{OWNER}(u)) \rightarrow \text{GIVE}(u,y,x))))$.

Some of these readings turn out to be logically equivalent. In fact, we have the following sets of logically equivalent readings: $\{1, 2\}$, $\{8, 9\}$, $\{6, 7\}$, $\{10, 11\}$ and $\{13, 14, 16, 17\}$. If we take these equivalences into account, we are left with 11 distinct readings. Moreover, if we examine these readings closely and determine their logical relationships we discover that they are partitioned into two logically unrelated groups, as shown in the following diagram:



The arrows in the diagram represent logical implication. Note that each group has a strongest reading (namely $\{13, 14, 16, 17\}$ and $\{8, 9\}$), and a weakest reading (namely 5 and $\{1, 2\}$), but there is no single weakest reading that covers all the possibilities. It is difficult to see how a pragmatic approach could account for this example. (Incidentally, we did *not*, as the reader will doubtless be relieved to hear, have to compute these readings and determine their logical relationships by hand. In fact, all we had to do was combine the semantic construction

methods discussed in this chapter with the inference tools discussed in Chapter 5. In Chapter 6 we show the reader exactly how this can be done.)

Indeed, the idea of computing the weakest reading and relying on pragmatics for the rest faces difficulties even when the weakest reading exists, for there is no guarantee that it is the weakest reading that will be generated by the methods of the previous chapter. Consider the sentence **A boxer is loved by every woman**. This has two possible readings, namely the stronger reading

$$\exists y(\text{BOXER}(y) \wedge \forall x(\text{WOMAN}(x) \rightarrow \text{LOVE}(x,y)))$$

and the weaker reading

$$\forall x(\text{WOMAN}(x) \rightarrow \exists y(\text{BOXER}(y) \wedge \text{LOVE}(x,y))).$$

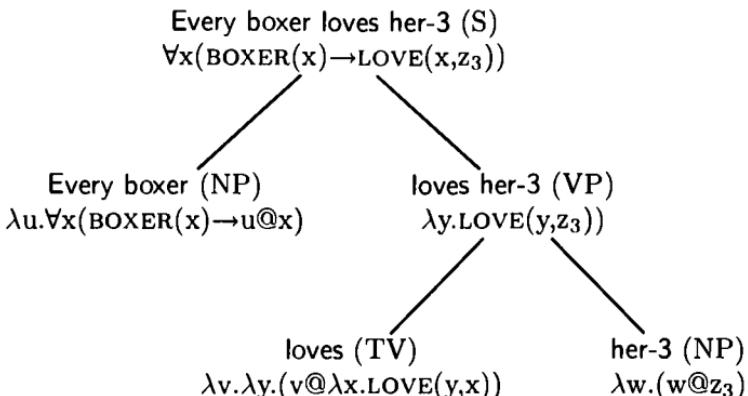
If we use the direct approach to semantic construction (that is, semantic construction is simply functional application guided by the parse tree) in the obvious way, we will generate the stronger reading.

3.2 Montague's Approach

As our discussion has made clear, scope ambiguities are a genuine problem, and if we want to compute semantic representations we need a solution. In the remainder of this chapter we shall discuss four. Starting with Richard Montague's pioneering work, we shall work our way, via storage methods, towards a modern underspecification technique called hole semantics. As we shall see, there is a pleasing continuity to the way the story unfolds: each development ushers in the next.

Classical Montague semantics makes use of (and indeed, is the source of) the direct method of semantic construction studied in the previous chapter. However, motivated in part by quantifier scope ambiguities, Montague also introduced a rule of quantification (often called *quantifier raising*) that allowed a more indirect approach. The basic idea is simple. Instead of directly combining syntactic entities with the quantifying noun phrase we are interested in, we are permitted to choose an ‘indexed pronoun’ and to combine the syntactic entity with the indexed pronoun instead. Intuitively, such indexed pronouns are ‘placeholders’ for the quantifying noun phrase. When this placeholder has moved high enough in the tree to give us the scoping we are interested in, we are permitted to replace it by the quantifying NP of interest.

As an example, let's consider how to analyse **Every boxer loves a woman**. Here's the first part of the tree we need:

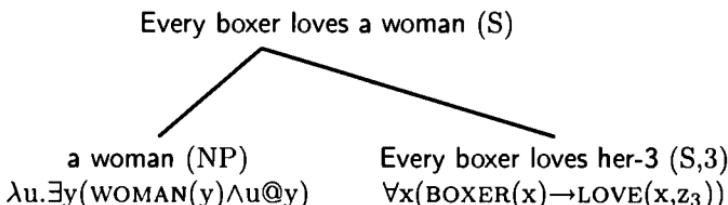


Instead of combining *loves* with the quantifying term *a woman* we have combined it with the placeholder pronoun *her-3*. This pronoun bears an ‘index’, namely the numeral ‘3’. The placeholder pronoun is associated with a ‘semantic placeholder’, namely $\lambda w. (w@z_3)$. As we shall see, it is the semantic placeholder that does most of the real work for us. Note that the pronoun’s index appears as subscript on the free variable in the semantic placeholder. From a semantic perspective, choosing an indexed pronoun really amounts to opting to work with the semantic placeholder (instead of the semantics of the quantifying NP) and stipulating which free variable the semantic placeholder should contain.

Now, the key point the reader should note about this tree is how *ordinary* it is. True, it contains a weird looking pronoun *her-3*—but, that aside, it’s just the sort of structure we’re used to. For a start, the various elements are syntactically combined in the expected way. Moreover $\lambda w. (w@z_3)$, the semantic representation associated with the placeholder pronouns, should look familiar. Recall that the semantic representation for *Mia* is $\lambda w. (w@MIA)$. Thus the placeholder pronoun is being given the semantics of a proper name, but with an individual variable (here z_3) instead of a constant, which seems sensible. Moreover (as the reader should check) the representations for *loves her-3* and *Every boxer loves her-3* are constructed using functional application just as we discussed in the previous chapter. In short, although some of the representations used are unorthodox, they are combined with the other representations in the orthodox way.

Now for the next step. We want to ensure that *a woman* out-scopes *Every boxer*. By using the placeholder pronoun *her-3*, we have delayed introducing *a woman* into the tree. But *Every boxer* is now firmly in place, so if we replaced *her-3* by *a woman* we would have the desired scoping relation. Predictably, there is a rule that lets us do this: given

a quantifying NP, and a sentence containing a placeholder pronoun, we are allowed to construct a new sentence by substituting the quantifying NP for the placeholder. In short, we are allowed to extend the previous tree as follows:

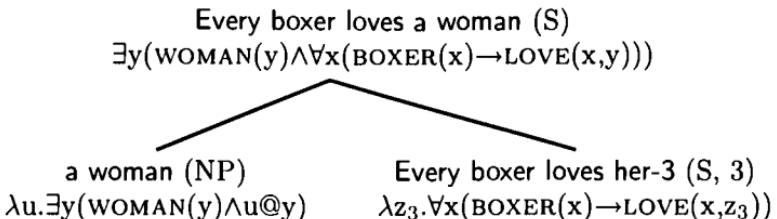


But what's happening semantically? We know which formula we want to be assigned to the node (namely, $\exists y(\text{WOMAN}(y) \wedge \forall x(\text{BOXER}(x) \rightarrow \text{LOVE}(x,y)))$) but how can we ensure it gets there? Let's think the matter through.

We want **a woman** to take wide scope over **Every boxer** semantically. Hence we should use the semantic representation associated with **a woman** as the function. (To see this, simply look at the form of its semantic representation. When we apply it to an argument and perform β -conversion, we will be left with an existentially quantified expression, which is what we want.) But what should its argument be? There is only one reasonable choice. It must be the representation associated with **Every boxer loves her-3** lambda abstracted with respect to z_3 :

$$(3) \quad \lambda z_3. \forall x(\text{BOXER}(x) \rightarrow \text{LOVE}(x,z_3)).$$

Why is this? Well, right at the bottom of the tree we made use of the semantic placeholder $\lambda u.(u@z_3)$. When we raised this placeholder up the tree using functional application, we were essentially ‘recording’ what the semantic representation of **a woman** would have encountered if we had used it directly. (Remember, we did nothing unusual, either syntactically or semantically, during the raising process.) The formula $\forall x(\text{BOXER}(x) \rightarrow \text{LOVE}(x,z_3))$ is the record of these encounters. When we are ready to ‘play back’ this recorded information, we lambda abstract with respect to z_3 (thus indicating that this variable is the crucial one, the one originally chosen) and feed the resulting expression as an argument to the semantic representation of **a woman**. β -conversion will glue this record into its rightful place, and, as the following tree shows, everything will work out just right:



That's it. Summing up, Montague's approach makes use of syntactic and semantic placeholders so that we can place quantifying NPs in parse trees at exactly the level required to obtain the desired scope relations. A neat piece of 'lambda programming' (we call it Montague's trick) ensures that the semantic information recorded by the placeholder is re-introduced into the semantic representation correctly.

But will Montague's approach help us computationally? As it stands, no. Why not? Because it does not mesh well with the grammar engineering principles adopted in this book. We want semantic construction methods which we can bolt onto pretty much any grammar which produces parse trees for a fragment of natural language. Moreover, we never want to be forced to modify a grammar: ideally we'd like to be able to treat the syntactical component as a black box, hook our semantic representations onto it, and then compute.

But Montague's approach doesn't work that way. To apply his method directly, we have to add extra syntax rules to our grammars, such as the rules for introducing placeholder pronouns and for eliminating placeholder pronouns in favour of quantifying noun phrases, that we saw in our example. And if we wanted to deal with more serious scope problems (for example, the interaction of quantifier scope ambiguities with negation) we would probably have to add a lot of extra rules as well. This is not only hard work, it seems linguistically misguided: grammar rules are there to tell us about syntactic structure—but now we're using them to manipulate mysterious-looking placeholder entities in a rather ad-hoc looking attempt to reduce scope issues to syntax.

But while we won't use Montague's approach directly, we will use it indirectly. For as we shall now see, it is possible to exploit Montague's key insights in a different way.

3.3 Storage Methods

Storage methods are an elegant way of coping with quantifier scope ambiguities: they neatly decouple scope considerations from syntactic issues, making it unnecessary to add new grammar rules. Moreover, both historically and pedagogically, they are the natural approach to explore next, for they draw on the key ideas of Montague's approach

(in essence, they exploit semantic placeholders and Montague's trick in a computationally natural way) and at the same time they anticipate key themes of modern underspecification-based methods.

Cooper Storage

Cooper storage is a technique developed by Robin Cooper for handling quantifier scope ambiguities. In contrast to the work of the previous section, semantic representations are built directly, without adding to the basic syntax rules. The key idea is to associate each node of a parse tree with a *store*, which contains a 'core' semantic representation together with the quantifiers associated with nodes lower in the tree. After the sentence is parsed, the store is used to generate scoped representations. The order in which the stored quantifiers are retrieved from the store and combined with the core representation determines the different scope assignments.

To put it another way, instead of simply associating nodes in parse trees with a single lambda expression (as we have done until now), we are going to associate them with a core semantic representation, together with the information required to turn this core into the kinds of representation we are familiar with. Viewed from this perspective, stores are simply a more abstract form of semantic representation—representations which encode, compactly and without commitment, the various scope possibilities; in short, they are a simple form of underspecified representation.

Let's make these ideas precise. Formally, a store is an n -place sequence. We represent stores using the angle brackets $\langle \cdot \rangle$ and \cdot . The first item of the sequence is the core semantic representation; it's simply a lambda expression. (Incidentally, if we wanted to, we could insist that we've been using stores all along: we need merely say that when we previously talked of assigning a lambda expression ϕ to a node, we *really* meant that we assigned the 1-place store $\langle \phi \rangle$ to a node.) Subsequent items in the sequence (if any) are pairs (β, i) , where β is the semantic representation of an NP (that is, another lambda expression) and i is an index. An index is simply a label which picks out a free variable in the core semantic representation. As we shall see, this index has the same purpose as the indexes we used for Montague-style raised quantifiers. We call pairs (β, i) *indexed binding operators*.

How do we use stores for semantic construction? Unsurprisingly, the story starts with the quantified noun phrases (that is, noun phrases formed with the help of a determiner). Instead of simply passing on the store assigned to them, quantified noun phrases are free to add new information to it before doing so. (Other sorts of NPs, such as

proper names, aren't allowed to do this.) More precisely, quantified noun phrases are free to make use of the following rule:

Storage (Cooper)

If the store $\langle \phi, (\beta, j), \dots, (\beta', k) \rangle$ is a semantic representation for a quantified NP, then the store $\langle \lambda u. (u@z_i), (\phi, i), (\beta, j) \dots (\beta', k) \rangle$, where i is some unique index, is also a representation for that NP.

The crucial thing to note is that the index associated with ϕ is identical with the subscript on the free variable in $\lambda u. (u@z_i)$. (After all, if we decide to store ϕ away for later use, it's sensible to keep track of what its argument is.)

In short, from now on when we encounter a quantified NP we will be faced by a choice. We can either pass on ϕ (together with any previously stored information) straight on up the tree, or we can decide to use $\lambda u. (u@z_i)$ as the core representation, store the quantifier ϕ on ice for later use (first taking care to record which variable it is associated with) and pass on this new package. The reader should be experiencing a certain feeling of *deja vu*. We're essentially using the pronoun representation $\lambda u. (u@z_i)$ as a semantic placeholder, just as we did in the previous section. Indeed, as will presently become clear, our shiny new storage technology re-uses the key ideas of quantifier storage in a fairly direct way.

Incidentally, the storage rule is *not* recursive. It offers a simple two way choice: either pass on the ordinary representation (that is, the store $\langle \phi, (\beta, j), \dots, (\beta', k) \rangle$) or use the storage rule to form

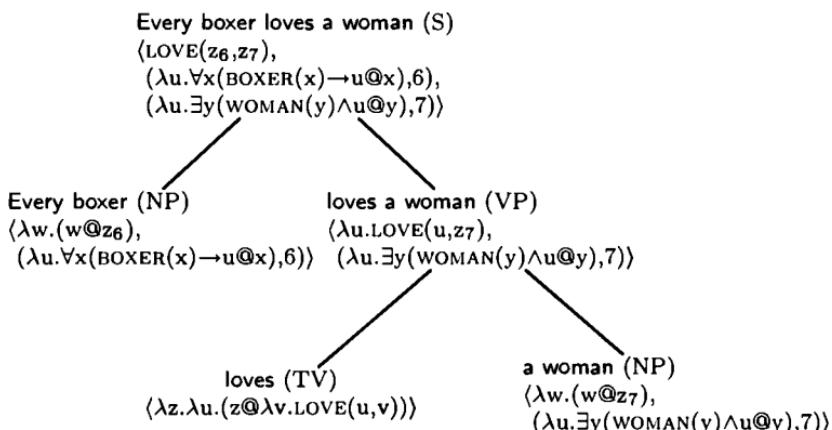
$$\langle \lambda u. (u@z_i), (\phi, i), (\beta, j) \dots (\beta', k) \rangle$$

and pass this new store on up instead. We're *not* offered—and we don't want or need—the option of reapplying the storage rule to this new store to form $\langle \lambda u. (u@z_m), (\lambda u. (u@z_i), m), (\phi, i), (\beta, j), \dots, (\beta', k) \rangle$. Intuitively, we're offered a straight choice between keeping the lambda expression associated with the quantified NP in the active part of the memory (that is, in the first slot of the store) or placing it, suitably indexed, in the freezer for later consumption.

It's time for an example. Let's analyse Every boxer loves a woman using Cooper storage. The relevant tree is given below; let's think about how it was built. First, note that the nodes for a woman and Every boxer are both associated with 2-place stores. Why is this? Consider the node for a woman. We know from our previous work that the lambda expression associated with a woman is $\lambda u. \exists y (\text{WOMAN}(y) \wedge u@y)$. In our new representations-are-stores world view, this means that the 1-place store

$$\langle \lambda u. \exists y (\text{WOMAN}(y) \wedge u@y) \rangle$$

is a legitimate interpretation for the NP *a woman*. But remember—this is not our only option. We are free to use the storage rule, and this is what we did when building the tree below: we picked a brand new free variable (namely z_7), used the placeholder $\lambda w.(w@z_7)$ as the first item in the store, and ‘iced’ $\lambda u.\exists y(\text{WOMAN}(y) \wedge u@y)$, first recording the fact that z_7 is the variable relevant to this expression. Note that essentially the same story could be told for the NP *Every boxer*, save that there we chose the new free variable z_6 .



Once this has been grasped, the rest is easy. In particular, if a functor node \mathcal{F} is associated with a store $\langle\phi, (\beta', j), \dots, (\beta', k)\rangle$ and its argument node \mathcal{A} is associated with the store $\langle\psi, (\beta'', l), \dots, (\beta'', m)\rangle$, then the store associated with the node \mathcal{R} whose parts are \mathcal{F} and \mathcal{A} is

$$\langle\phi @ \psi, (\beta', j), \dots, (\beta', k), (\beta'', l), \dots, (\beta'', m)\rangle.$$

That is, the first slot of the store really is the active part: it’s where the core representation is built. If you examine the above tree, you’ll see that the stores associated with *loves a woman* and *Every boxer loves a woman* were formed using this functional-application-in-the-first-slot method. Note that in both cases we’ve simplified $\phi @ \psi$ using β -conversion.

But we’re not yet finished. We now have a sentence, and this sentence is associated with an abstract unscoped representation (that is, a store), but of course, at the end of the day we really want to get our hands on some ordinary scoped first-order representations. How do we do this?

This is the task of *retrieval*, a rule which is applied to the stores associated with sentences. Retrieval removes one of the indexed binding operators from the freezer, and combines it with the core representation to form a new core representation. (If the freezer is empty, then the store associated with the S node must already be a 1-place sequence, and thus

we already have the expression we are looking for.) It continues to do this until it has used up all the indexed binding operators. The last core representation obtained in this way will be the desired scoped semantic representation.

What does the combination process involve? Suppose retrieval has removed a binding operator indexed by i . It lambda abstracts the core semantic representation with respect to the variable bearing the subscript i , and then functionally applies the newly retrieved binding operator to the newly lambda-abstracted-over core representation. The result is the new core representation, and it replaces the old core representation as the first item in the store. More precisely:

Retrieval (Cooper)

Let σ_1 and σ_2 be (possibly empty) sequences of binding operators. If the store $\langle \phi, \sigma_1, (\beta, i), \sigma_2 \rangle$ is associated with an expression of category S, then the store $\langle \beta @ \lambda z_i. \phi, \sigma_1, \sigma_2 \rangle$ is also associated with this expression.

Hey—we're simply performing Montague's trick with the aid of stores!

Let's return to our example and apply the retrieval rule to the store associated with the S node. Now, this store contains two indexed binding operators. The retrieval rule allows us to remove either of them and to combine it with the core representation. Suppose we choose to first retrieve the quantifier for Every boxer (that is, the indexed binding operator in the second slot of the store). Then the retrieval rule tells us that the following store must be associated with the S node:

$$\langle \lambda u. \forall x (\text{BOXER}(x) \rightarrow u @ x) @ \lambda z_6. \text{LOVE}(z_6, z_7), (\lambda u. \exists y (\text{WOMAN}(y) \wedge u @ y), 7) \rangle.$$

Using β -conversion, this simplifies to:

$$\langle \forall x (\text{BOXER}(x) \rightarrow \text{LOVE}(x, z_7)), (\lambda u. \exists y (\text{WOMAN}(y) \wedge u @ y), 7) \rangle.$$

No more β -conversions are possible, but there's still a quantifier left in store. Retrieving it produces:

$$\langle \lambda u. \exists y (\text{WOMAN}(y) \wedge u @ y) @ \lambda z_7. \forall x (\text{BOXER}(x) \rightarrow \text{LOVE}(x, z_7)) \rangle.$$

The result is the reading where a woman out-scopes Every boxer, as becomes clear if we perform two more β -conversions to obtain:

$$\langle \exists y (\text{WOMAN}(y) \wedge \forall x (\text{BOXER}(x) \rightarrow \text{LOVE}(x, y))) \rangle.$$

How do we get the other reading? We simply retrieve the quantifiers in the other order. We suggest that the reader works through the details.

Implementing Cooper Storage

We are ready to turn to computation: how do we implement Cooper storage in Prolog?

The first steps are straightforward. We'll represent indexed binding operators as terms of the form `bo(Quant, Index)`. In such terms `Quant` will be the (Prolog representation of) a lambda expression, and `Index` will be a Prolog variable. Why use Prolog variables for indexes? Because it's so simple. Recall that Prolog internally represents variables by means of such expressions such as `_G218` and `_G328`. That is, Prolog variables carry numerical indexes, so we might as well make use of this.

Stores will be represented as lists: the head will be a lambda expression, the tail (if non-empty) a list of binding operators. So the following Prolog list represents a store:

```
[walk(X), bo(lam(P, all(Y, imp(boxer(Y), app(P, X)))), X)].
```

But now we need to think a little. The semantic representations we want to work with are stores, not just plain lambda expressions. So we will need to associate stores, not lambda expressions, with the lexical items. Moreover, we will need to define how stores are to be combined, and ensure that all retrieval possibilities are tried out.

How are we to do this? Here's where our grammar engineering architecture comes to our aid: we can take care of these requirements by making three kinds of changes, in three separate locations. Let's go through them all in turn.

Our first task is to make store-based semantic representations available to the lexical items. Recall that we abstracted the semantic representation patterns we associate with the words into what we called semantic macros. The semantic expressions for lambda-based representations were stored in the file `semLexLambda.pl` and a typical macro looked like this:

```
semLex(iv,M):-  
    M = [symbol:Sym,  
          sem:lam(X,Formula)],  
          compose(Formula,Sym,[X]).
```

So it's clear what we should do: create a new file (let's call it `semLexStorage.pl`) for our storage-based semantic macros. Here's the new semantic macro for intransitive verbs:

```
semLex(iv,M):-  
    M = [symbol:Sym,  
          sem:[lam(X,Formula)]],  
          compose(Formula,Sym,[X]).
```

Note that this macro is virtually identical with the lambda-based version (the only difference is the list brackets around the semantic representation). Indeed, *all* the storage-based macros differ from their lambda-based counterparts in precisely this way. Thus we have com-

pleted our first task, and done so effortlessly.

Our second task is to provide a new set of semantic rules defining how stores are to be combined. Now, recall our discussion of grammar engineering at the end of the previous chapter. There we created a file called `semRulesLambda.pl` containing such items as

```
combine(vp:app(A,B),[tv:A,np:B]).
```

which defined the apply-function-to-argument semantic rules used earlier. So, once again, it's clear what we should now do: create a new file (let's call it `semRulesCooper.pl`) which contains new definitions of `combine/2` suitable for working with stores.

Let's get to work. Actually, in many cases there is little to do. For example, here's the new store-based clause for verb phrases:

```
combine(vp:[app(A,B)|S],[tv:[A],np:[B|S]]).
```

This is a straightforward enhancement of our old lambda-based version:

```
combine(vp:app(A,B),[tv:A,np:B]).
```

Sometimes, however, we need to do a little more; in particular, sometimes we need to append stores. The clause for coordination shows what is involved in such cases:

```
combine(np:[app(app(B,A),C)|S3],[np:[A|S1],coord:[B],np:[C|S2]]):-  
appendLists(S1,S2,S3).
```

But the most interesting rules are those for noun phrases formed by determiners and nouns (these license the basic store-or-carry-on-as-usual choice underlying the storage method) and the sentential rule which licenses retrieval. We'll look at both. Here are the semantic rules for combining determiners and nouns:

```
combine(np:[!am(P,app(P,X)),bo(app(A,B),X)|S],[det:[A],n:[B|S]]).  
combine(np:[app(A,B)|S],[det:[A],n:[B|S]]).
```

The content of these rules should be clear: the first puts the quantified noun phrase on the store, the second carries out functional application. But one remark is worth making. We have made pushing a quantifier onto the store an *optional* operation; that's why there are two semantic rules for the same syntactic rule above. Now, we leave the reader to investigate why it is linguistically desirable to make storage optional (see Exercise 3.3.4). But note that making it optional can lead to unnecessary work. Consider **Every boxer loves a woman**. If storage is optional, there are actually *five* strategies that can be used when trying to build representations for this sentence: leave both quantified noun phrases in place, just store **Every boxer**, just store **a woman**, store both and retrieve **Every boxer** first, or store both and retrieve **a woman** first. But

this sentence only has two logically distinct readings, so three of these options don't lead to anything new (in fact, they'll simply be alphabetic variants of the two basic readings). Nonetheless, optionality is the linguistically sensible path, so we'll just have to find ways of coping with such redundancies. At the end of the section we'll see how to define a filter that will eliminate alphabetic variants.

What about the semantic rule which licenses retrieval at the sentential level? Here's what we require:

```
combine(s:S, [np:[A|S1], vp:[B|S2]]):-  
    appendLists(S1, S2, S3),  
    sRetrieval([app(A,B)|S3], Retrieved),  
    betaConvert(Retrieved, S).
```

With this rule defined, our second task is completed (we'll shortly see how `sRetrieval/2` is defined).

Now for the third task: we have to ensure that this new store-based information is correctly integrated, and that all retrieval possibilities are tried out. Recall that in the previous chapter we created a top-level file called `lambda.pl` which coordinated the lexicon, the syntax rules, the semantic macros, and the semantic rules. Here we follow the same strategy: we shall define a file called `cooperStorage.pl` which integrates our (standard) lexicon and syntax rule with our (new) storage based semantic rules and macros. This file will contain the predicate that ensures that all retrieval possibilities are systematically explored (and the filter for eliminating alphabetic variants).

The Prolog predicate that copes with retrieval is `sRetrieval/2`. Its first argument is a store, its second argument is the derived scoped representation. If the store contains just one element, then this is the scoped formula—the first clause deals with this case:

```
sRetrieval([S], S).
```

Otherwise, this predicate takes a quantifier from the store using the `selectFromList/3` predicate, lambda abstracts `Sem` (the first item in the list) with respect to the retrieved variable, and applies the retrieved quantifier to the result. This process yields a new semantic representation that will be β -converted later. Note that the `sRetrieval/2` predicate is recursive, thus it will eventually reduce the store to a list containing just one item, namely a scoped representation:

```
sRetrieval([Sem|Store], S):-  
    selectFromList(b0(Q,X), Store, NewStore),  
    sRetrieval([app(Q, lam(X, Sem))|NewStore], S).
```

But why does this generate *all* the quantifier scope representations?

This is thanks to `selectFromList/3`. This predicate (which is included in `comsemPredicates.pl`) is defined in such a way that, if there is more than one element in the store, it succeeds when it removes *any* element at all from it. Therefore, `sRetrieval/2` produces (via the Prolog backtracking mechanism) all the scoped representations possible.

With `sRetrieval/2` at our disposal we are ready to define a top-level predicate `cooperStorage/0` that wraps everything together. And as we promised earlier, we shall include a filter to ensure that we don't output unnecessary alphabetic variants. Here's the wrapper:

```
cooperStorage :-  
    readLine(Sentence),  
    setof(Sem, t([sem:Sem], Sentence, []), Sems1),  
    filterAlphabeticVariants(Sems1, Sems2),  
    printRepresentations(Sems2).
```

That is, we use the `readLine/1` predicate discussed in the last chapter to read the sentence typed by the user, and then use the standard Prolog `setof/3` predicate to gather all the readings. With this set gathered, we filter out the alphabetic variants, and then print the representations.

How do we filter the alphabetic variants? With the help of the `alphabeticVariants/2` predicate defined in the previous chapter:

```
filterAlphabeticVariants(L1,L2) :-  
    selectFromList(X,L1,L3),  
    memberList(Y,L3),  
    alphabeticVariants(X,Y), !,  
    filterAlphabeticVariants(L3,L2).  
  
filterAlphabeticVariants(L,L).
```

That is, we use the `selectFromList/3` predicate to select an element `X` from the input list `L1` to produce the list `L3`. We then use `memberList/2` to select an element `Y` from `L3`. At this point we make use of `alphabeticVariants/2` to test whether `X` and `Y` are variants. Now, first consider what happens if they are *not* alphabetic variants. This forces backtracking, so another potential candidate `Y` will be selected by `memberList/2`, and again we test for alphabetic variance. If none of the candidate `Y`s so generated are alphabetic variants of `X` then backtracking will force us even further back, namely to `selectFromList/3`; we generate another candidate `X`, and try again. On the other hand, if some choice of `Y` is an alphabetic variant of one of our selected `X`s, then we cut (thus killing further backtracking at this level) and recursively carry out the filtration process to the list `L3` (note that `L3` will contain one item less than `L1`). This process continues, recursively removing all

alphabetic variants, and ultimately the first clause must fail (why?). At this point the second clause trivially succeeds, and we are left with a list of representations of which no two elements are alphabetic variants.

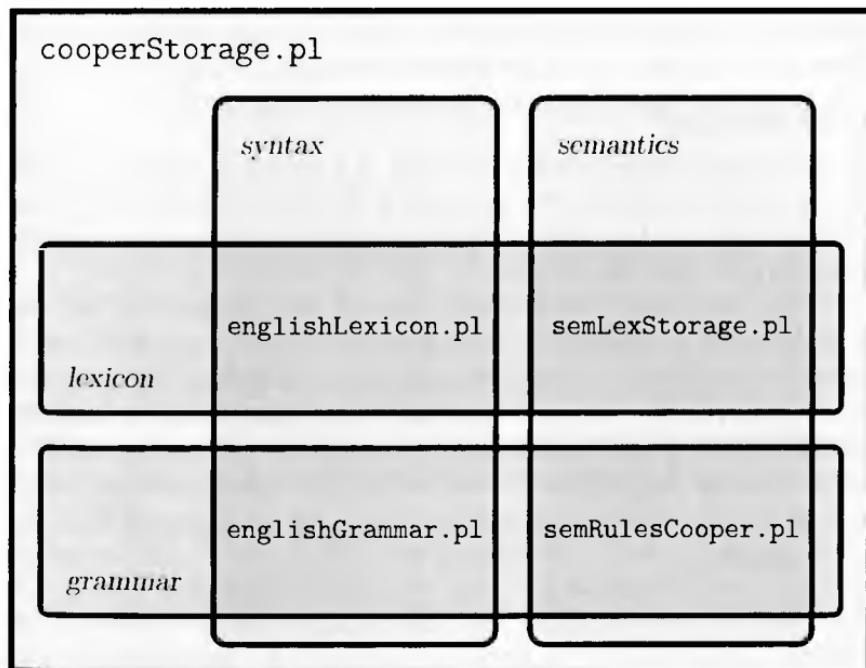
Here's an example of our program at work:

?- `cooperStorage.`

> `Every boxer loves a woman.`

```
1 all(A,imp(boxer(A),some(B, and(woman(B),love(A,B)))))  
2 some(A, and(woman(A),all(B,imp(boxer(B),love(B,A)))))
```

Summing up, Cooper storage is a more abstract version of Montague's approach to quantification that makes use of special representations called stores. From the perspective of computational semantics, it has a distinct advantage over Montague's approach: it doesn't require additional syntax rules. Indeed, when you get down to it, all we really needed to do to cope with Cooper storage was to define new semantic macros, new semantic rules, and wrap it all together. The diagram below summarises our Cooper storage setup.



Exercise 3.3.1 Try the Cooper storage implementation without filtering out alphabetic variants. What happens and why?

Exercise 3.3.2 How many scoped representations are retrieved for **Every piercing that is done with a needle is okay?** (Take **is done with** as a two place relation, and **view is okay** as a one place predicate.) Are they all correct?

Exercise 3.3.3 Extend the storage analysis to ditransitive verbs, and check how many readings it gives for sentences like **A boxer gives every woman a foot massage.** Are these the readings you would expect?

Exercise 3.3.4 Why should storage be optional? Think about the way quantified noun phrases interact with negation. More concretely, think about the possible readings of the sentence **Every boxer doesn't love a woman.** How many readings does this sentence have? How many readings does Cooper storage assign to this sentence if storage is optional? And if storage is not optional?

Exercise 3.3.5 According to some linguistic theories of quantifier scope, quantified noun phrases cannot be raised out of a relative clause, and hence cannot out-scope anything outside the clause. For instance, in the complex noun phrase **a woman that knows every boxer**, it is impossible to get a reading where **every boxer** out-scopes **a woman**. Change the semantic rule for relative clauses so that quantified noun phrases stay in the scope of relative clauses. (You need to do this in the file **semRulesCooper.pl**.)

Keller Storage

Cooper storage allows us a great deal of freedom in retrieving information from the store. We are allowed to retrieve quantifiers in any order we like, and the only safety-net provided is the use of co-indexed variables and Montague's trick.

Is this really safe? We haven't spotted any problems so far—but then we've only discussed one kind of scope ambiguity, namely those in sentences containing a transitive verb with quantifying NPs in subject and object position. However, there are lots of other syntactic constructions that give rise to quantifier scope ambiguities, for instance relative clauses (4) and prepositional phrases in complex noun phrases (5):

- (4) **Every piercing that is done with a gun goes against the entire idea behind it.**
- (5) **Mia knows every owner of a hash bar.**

Both examples give rise to scope ambiguities. For example, in (5) there is a reading where **Mia** knows all owners of (possibly different) hash bars, and a reading where **Mia** knows all owners that own one and the same hash bar. Moreover, both examples contain nested NPs. In the first example **a gun** is a sub-NP of **Every piercing that is done with**

a gun, while in the second, a hash bar is a sub-NP of every owner of a hash bar.

We've never had to deal with nested NPs before. Is Cooper storage delicate enough to cope with them, and does it allow us to generate all possible readings? Let's examine example (5) more closely and find out. This is the store (as the reader should verify):

$$\langle \text{KNOW}(\text{MIA}, z_2), \\ (\lambda u. \forall y (\text{OWNER}(y) \wedge \text{OF}(y, z_1) \rightarrow u @ y), 2), \\ (\lambda w. \exists x (\text{HASHBAR}(x) \wedge w @ x), 1) \rangle.$$

There are two ways to perform retrieval: by pulling the universal quantifier off the store before the existential, or vice versa. Let's explore the first possibility. Pulling the universal quantifier off the store yields (after β -conversion):

$$\langle \forall y (\text{OWNER}(y) \wedge \text{OF}(y, z_1) \rightarrow \text{KNOW}(\text{MIA}, y)), \\ (\lambda w. \exists x (\text{HASHBAR}(x) \wedge w @ x), 1) \rangle.$$

Retrieving the existential quantifier then yields (again, after β -conversion):

$$\langle \exists x (\text{HASHBAR}(x) \wedge \forall y (\text{OWNER}(y) \wedge \text{OF}(y, x) \rightarrow \text{KNOW}(\text{MIA}, y))) \rangle.$$

This states that there is a hash bar of which Mia knows every owner. This is one of the readings we would like to have. So let's explore the other option. If we pull the existential quantifier from the S store first we obtain:

$$\langle \exists x (\text{HASHBAR}(x) \wedge \text{KNOW}(\text{MIA}, z_2)), \\ (\lambda u. \forall y (\text{OWNER}(y) \wedge \text{OF}(y, z_1) \rightarrow u @ y), 2) \rangle.$$

Pulling the remaining quantifier off the store then yields:

$$\langle \forall y (\text{OWNER}(y) \wedge \text{OF}(y, z_1) \rightarrow \exists x (\text{HASHBAR}(x) \wedge \text{KNOW}(\text{MIA}, y))) \rangle.$$

But this is not at all what we wanted! Cooper storage has given us not a sentence, but a formula containing the free variable z_1 . What is going wrong?

Essentially, the Cooper storage mechanism is ignoring the hierarchical structure of the NPs. The sub-NP a hash bar contributes the free variable z_1 . However, this free variable does not stay in the core representation: when the NP every owner of a hash bar is processed, the variable z_1 is moved out of the core representation and put on ice. Hence lambda abstracting the core representation with respect to z_1 isn't guaranteed to take into account the contribution that z_1 makes—for z_1 makes its contribution indirectly, via the stored universal quantifier. Everything is fine if we retrieve this quantifier first (since this has the effect of 'restoring' z_1 to the core representation) but if we use the other retrieval option it all goes horribly askew. Cooper storage doesn't

impose enough discipline on storage and retrieval, thus when it has to deal with nested NPs, it over-generates.

What are we to do? An easy solution would be to build a ‘free variable check’ into the retrieval process. That is, we might insist that we can only retrieve an indexed binding operator if the variable matching the index occurs in the core representation.

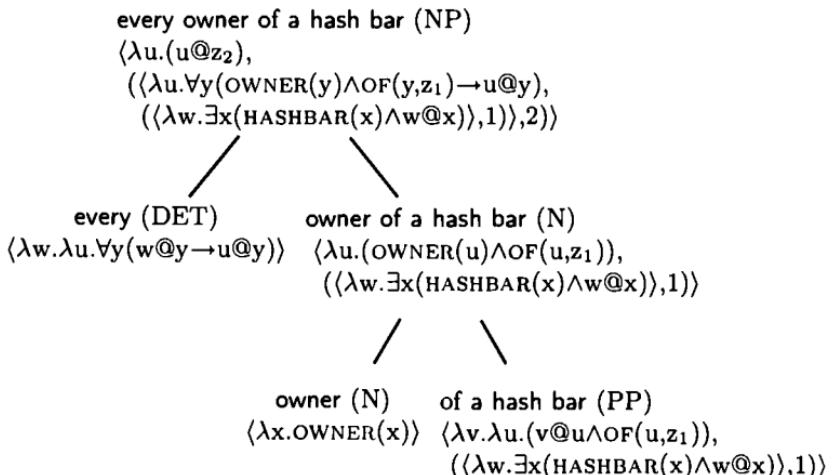
But this isn’t very principled; it deals with the symptoms, not the cause. The heart of the problem is that Cooper storage rides roughshod over the hierarchical structure of NPs; we should try to deal with this head on. (Incidentally, arguably there’s also an empirical problem: the free variable solution isn’t adequate if one extends the grammar somewhat. We won’t discuss this here but refer the reader to the Notes.)

Here’s an elegant solution due to Bill Keller: allow nested stores. That is, allow stores to contain other stores. Intuitively, the nesting structure of the stores should automatically track the nesting structure of NPs in the appropriate way. Here’s the new storage rule:

Storage (Keller)

If the (nested) store $\langle \phi, \sigma \rangle$ is an interpretation for an NP, then the (nested) store $\langle \lambda u. (u@z_i), (\langle \phi, \sigma \rangle, i) \rangle$, for some unique index i , is also an interpretation for this NP.

To see how this works, consider how we assemble the representation associated with the complex noun phrase **every owner of a hash bar**:



As for the retrieval rule, it will now look like this:

Retrieval (Keller)

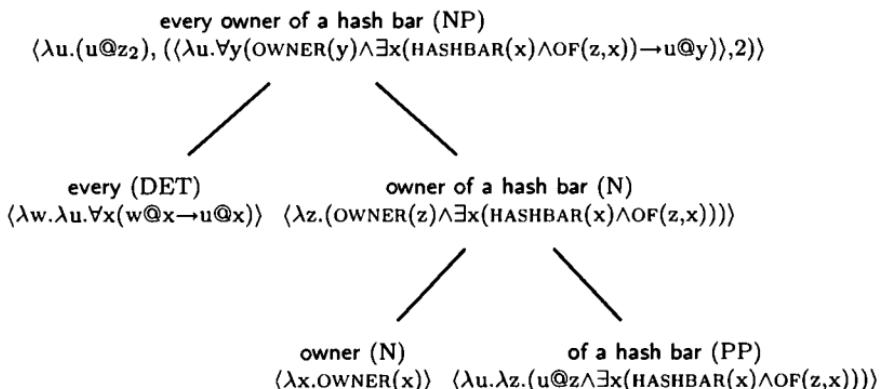
Let σ , σ_1 and σ_2 be (possibly empty) sequences of binding operators. If the (nested) store $\langle \phi, \sigma_1, (\langle \beta, \sigma \rangle, i), \sigma_2 \rangle$ is an interpretation for an expression of category S, then $\langle \beta @ \lambda z_i. \phi, \sigma_1, \sigma, \sigma_2 \rangle$ is too.

The new retrieval rule ensures that any operators stored while processing β become accessible for retrieval only after β itself has been retrieved. Nesting automatically overcomes the problem of generating readings with free variables. To see how it works in practice, let's return to our original example. The nested store associated with **Mia knows every owner of a hash bar** is

$$\langle \text{KNOW}(\text{MIA}, z_2), \\ (\langle \lambda u. \forall y (\text{OWNER}(y) \wedge \text{OF}(y, z_1) \rightarrow u @ y), \\ (\langle \lambda w. \exists x (\text{HASHBAR}(x) \wedge w @ x) \rangle, 1) \rangle, 2) \rangle.$$

There is only one way to perform retrieval: first pulling of the universal quantifier, followed by the existential quantifier, resulting in the correct reading. Since this is the only possibility, the unwanted reading as produced by Cooper storage is not generated.

But wait a minute: how do we get the reading where **Mia knows all owners of possible different hash bars**? In fact, this couldn't be easier. All we have to do is avoid storing the sub-NP **a hash bar**. If we do this, we can produce the following tree:



This leads to the following analysis for **Mia knows every owner of a hash bar**:

$$\langle \text{KNOW}(\text{MIA}, z_2), (\langle \lambda u. \forall y (\text{OWNER}(y) \wedge \exists x (\text{HASHBAR}(x) \wedge \text{OF}(y, x)) \rightarrow u @ y) \rangle, 2) \rangle.$$

There is only one operator in the store. Retrieving it yields the reading we want:

$$\langle \forall y (\text{OWNER}(y) \wedge \exists x (\text{HASHBAR}(x) \wedge \text{OF}(y, x)) \rightarrow \text{KNOW}(\text{MIA}, y)) \rangle.$$

Implementing Keller Storage

Now to make this computational: how do we implement Keller storage in Prolog? In fact it's a very simple modification of our earlier code for Cooper storage. First, we tweak the underlying representation a little. Keller-style indexed binding operators will be represented as terms

of the form `bo(Quant,Index)`. As in our implementation of Cooper storage, `Index` will be a variable; `Quant`, however, will now be a store. Stores will be represented as lists: the head will be a lambda expression, the tail (if non-empty) a list of Keller-style binding operators. So the following Prolog list represents a (Keller-style) store:

```
[walk(X),bo([lam(P,all(Y,imp(boxer(Y),app(P,X))))],X)].
```

And now there is almost nothing to do. First, we *don't* need to alter the semantic macros at all; we can continue to use the file `semLexStorage.pl` unchanged. Second, only two of the semantic rules need to be changed, namely those that deal with noun phrases:

```
combine(np:[app(A,B)|S],[det:[A],n:[B|S]]).  
combine(np:[lam(P,app(P,X)),bo([app(A,B)|S],X)], [det:[A],n:[B|S]]).
```

So we'll simply create a new file called `semRulesKeller.pl` which is identical with `semRulesCooper.pl` except that it contains these new rules. Third, we need a new definition of `sRetrieval/2`:

```
sRetrieval([S],S).
```

```
sRetrieval([Sem|Store],S):-  
    selectFromList(bo([Q|NestedStore],X),Store,TempStore),  
    appendLists(NestedStore,TempStore,NewStore),  
    sRetrieval([app(Q,lam(X,Sem))|NewStore],S).
```

(The main difference with the old version is the call to `appendLists/3` to merge stores.) The final change we shall make is to create a new wrapper called `kellerStorage.pl`. This is just our old `cooperStorage.pl` but with the new version of `sRetrieval/2`. The diagram below summarises our Keller storage setup.

And that's it. The over-generation problem is solved. We'll never see unwanted free variables again. Let's try out our program and see what happens:

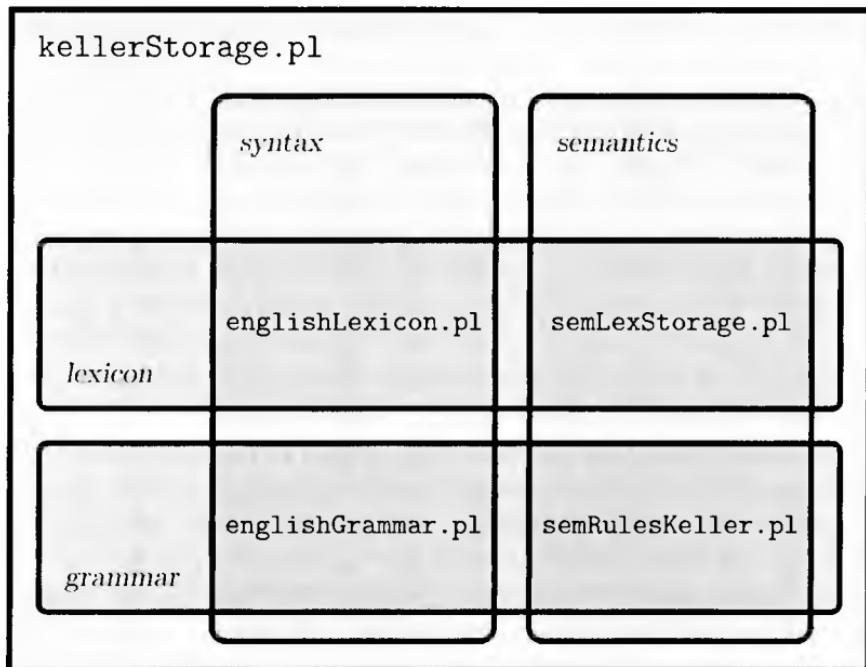
```
?- kellerStorage.
```

```
> Mia knows every owner of a hash bar.
```

Readings:

```
1 all(A,imp(and(some(B, and(hashbar(B), of(A,B))), owner(A)), know(mia,A)))  
2 some(A, and(hashbar(A), all(B, imp(and(of(B,A), owner(B)), know(mia,B)))))
```

Let's sum up what we have learned. The original version of Cooper storage doesn't handle storage and retrieval in a sufficiently disciplined way, and this causes it to generate spurious readings when faced with nested NPs. The problem can be elegantly cured by making use of nested stores, and implementing this idea requires only minor changes to our earlier Cooper storage code.



Exercise 3.3.6 In Exercise 3.3.4 we saw that Cooper storage works better with negated sentences if storage is made optional. Explain why optionality is even more important for Keller storage than it is for Cooper storage.

Exercise 3.3.7 Give first-order representations for the five readings of the sentence **a man likes every woman with a five-dollar shake**. Note that there are fewer correct readings than the combinatorial possibilities of the quantifiers involved suggests: naively, we would expect to have $3! = 6$ readings for this example. Why is one reading excluded?

3.4 Hole Semantics

Although storage methods are useful, they have their limitations. For a start, they are not as expressive as we might wish: although Keller storage predicts the five readings for

One criminal knows every owner of a hash bar,

we might want to insist that **every owner** should out-scope **a hash bar**, while at the same time leaving the scope relation between subject and object noun phrase unspecified. To put it another way, storage is essentially a technique which enables us to represent all possible meanings compactly; it doesn't allow us to express additional *constraints* on possible readings.

Moreover, storage is a technique specifically designed to handle *quantifier* scope ambiguities. Unfortunately, many other constructs (for example, negation) also give rise to scope ambiguities, and storage has nothing to say about these. In Exercise 3.3.4 we saw that Cooper storage couldn't generate all the readings of *Every boxer doesn't love a woman*, and Keller storage can't do so either (as the reader should verify). What are we to do? Inventing a special scoping mechanism for negation, and then trying to combine it with storage, doesn't seem an attractive option. We would like a uniform approach to scope ambiguity, not a separate mechanism for each construct—and this is another motive for turning to the more abstract view offered by current work on *semantic underspecification*.

In recent years there has been a great deal of interest in the use of *underspecified representations* to cope with scope ambiguities; so much so that it often seems as if semantics has entered an age of underspecification. With the benefit of hindsight, however, we can see that the idea of underspecified representations isn't really new. In our work on storage, for example, we associated stores, not simply lambda expressions, with parse tree nodes; and a store is in essence an underspecified representation. What is new is both the sophistication of the new generation of underspecified representations (as we shall see, they offer us a great deal of flexibility and expressive power), and, more importantly, the way such representations are now regarded by semanticists.

In the past, storage-style representations seem to have been regarded with some unease. They were (it was conceded) a useful tool, but they appeared to live in a conceptual no-man's-land—not really semantic representations, but not syntax either—that was hard to classify. The key insight that underlies current approaches to underspecification is that it is both *fruitful* and *principled* to view representations more abstractly. That is, it is becoming increasingly clear that the level of representations is richly structured, that computing appropriate semantic representations demands deeper insight into this structure, and that—far from being a sign of some fall from semantic grace—semanticists should learn to play with representations in more refined ways.

We shall now introduce an approach to underspecification called *hole semantics*. We have chosen hole semantics because it is the underspecification method we are most familiar with (the method is due to Johan Bos) and it illustrates most of the key ideas of current approaches to underspecification in a fairly simple way. References to other approaches to underspecification can be found in the Notes at the end of the chapter.

First Steps in Hole Semantics

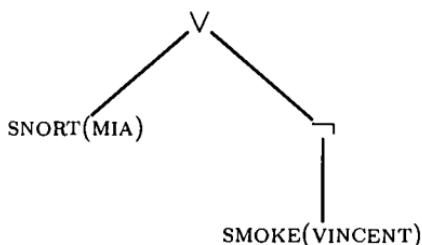
Viewed from a distance, hole semantics shares an obvious similarity with storage methods: at the end of the parsing stage, sentences *won't* be associated with a first-order formula. Rather, they will be associated with an abstract representation, called an *underspecified semantic representation* (or USR) from which the desired first-order representation can be read off. Viewed closer up, however, it becomes clear that hole semantics adopts a more radical perspective on representation than storage does.

Hole semantics is a constraint-based approach to semantic representation. That is, a hole semantics USR is essentially a set of *constraints*: any first-order formula which fulfils these constraints—which govern how the components making up first-order formulas (that is, quantifiers, boolean connectives, relation symbols, variables, and constants) can be plugged together—is a permissible semantic representation for the sentence. This contrasts sharply with the essentially generative approach offered by storage (*Here's the store! Enumerate the readings!*) and is the source of much of hole semantics's power.

Now for the key question—what sort of formalism will we use for USRs? That is, what language will we use to express constraints on our semantic representations (that is, on first-order formulas)? The answer may come as a surprise: we are going to use (sorted) first-order logic! This is worth repeating: yes, we are going to use a (sorted) first-order language called the *underspecified representation language* (URL) to impose constraints on the structure of the first-order formulas used for semantic representations. Let's think this through a bit.

In this book, our semantic representation languages (SRLs) are always first-order languages (with equality) over some vocabulary. Now, the basic idea of hole semantics is to impose constraints on SRL formulas. We are going to design a (sorted) first-order language (the URL) for talking about the structure of SRL formulas. The vocabulary of the URL will contain special symbols for talking about the quantifiers, connectives, relation symbols, constants and variables of the SRL, and a special symbol \leq for saying how different subparts of an SRL formula are to be related. That is, the URL is designed for talking about the way the subformulas that make up an SRL formula are embedded inside one another; it enables us to describe *SRL formula trees*.

What's a formula tree? Simply the natural tree structure that every first-order formula has. For example, $\text{SNORT}(\text{MIA}) \vee \neg\text{SMOKE}(\text{VINCENT})$ can be thought of as the following tree:



As will become clear, we think of SRL formulas as containing *holes* (that is, certain nodes in the formula trees) that need to be *filled* with certain kinds of subformulas. To put it another way, we will typically want to be able to insist that some hole must *dominate* (that is, be higher in the formula tree) than certain other nodes. For example, we might want to insist that a hole dominates an existentially quantified subformula. Such *dominance constraints* lie at the heart of hole semantics: they tell us how the subformulas that make up the semantic representation can be *plugged* together.

It's time to get to work. Suppose we have fixed the vocabulary of our SRL (the first-order language in which semantic representations will be written). Then the vocabulary of the *underspecified representation language* (URL) for this SRL consists of the following items:

1. The 2-place predicates :NOT and \leq .
2. The 3-place predicates :IMP, :AND, :OR, :ALL, :SOME and :EQ.
3. Moreover, every constant in the SRL vocabulary is also a URL constant. (For example, if MIA is an SRL constant, then MIA is also a URL constant.)
4. Finally, if the SRL vocabulary contains a relation symbol PRED of arity n , then :PRED is a $n+1$ -place URL symbol. (For example, if LOVE is a two place relation symbol in the SRL, then :LOVE will be a three place relation in the URL.)

Why define the URL vocabulary this way? Because it gives us all we need to talk about the subformula structure of SRL formulas. In particular, the symbols :NOT, :IMP, :AND, :OR, :ALL, :SOME and :EQ enable us to talk about occurrences of the symbols \neg , \rightarrow , \wedge , \vee , \forall , \exists , and $=$ in SRL (sub)formulas. Furthermore, note that we have taken care to systematically mirror the SRL vocabulary in the URL vocabulary: each SRL constant is also used as a URL constant, and each SRL n -place relation symbol is used (with a : symbol in front of it) as an $n+1$ -place URL relation symbol. Don't worry about why we want the extra argument place in the URL relation symbols—that will soon become apparent. The important point for now is simply that this notation

makes it possible to talk about all items in the SRL vocabulary. Last (but certainly not least) we have the \leq symbol. We shall use this to state the dominance constraints.

Now, we could define our URL simply to be the ordinary first-order language built over the vocabulary just defined. But it will make things a little more readable if we define our URL to be a *sorted* first-order language (the reader may wish to consult our discussion of sorted first-order logic in Chapter 1).

Let's be more precise. Our URL will have three sorts, namely *holes* (hole variables will be written h , h' , h_1 , h_2 , and so on), *labels* (label variables will be written l , l' , l_1 , l_2 , and so on) and *meta-variables* (these will be written v , v' , v_1 , v_2 , and so on). We shall also use the following terminology. First, by a *node* we mean a hole or a label. Second, we define a *meta-term* of the URL to be either a meta-variable or a URL constant (that is, a constant from the SRL vocabulary). We call such terms meta-terms since meta-variables will be used to talk about SRL variables, and URL constants will denote SRL constants in the obvious way (for example, we will talk about the SRL constant **MIA** using the URL constant **MIA**).

Actually, in this chapter we won't need to use all the sorted first-order formulas that can be constructed over this vocabulary. We'll mainly be interested in the fragment of this language consisting of all the *existentially closed conjunctive formulas*, for such formulas will be our underspecified representations (USRs). So, what exactly are the USRs we shall use? First, a *basic USR* is defined as follows:

1. If l is a label, and h is a hole, then $l \leq h$ is a basic USR;
2. If l is a label, and n and n' are nodes, then $l:\text{NOT}(n)$, $l:\text{IMP}(n,n')$, $l:\text{AND}(n,n')$, and $l:\text{OR}(n,n')$ are basic USRs;
3. If l is a label, and t and t' are meta-terms, then $l:\text{EQ}(t,t')$ is a basic USR;
4. If l is a label, and S is a symbol in the SRL language with arity n , and $t_1 \dots t_n$ are meta-terms, then $l:S(t_1, \dots, t_n)$ is a basic USR.
5. If l is a label, and v is a meta-variable, and n a hole or a label, then $l:\text{SOME}(v,n)$ and $l:\text{ALL}(v,n)$ are basic USRs.
6. Nothing else is a basic USR.

Note that clauses 2–5 have the same form: a label variable in front of something that talks about an SRL symbol. Such formulas assert that the node in the SRL formula tree denoted by the label variable is decorated by the symbol in question. (Incidentally, this is why n -ary SRL relation symbols become $n+1$ -ary URL symbols—the extra argument slot occurs before the $:$ symbol and is filled by the label

variable.) In short, these basic formulas allow us to talk about the formulas that nodes of SRL formula trees are decorated with. We call the basic USRs defined by clauses 2–5 *labelled formulas*.

What about item 1? This makes use of the crucial \leq symbol. A USR of the form $l \leq h$ says that a certain position h in the SRL formula tree needs to be higher up than the syntactic position labelled l . That is: the hole h needs to dominate the node labelled l . As we shall shortly see, such requirements are fundamental to the way hole semantics is used. We call basic USRs of this form *dominance constraints*.

Now that we know what the basic USRs at our disposal are, we can define the remaining USRs as follows:

1. All basic USRs are USRs;
2. If ϕ is a USR, and n is a node then $\exists n\phi$ is a USR;
3. If ϕ is a USR, and v is a meta-variable then $\exists v\phi$ is a USR;
4. If ϕ and ψ are USRs, then $(\phi \wedge \psi)$ is a USR;
5. Nothing else is a USR.

In short, as promised above, the only USRs we shall use in this chapter are the existentially closed conjunctive formulas of our three sorted URL language.

An example

We've introduced a lot of machinery—it's high time we saw how to put it to work. Here's a familiar example: how does hole semantics handle **Every boxer loves a woman?** This is the USR for this sentence:

$$\begin{aligned} & \exists l_1 \exists l_2 \exists v_1 (l_1:ALL(v_1, l_2) \wedge \exists l_3 \exists h_1 (l_2:IMP(l_3, h_1) \wedge l_3:BOXER(v_1) \\ & \wedge \exists l_4 \exists l_5 \exists v_2 (l_4:SOME(v_2, l_5) \wedge \exists l_6 \exists h_2 (l_5:AND(l_6, h_2) \wedge l_6:WOMAN(v_2) \\ & \wedge \exists l_7 (l_7:LOVE(v_1, v_2) \wedge l_7 \leq h_1 \wedge l_7 \leq h_2 \wedge \exists h_0 (l_1 \leq h_0 \wedge l_4 \leq h_0)))))). \end{aligned}$$

This is not very readable, but we can improve matters by moving the quantifiers to the front:

$$\begin{aligned} & \exists h_0 \exists h_1 \exists h_2 \exists l_1 \exists l_2 \exists l_3 \exists l_4 \exists l_5 \exists l_6 \exists l_7 \exists v_1 \exists v_2 (l_1:ALL(v_1, l_2) \wedge l_2:IMP(l_3, h_1) \\ & \wedge l_3:BOXER(v_1) \wedge l_4:SOME(v_2, l_5) \wedge l_5:AND(l_6, h_2) \wedge l_6:WOMAN(v_2) \\ & \wedge l_7:LOVE(v_1, v_2) \wedge l_7 \leq h_1 \wedge l_7 \leq h_2 \wedge l_1 \leq h_0 \wedge l_4 \leq h_0). \end{aligned}$$

This is a lot nicer—but what does it mean? It tells us how we are permitted to plug together the various components to build the required first-order semantic representation. Before going further, one convention needs to be mentioned: the hole h_0 is the variable that we reserve for the SRL formula that we will eventually build (we call this hole the top hole). Thus the outermost quantifier says that there exist certain SRL formula(s), namely the one(s) that represent the meaning of **Every boxer loves a woman**.

But what should these formula(s) look like? The basic USRs that make up this formula will tell us. From the conjunct

$l_1:\text{ALL}(v_1, l_2)$

we learn that there must be a universally quantified subformula in the SRL formula we are trying to build. Moreover, this conjunct tells us that the matrix of this universal formula is the formula labelled l_2 . And what is l_2 ? Well, from $l_2:\text{IMP}(l_3, h_1)$ we learn that it is an implication. Moreover, from $l_3:\text{BOXER}(v_1)$ we learn that the antecedent of this implication is an atomic symbol made up from **BOXER** and some variable. Note, however, that we *don't* have precise information about the consequent formula: h_1 is a hole variable. This hole will eventually need to be plugged by some subformula, but we don't yet know how to do this.

What else are we told? Well, from

$l_4:\text{SOME}(v_2, l_5)$

we learn that there is also an existentially quantified subformula in the semantic representation. From $l_5:\text{AND}(l_6, h_2)$ we learn that this existentially quantified subformula is a conjunction, and from $l_6:\text{WOMAN}(v_2)$ we learn that the first conjunct is an atomic symbol made up from **WOMAN** and some variable. Note that we don't have precise information about the second conjunct: once again, a hole variable, namely h_2 , has been used to mark this spot. Finally, from

$l_7:\text{LOVE}(v_1, v_2)$

we learn that the semantic representation contains the two place relation symbol **LOVE**, and that both its arguments are variables.

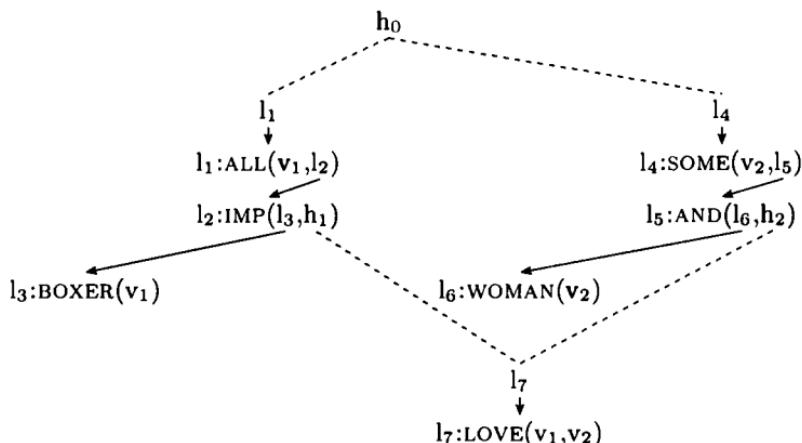
And now for the crucial step—what do the dominance constraints tell us? That is, what do we learn from the conjuncts

$$l_7 \leq h_1 \wedge l_7 \leq h_2 \wedge l_1 \leq h_0 \wedge l_4 \leq h_0?$$

First, recall that l_7 names the node in the SRL formula tree where the two place relation symbol **LOVE** occurs. Further, recall that h_1 and h_2 are the holes in the universal and existential SRL subformulas respectively. Hence the conjuncts $l_7 \leq h_1$ and $l_7 \leq h_2$ tell us that the **LOVE** node is dominated by both holes (so both quantifiers have scope over this subformula). Next, recall that l_1 names the universally quantified subformula and that l_2 names the existentially quantified subformula. Hence the conjuncts $l_1 \leq h_0$ and $\wedge l_4 \leq h_0$ tell us that both quantified subformulas occur in the semantic representation (as h_0 is the name of the representation we are trying to build).

But to really understand what is going on, we should put all this information into visual form. Indeed, we strongly encourage the reader to

view hole semantics USRs simply as a linear notation for the following type of picture:

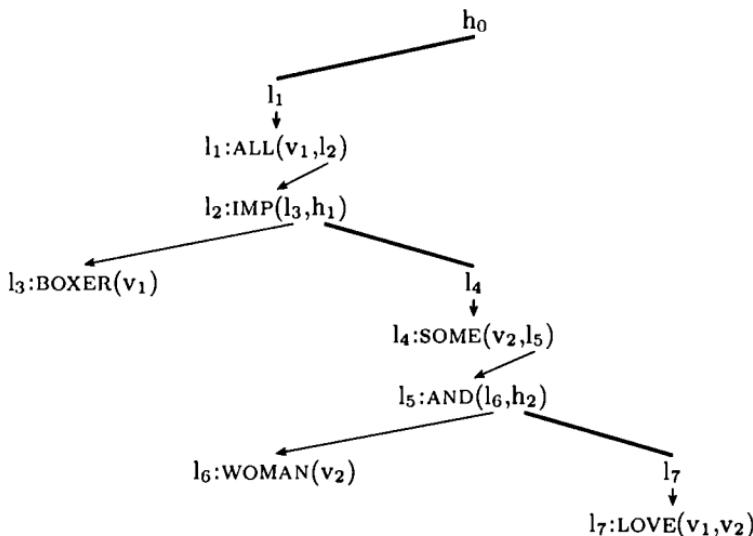


Now the constraints imposed on possible SRL formulas should be a lot more comprehensible. As we can see, the formula $l_7:\text{LOVE}(v_1,v_2)$ is forced to be out-scoped by the consequent of the universal quantifier's scope, and the second conjunct of the existential quantifier's scope. Moreover—crucially—we see that neither l_1 (the universal subformula) nor l_4 (the existential subformula) has been forced to be out-scoped by the other. Intuitively, this is what we want: we have a USR that permits either quantifier scope.

Plugging

So far, so good. But we are not yet finished. We now have a constraint on possible readings—but how do we get our hands on the actual semantic representations? By *plugging*. Holes need to be filled. So each hole should be plugged with a formula in such a way that all the constraints are satisfied. In other words, we should be sure that each hole gets associated with some label. Now, no label can be plugged into two different holes at the same time. Therefore, a plugging is a one-to-one mapping from holes to labels (that is, an injective function, with the set of holes as domain and the set of labels as codomain). A plugging for a proper USR is admissible if the instantiations of the holes with labels result in a representation in which there is no contradiction with anything demanded by the constraints.

In the above example there are two admissible pluggings, namely P_1 such that $P_1(h_0)=l_1$, $P_1(h_1)=l_4$, $P_1(h_2)=l_7$, and P_2 such that $P_2(h_0)=l_4$, $P_2(h_1)=l_7$, $P_2(h_2)=l_1$. Let's look at each in turn. If we modify the previous diagram in accordance with P_1 we obtain:



Make sure you understand the relationship between this picture and the one given earlier of the USR. The main point to observe is that the new picture shows what you get when you insert the righthand side of the original USR picture (that is, the part below the l_4 node) underneath h_1 , in accordance with the plugging instruction $P_1(h_1)=l_4$. Observe, in addition, that in the new picture the top hole h_0 has been identified with l_1 , and h_2 has been identified with l_7 .

The other thing to observe about the new picture is that (in contrast to the original USR picture) it is a *tree*. In fact, it's pretty much an SRL formula tree. To see this, note that if we linearise this tree, substituting formulas for labels in the way indicated by the arrows, we obtain:

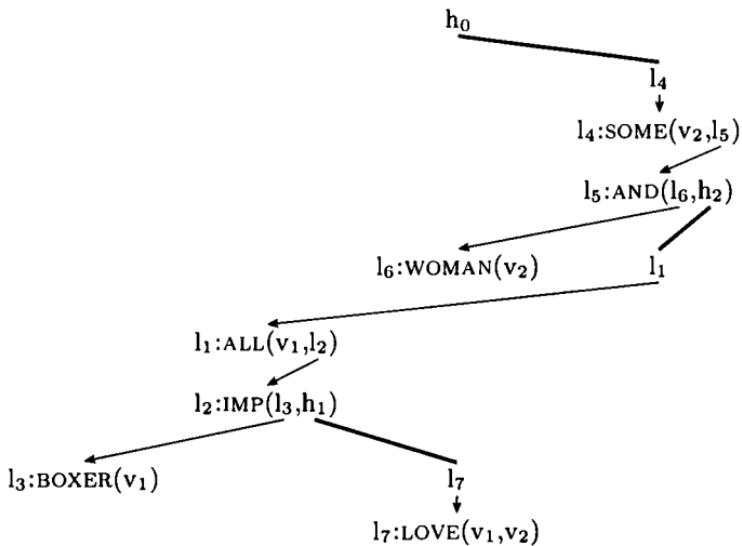
$$\text{ALL}(v_1, \text{IMP}(\text{BOXER}(v_1, \text{SOME}(v_2, \text{AND}(\text{WOMAN}(v_2, \text{LOVE}(v_1, v_2))))))).$$

This is, it should be pretty clear that we have constructed the SRL formula

$$\forall x(\text{BOXER}(x) \rightarrow \exists y(\text{WOMAN}(y) \wedge \text{LOVE}(x,y))).$$

Summing up: plugging P_1 does indeed give us one of the required readings, namely the reading where the universal quantifier takes wide scope.

Unsurprisingly, our other plugging P_2 gives us the other reading. If we modify the diagram of the USR in accordance with P_2 , we obtain the following tree:



This time, the new picture shows what you get when you insert the lefthand side of the original USR picture (that is, the part below the l_1 node) underneath h_2 , in accordance with the plugging instruction $P_2(h_2)=l_1$. In addition, the top hole h_0 has been identified with l_4 , and h_1 has been identified with l_7 .

Once again, the result is an SRL formula tree. If we linearise this by following the arrows we obtain:

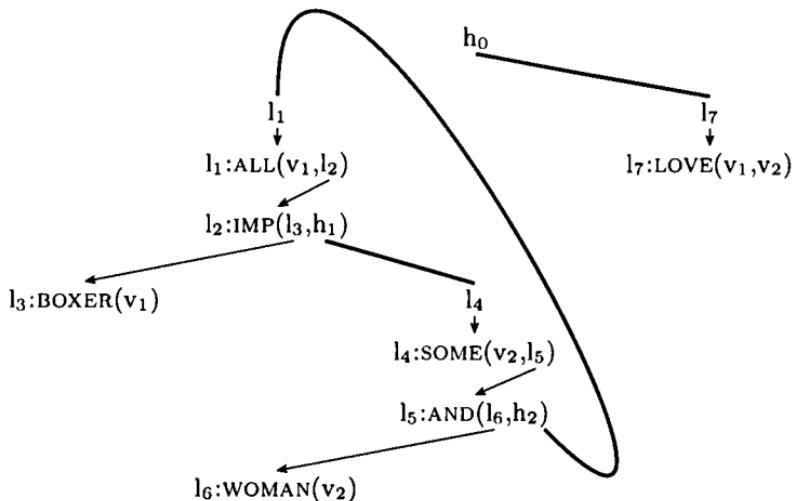
$$\text{SOME}(v_2, \text{AND}(\text{WOMAN}(v_2, \text{ALL}(v_1, \text{IMP}(\text{BOXER}(v_1, \text{LOVE}(v_1, v_2))))))).$$

Again, it is pretty clear that we have a first-order representation, namely:

$$\exists y(\text{WOMAN}(y) \wedge \forall x(\text{BOXER}(x) \rightarrow \text{LOVE}(x,y))).$$

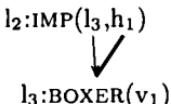
Here the existential quantifier out-scopes the universal quantifier. Our second plugging has given us the second reading we wanted.

Before we go on, recall that we said there were only two admissible pluggings for the USR for Every boxer loves a woman. Why is that? Well, let's consider another plugging P_3 , such that $P_3(h_0)=l_7$, $P_3(h_1)=l_4$, $P_3(h_2)=l_1$. What's wrong with this plugging? Let's examine its diagram and see:



It should be clear that this plugging doesn't describe a formula tree (it contains a cycle and it is not connected). Hence it is not an admissible plugging.

There is another situation that we want to avoid. Consider the (partial) plugging $P'(h_1)=l_3$. This will give us the following structure:



Once more, this doesn't describe a formula tree, so it's not an admissible plugging. More generally, we don't want to generate any plugging where two nodes with the same parent dominate a common node.

Computing with Hole Semantics

The basic ideas underlying hole semantics should now be clear—but is hole semantics computationally useful? And can it be integrated into our grammar architecture?

Indeed it can. In fact, it is extremely straightforward to incorporate it into our architecture—doing so is almost like stepping back to the pre-store lambda-driven work of the previous chapter. Think about it. Hole semantics USRs are simply formulas of a (sorted) first-order language. But in the previous chapter we studied in detail how to use lambda calculus to glue together first-order representations. Very well then—let's simply use the lambda calculus to glue together USRs!

Needless to say, this approach fits beautifully with our grammar architecture. To build USRs we'll merely need to define new semantic macros (based on USRs) and new semantic rules, and our existing

programs will take care of the rest. Of course, we will also need to perform plugging at the end of the process, and this requires extra code. Nonetheless, the required code will fit into our architecture in much the same way as the extra retrieval code did when we discussed storage.

In what follows we shall take the reader through the two main steps involved (that is, building the USRs, and plugging). But before doing this, a quick word about the Prolog notation we shall use. Here is a USR for a boxer collapses:

$$\exists h_0 \exists l_1 \exists h_1 \exists l_2 \exists l_3 \exists l_4 \exists v_1 (l_3 : \text{SOME}(v_1, l_4) \wedge l_4 : \text{AND}(l_2, h_1) \wedge l_1 \leq h_1 \\ \wedge l_3 \leq h_0 \wedge l_2 : \text{BOXER}(v_1) \wedge l_2 \leq h_0 \wedge l_1 : \text{COLLAPSE}(v_1) \wedge l_1 \leq h_0).$$

And here is its Prolog representation:

```
some(A, and(hole(A), some(B, and(label(B), some(C, some(D,
    some(E, some(F, some(G, and(hole(C), and(label(D), and(label(E),
        and(label(F), and(some(E, G, F), and(and(F, D, C), and(less_eq(B, C),
            and(less_eq(E, A), and(and(pred1(D, boxer, G), less_eq(D, A)),
                and(pred1(B, collapse, G), less_eq(B, A)))))))))))))))))).
```

This is just our usual Prolog notation for first-order logic. Note that we have used `leq` for \leq . To distinguish holes and labels from each other (and from meta-variables), we have used `hole(H)` to designate a hole h and `label(L)` for a label l . Furthermore, note that we use expressions of the form

`pred1(L, boxer, X)`

to represent $l : \text{BOXER}(x)$. Nothing deep lies behind this choice—it's just a simple way of representing labelled formulas in Prolog. Similarly, we would use

`pred2(L, love, X, Y)`

to represent $l : \text{LOVE}(x, y)$.

Building USRs with Lambdas

As promised, we shall build USRs using the lambda-driven approach to semantic construction that we are familiar with. That is, each lexical item will be associated with a lambda expression (though this time it will be an expression that glues together USRs) and as we work our way up the natural language syntax tree we will use functional application and β -conversion.

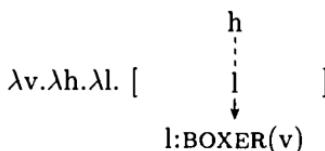
Let's look at some sample lexical entries. Here's the lambda expression associated with the noun `boxer`:

$\lambda v. \lambda h. \lambda l. (l : \text{BOXER}(v) \wedge l \leq h)$.

Here the lambda-abstraction v plays the role we are familiar with from our work in the previous chapter: that is, it marks a slot that needs

to be filled with missing information. What about the h and the l ? Basically, we view each lexical item as contributing a hole and a label to the sentence. The hole h can be viewed as the abstract (underspecified) scope contributed by that lexical item. Note that we constrain the label contributed by the item (that is, the place in the SRL formula where the item actually appears) to be dominated by h . Together h and l define the *scope domain* contributed by the noun.

Put like this, it may seem rather abstract. But if you draw the previous item as a little chunk of a tree, you can see that it is actually quite concrete:

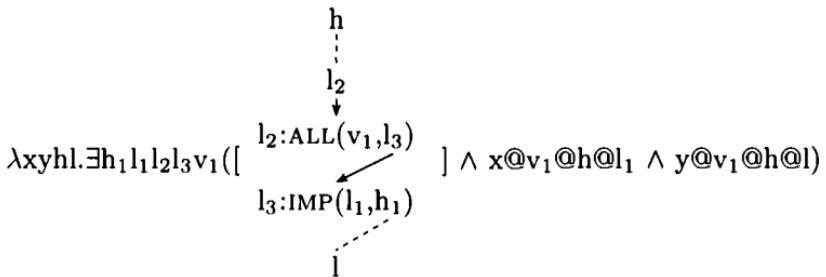


As the picture makes clear, the l marks the place in the semantic representation where the item (here **BOXER**) will appear, and h marks the scope of this item. As the dotted lines show, this scope is at present underspecified. All we know for the present is that the hole must dominate the label.

Let's consider a more complex example. Here is the lambda expression associated with the determiner **every** (we have used the following notational convention: indexes have been attached to existentially quantified meta-variables, and lambda-bound variables have been left index-free):

$$\begin{aligned} & \lambda x. \lambda y. \lambda h. \lambda l. \exists h_1 \exists l_1 \exists l_2 \exists l_3 \exists v_1 (l_2: \text{ALL}(v_1, l_3) \wedge l_3: \text{IMP}(l_1, h_1) \wedge l \leq h_1 \\ & \wedge l_2 \leq h \wedge x @ v_1 @ h @ l_1 \wedge y @ v_1 @ h @ l). \end{aligned}$$

As in the previous example, the lambda-abstraction variables x and y play their familiar role, and the lambdas binding h and l define the scope domain contributed by **every**. The quantifier itself gets label l_2 with restriction l_1 and nuclear scope h_1 . The constraint $l_2 \leq h$ ensures that the quantifier is out-scoped by the top of the scope domain, and $l \leq h_1$ states that its verbal argument (y) is in its scope. The applications ($x @ v_1 @ h @ l_1$) and ($y @ v_1 @ h @ l$) are crucial. The first application associates the top hole (h) and restriction label of the quantifier (l_1) with the scope domain of the noun representation. The second applications associates the top hole (h) and label (l) of the quantifier with the scope domain of the verb phrase argument. Let's put this all in pictorial form:



Now, this is rather more complicated than our previous example, so let's consider what happens when we apply the semantic representation of every to that of boxer. That is (reverting again to linear notation), let's consider the following functional application:

$$\lambda x.\lambda y.\lambda h.\lambda l.\exists h_1\exists l_1\exists l_2\exists l_3\exists v_1(l_2:\text{ALL}(v_1,l_3) \wedge l_3:\text{IMP}(l_1,h_1) \wedge l_1\leq h_1 \wedge l_2\leq h \wedge x@v_1@h@l_1 \wedge y@v_1@h@l) @ \lambda v.\lambda h.\lambda l.(l:\text{BOXER}(v) \wedge l\leq h).$$

We can β -convert this using the outermost lambda:

$$\lambda y.\lambda h.\lambda l.\exists h_1\exists l_1\exists l_2\exists l_3\exists v_1(l_2:\text{ALL}(v_1,l_3) \wedge l_3:\text{IMP}(l_1,h_1) \wedge l_1\leq h_1 \wedge l_2\leq h \wedge \lambda v.\lambda h.\lambda l.(l:\text{BOXER}(v) \wedge l\leq h)@v_1@h@l_1 \wedge y@v_1@h@l).$$

Again we β -convert, thus moving the variable v_1 into the argument position of BOXER:

$$\lambda y.\lambda h.\lambda l.\exists h_1\exists l_1\exists l_2\exists l_3\exists v_1(l_2:\text{ALL}(v_1,l_3) \wedge l_3:\text{IMP}(l_1,h_1) \wedge l_1\leq h_1 \wedge l_2\leq h \wedge \lambda h.\lambda l.(l:\text{BOXER}(v_1) \wedge l\leq h)@h@l_1 \wedge y@v_1@h@l).$$

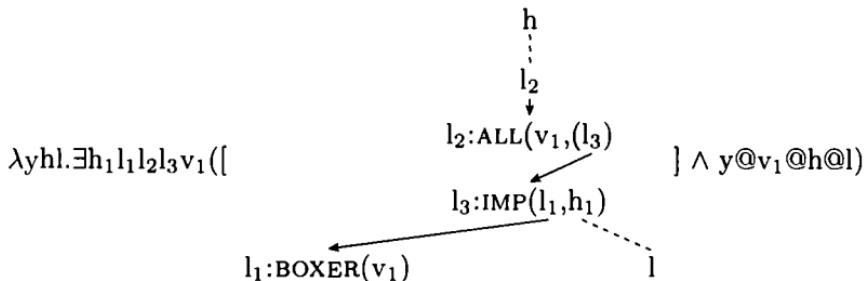
Again we β -convert, thereby associating the determiner's scope domain hole h with that of the noun:

$$\lambda y.\lambda h.\lambda l.\exists h_1\exists l_1\exists l_2\exists l_3\exists v_1(l_2:\text{ALL}(v_1,l_3) \wedge l_3:\text{IMP}(l_1,h_1) \wedge l_1\leq h_1 \wedge l_2\leq h \wedge \lambda l.(l:\text{BOXER}(v_1) \wedge l\leq h)@l_1 \wedge y@v_1@h@l).$$

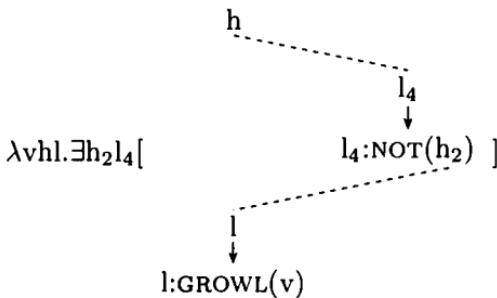
A further β -conversion yields the dominance constraint $l_1\leq h$:

$$\lambda y.\lambda h.\lambda l.\exists h_1\exists l_1\exists l_2\exists l_3\exists v_1(l_2:\text{ALL}(v_1,l_3) \wedge l_3:\text{IMP}(l_1,h_1) \wedge l_1\leq h_1 \wedge l_2\leq h \wedge l_1:\text{BOXER}(v_1) \wedge l_1\leq h \wedge y@v_1@h@l).$$

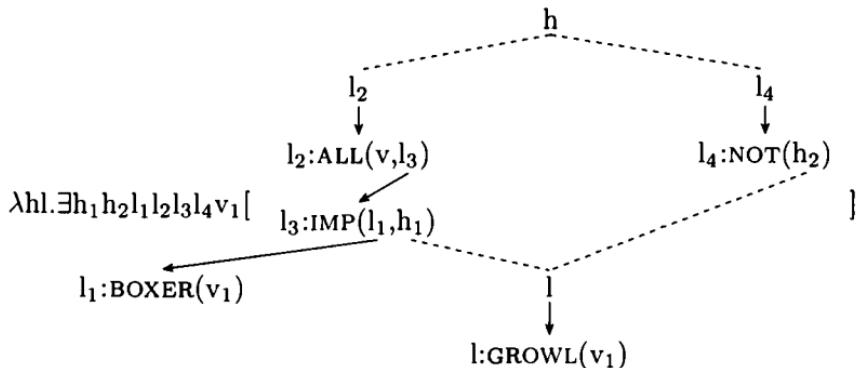
And that's all we can do for now. Switching to pictorial mode we see that we've built the following:



Having come this far, we might as well continue and build a USR for a complete sentence. Let's build one for every boxer does not growl. The USR for does not growl is (in pictorial format):



So, to build a representation for every boxer does not growl, we apply the USR for every boxer to the USR for does not growl and β -convert. This yields the following:



As a little experimentation should make clear, this USR can be plugged in two ways, and doing so yields two different readings for the sentence, namely

$$\forall x(\text{BOXER}(x) \rightarrow \neg\text{GROWL}(x)),$$

and

$\neg \forall x (\text{BOXER}(x) \rightarrow \text{GROWL}(x))$.

We leave the details to the reader.

Semantic macros and rules for hole semantics

If we are to integrate hole semantics into our grammar architecture, we don't have any choice about what to do next: we need to define a collection of (hole semantics based) semantic macros, and semantic rules for working with them.

And this is exactly what we shall do. We shall create a new file (called `semLexHole.pl`) for our hole-semantics-based semantic macros. Let's look at some of the entries this file will contain.

First, here's the semantic macro for nouns:

```
semLex(noun,M):-  
M = [symbol:Sym,  
     sem:lam(X, lam(H, lam(L, and(pred1(L,Sym,X), leq(L,H)))))].
```

This is just the USR for nouns we used in the previous example (recall our discussion of `boxer`) written in our usual Prolog notation. Similarly, here's the semantic macro for the determiner `every`. Once again, this merely puts the USR discussed earlier in prolog notation:

```
semLex(det,M):-  
M = [type:uni,  
     sem:lam(N, lam(V, lam(H, lam(L, some(H1, some(L1, some(L2, some(L3, some(X,  
         and(hole(H1), and(label(L1), and(label(L2), and(label(L3),  
             and(all(L2,X,L3), and(imp(L3,L1,H1), and(leq(L,H1),  
                 and(leq(L2,H), and(app(app(app(N,X),H),L1),  
                     app(app(app(V,X),H),L))))))))))))))).
```

So let's look at some other examples. Here are the macros for proper names, intransitive verbs, and transitive verbs:

```
semLex(pn,M):-  
M = [symbol:Sym,  
     gender:_,  
     sem:lam(V, lam(H, lam(L, app(app(app(V,Sym),H),L))))].
```

```
semLex(iv,M):-  
M = [symbol:Sym,  
     sem:lam(X, lam(H, lam(L, and(pred1(L,Sym,X),  
         leq(L,H)))))].
```

```
semLex(tv,M):-  
M = [symbol:Sym,  
     sem:lam(Z, lam(X, app(Z, lam(Y, lam(H, lam(L,  
         and(pred2(L,Sym,Y,X), leq(L,H))))))))].
```

Note that these are simply the usual macros for proper names, in-

transitive verbs, and transitive verbs—except that they are augmented with extra H and L slots for the scope domain.

As we've said several times, underspecification allows us to be very flexible. For example, we can use one and the same technique for both quantifiers and negation. Actually, we've already seen a negation example (recall that we constructed a USR for Every boxer does not growl) but we did not discuss how the USR associated with the negated verb phrase (does not growl) was built. Let's put that right. Here's the semantic macro for negative polarity auxiliary verbs:

```
semLex(av,M):-  
M = [pol:neg,  
     sem:lam(V, lam(X, lam(H, lam(L, some(S, some(N, and(hole(S),  
           and(label(N), and(not(N,S), and(1eq(N,H), and(1eq(L,S),  
             app(app(app(V,X),H),L))))))))))]);
```

These examples give a taste of the semantic macros the reader will find in `semLexHole.pl`. In addition, the reader will find there macros for copulas, relative pronouns, adjectives, prepositions, and coordination, and all the items in our lexicon.

Now for the semantic rules. Once again, we create a new file (call it `semRulesHole.pl`) which contains new definitions of `combine/2` suitable for working with USRs. Actually, not much needs to be changed (after all, we're basically doing straightforward lambda manipulation) but one point needs to be mentioned.

Consider the following rule:

```
combine(t:U,[s:S]):-  
  betaConvert(some(T, and(hole(T), some(L, and(label(L),  
    app(app(S,T),L)))),U).
```

This closes off the outermost scoping domain. Here `S`, the representation for the sentence, is of the form $\lambda h. \lambda l. \phi$. So when we use this new rule, which is of the form $\exists h_0 \exists l_1 S @ h_0 @ l_1$, we get $\exists h_0 \exists l_1 \lambda h. \lambda l. \phi @ h_0 @ l_1$. This can then be β -reduced to a USR without any occurrences of lambda-bound variables. Whenever we reach sentence level, we have to close off the scoping domain in this way. For example, you will see the same technique is used for complex sentences.

The Plugging Algorithm

We now have all the information needed to build USRs, but we have to ensure that this information is correctly integrated, and moreover we need some code to carry out the plugging process. Accordingly, we shall now create a top-level file called `holeSemantics.pl` which coordinates the lexicon, the syntax rules, the semantic macros, and the semantic rules. This file will also contain the predicates that carry out the actual

plugging. The plugging process is interesting (and fairly intricate) so we will discuss it in some detail.

At the end of the semantic construction process just discussed, a sentence is associated with a hole semantics USR, that is, an existentially closed conjunctive formula of (sorted) first-order logic. The algorithm we shall use for building semantic representations using the information in this USR has four steps:

1. Skolemise away the variables in the USR.
2. Break down the USR into its components and assert them to the Prolog database.
3. Calculate a plugging.
4. Apply the plugging to construct the SRL formula.

Let's look at each of these steps in turn.

Actually, not much needs to be said about Step 1. Here, skolemisation simply means getting rid of the block of existential quantifiers that every USR is prefixed with: we throw them away and replace the variables they bind with unique constants (we'll learn more about skolemisation in Chapter 5, but for present purposes this is all the reader needs to know). Needless to say, dropping the block of quantifiers is easy to do, and to carry out the substitution of constants for variables we simply use the built-in Prolog `numbervars/3` predicate (recall that this instantiates Prolog variables with unique non-variables; we used it in the previous chapter when we defined a predicate to test for alphabet variants).

What about Step 2? Our plan is to use the Prolog database to store information about admissible pluggings. But this means we should declare the predicates we use as dynamic (that is, predicates whose definition can change during runtime). Why is this? Because there may well be several admissible pluggings, and we want to compute them all. (Moreover, we want to be able to handle more than one example!) Accordingly, we make the following declarations:

```
:-
    dynamic plug/2, leq/2, hole/1, label/1.
:- dynamic some/3, all/3, que/4.
:- dynamic not/2, or/3, imp/3, and/3.
:- dynamic pred1/3, pred2/4, eq/3.
```

We then assert the USR to the Prolog database. Now, USRs have a very simple form (they are existentially quantified conjunctive formulas) so we need only to recurse on this structure. Here's the required code:

```
assertUSR(some(_,F)):-
```

```

assertUSR(F).

assertUSR(and(F1,F2)):-
  assertUSR(F1),
  assertUSR(F2).

assertUSR(F):-
  \+ F=and(_,_),
  \+ F=some(_,_),
  assert(F).

```

With Steps 1 and 2 out of the way the preliminaries are over. It's time for Step 3, actually calculating the plugging. First we use `parent/2` to define parenthood between labels and holes:

```

parent(A,B):- imp(A,B,_).
parent(A,B):- imp(A,_,B).
parent(A,B):- or(A,B,_).
parent(A,B):- or(A,_,B).
parent(A,B):- and(A,B,_).
parent(A,B):- and(A,_,B).
parent(A,B):- not(A,B).
parent(A,B):- all(A,_,B).
parent(A,B):- some(A,_,B).

```

Actually, we need another clause here. We will represent the fact that hole A is to be plugged by label B as `plug(A,B)`. This also gives rise to a parenthood relation, so we need a clause to cover this case:

```
parent(A,B):- plug(A,B).
```

Parenthood is the one-step relation of dominance between nodes. But of course, the true dominance relation we are interested is the multiple-step (that is, transitively closed) relation of dominance. Using the `parent` relation just defined, together with the \leq information in the USR, we define dominance as follows:

```

dom(X,Y):- dom([],X,Y).

dom(L,X,Y):-
  parent(X,Y),
  \+ memberList(parent(X,Y),L).

dom(L,X,Y):-
  leq(Y,X),
  \+ memberList(leq(Y,X),L).

dom(L,X,Z):-

```

```

parent(X,Y),
\+ memberList(parent(X,Y),L),
dom([parent(X,Y)|L],Y,Z).

dom(L,X,Z):-
leq(Y,X),
\+ memberList(leq(Y,X),L),
dom([leq(Y,X)|L],Y,Z).

```

And, once we have defined dominance, we can also define the top of a USR (recall that the top of a USR is the hole that out-scopes everything else; that is, it points to the SRL formula that we are trying to build):

```

top(X):-
dom(X,_),
\+ dom(_,X), !.

```

That is, a node X is a top node if it dominates some other node, and is not itself dominated by anything. Assuming all goes well (that is, an admissible plugging exists) the top node will be unique.

We can now calculate the admissible pluggings. First we retract previous attempts at plugging from the database and assert the new pluggings. Now, recall our discussion of what admissible means. We need to ensure that the plugging does not create cycles, and that distinct nodes with a common parent never dominate the same node:

```

admissiblePlugging(Plugs):-
retractall(plug(_,_)),
findall(X,(memberList(X,Plugs),assert(X)),_),
\+ dom(A,A),
\+ (parent(A,B), parent(A,C), \+ B=C, dom(B,D), dom(C,D)).

```

Once we have a plugging, we can recursively plug all holes with the available labels.

```

plugHoles([],_,Plugs):-
admissiblePlugging(Plugs).

plugHoles([H|Holes],Labels1,Plugs):-
admissiblePlugging(Plugs),
selectFromList(L,Labels1,Labels2),
plugHoles(Holes,Labels2,[plug(H,L)|Plugs]).

```

We are ready for Step 4. We now have not merely the USR, we also have a plugging (still asserted to the database!). We need to use these two sources of information to actually reconstruct a semantic representation. This is done by `url2srl/2`, which recursively converts

a USR into an ordinary first-order formula with respect to a plugging.

If a USR-term is a hole, it inspects the plugging for that hole and continues converting the label:

```
url2srl(H,F) :-  
    hole(H),  
    plug(H,L),  
    url2srl(L,F).
```

If a label points to a quantifier, the scope of that quantifier is further converted. Here is the definition for converting the universal quantifier:

```
url2srl(L,all(X,F)) :-  
    all(L,X,H),  
    url2srl(H,F).
```

For the booleans, again this is straightforward. Consider for instance the definition for plugging an implication:

```
url2srl(L,imp(F1,F2)) :-  
    imp(L,H1,H2),  
    url2srl(H1,F1),  
    url2srl(H2,F2).
```

Finally, basic formulas:

```
url2srl(L,F) :-  
    pred1(L,Symbol,Arg),  
    compose(F,Symbol,[Arg]).  
  
url2srl(L,F) :-  
    pred2(L,Symbol,Arg1,Arg2),  
    compose(F,Symbol,[Arg1,Arg2]).
```

Time to wrap all our plugging predicates together:

```
plugUSR(USR,Sem) :-  
    numbervars(USR,0,_),           % 1 Skolemise USR  
    initUSR,  
    assertUSR(USR),              % 2 Break down and assert USR  
    top(Top),  
    findall(H,hole(H),Holes),  
    findall(L,  
            (label(L),\+ parent(_,L)),  
            Labels),  
    plugHoles(Holes,Labels,[]),   % 3 Calculate a plugging  
    url2srl(Top,Sem).           % 4. Construct SRL formula
```

(The call to `initUSR`, incidentally, cleans up the database before the work starts.)

That completes our discussion of the plugging algorithm. So let's

now gather together all our work on hole semantics. The definition is much like those used in our work on Cooper and Keller storage (though see Exercise 3.4.1 for an interesting difference).

```
holeSemantics:-  
    readLine(Sentence),  
    parse(Sentence,USR),  
    printRepresentations([USR]),  
    setof(Sem, plugUSR(USR,Sem), Sems),  
    printRepresentations(Sems).
```

Here is an example session, showing the USR for every woman does not snort and the two obtained readings.

```
?- holeSemantics.  
  
> Every woman does not snort  
  
1 some(A, and(hole(A), some(B, and(label(B), some(C, some(D, some(E,  
        some(F, some(G, and(hole(C), and(label(D), and(label(E),  
            and(label(F), and(all(E,G,F), and(imp(F,D,C), and(leq(B,C),  
                and(leq(E,A), and(and(pred1(D,woman,G), leq(D,A)),  
                    some(H, some(I, and(hole(H), and(label(I), and(not(I,H),  
                        and(leq(I,A), and(leq(B,H), and(pred1(B,snort,G),  
                            leq(B,A)))))))))))))))))))))))
```

```
1 not(all(G, imp(woman(G), snort(G))))  
2 all(G, imp(woman(G), not(snort(G))))
```

The diagram below summarises our hole semantics setup.

Exercise 3.4.1 Our implementations of Cooper and Keller storage contained a filter to eliminate alphabetic variants of readings. But our hole semantics code contains no such filter. Is this an oversight on our part?

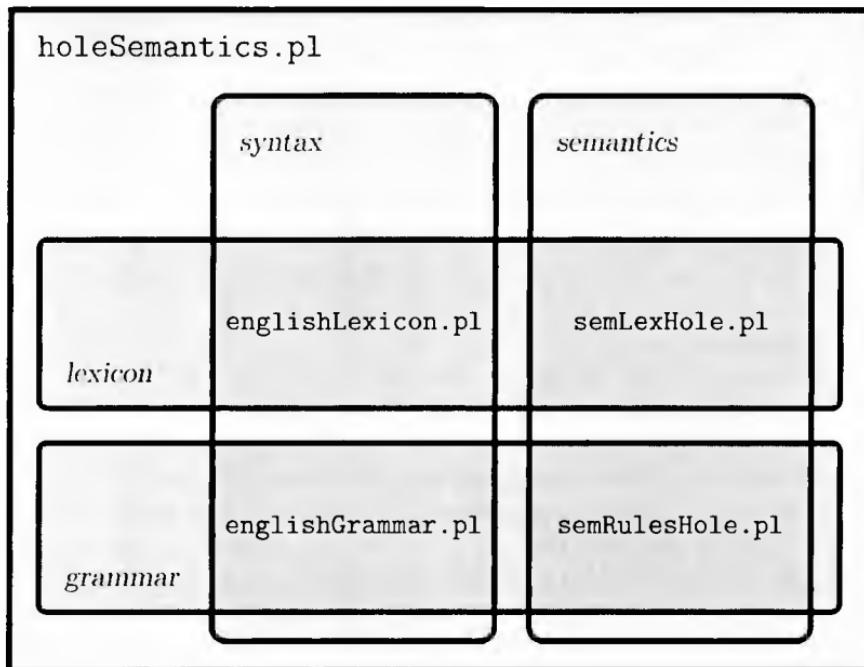
Exercise 3.4.2 The Prolog definition of the transitive closure of the dominance relation given in the text is perhaps (with its inclusion of `memberList/2` and an accumulator list of previously unaccounted for immediate dominance relations) slightly more complicated than expected. A simpler alternative could be:

```
dom(X,Y):- parent(X,Y).  
dom(X,Z):- parent(X,Y), dom(Y,Z).
```

Test this definition, by querying the goal `?- dom(A,A)`, on the following two databases (that is, we want to test whether the databases contain cycles):

1. `parent(a,b). parent(b,c). parent(c,a).`
2. `parent(d,a). parent(a,b). parent(b,c). parent(c,a).`

Although both of these instances of the `parent/2` relation contain cycles, the simple definition for `dom/2` only detects a cycle on the first database, and gets entangled in a loop on the second database. Explain why this is so, and



also explain how the definition for `dom/2` using an accumulator deals with both cases.

Exercise 3.4.3 According to some linguistic theories of quantifier scope, quantified noun phrases cannot be raised out of a relative clause, and hence cannot out-scope anything outside the clause (see Exercise 3.3.5). Change the semantic macro for relative clauses (by altering the dominance constraints) so that quantified noun phrases stay in the scope of relative clauses. (You need to do this in the file `semLexHole.pl`.)

Notes

Montague's paper "The Proper Treatment of Quantification in Ordinary English" (Montague, 1973) is the source of the quantifier raising method described at the start of the chapter. For a more accessible presentation of Montague's work, see Dowty et al. (1981). The sentence *Every owner of a hash bar gives every criminal a big kahuna burger*, which has 18 readings, 11 of which are logically distinct, was taken from Gabsdil and Striegnitz (2000); this paper describes a system for determining the logical relationships between sentences with multiple interpretations. For algorithms that generate non-redundant quantifier scopings, consult Vestre (1991) and Chaves (2003).

Programs for Underspecified Representations

cooperStorage.pl

Implementation of Cooper Storage. Defines storage of noun phrases and retrieval from the store.

kellerStorage.pl

Implementation of Keller Storage. Defines storage and retrieval.

holeSemantics.pl

Main file for hole semantics.

pluggingAlgorithm.pl

The plugging algorithm for hole semantics.

sentenceTestSuite.pl

Tests suite of natural language examples.

Cooper storage was first used in Cooper (1975); a more refined version is presented in Cooper (1983). Nested Cooper storage is due to Keller (1988). This very readable paper introduces Cooper storage, explains where the problems lie, gives an example (involving an interaction between scope and anaphoric pronouns) which suggests that a simple check for free variables during retrieval isn't an adequate solution, and then introduces nested Cooper storage, which we have called Keller storage. Hobbs and Shieber (1987) provide an alternative to Cooper storage that overcomes some of the problems mentioned in this chapter. An improved version of their algorithm has been developed by Lewin (1990).

Stores (and nested stores) are essentially a more abstract form of semantic representation—representations which encode multiple possibilities, without committing us to any particular choice. Viewed in this light, stores are ancestors of the current crop of underspecification formalisms, representation languages specifically designed to cope with ambiguity by avoiding overcommitment. (Indeed, the representations used in Schubert and Pelletier (1982), one of the earliest papers to take a computational perspective on Montague semantics, can also be viewed as precursors of the current generation of underspecification formalisms.) Nerbonne and Reyle seem to have been the first to use the term underspecification in connection with semantic representations for

natural language (see Nerbonne (1992b), Nerbonne (1992a) and Reyle (1992)).

Player (2004), a recent PhD thesis, gives an interesting high level overview and analysis of underspecification formalisms and their inter-relationships. Player classifies underspecification formalisms into three groups and in the following paragraphs we shall make use of Player's classification to structure our discussion.

The first group of underspecification formalisms that Player distinguishes is the "raising group". In essence, these are underspecification formalisms that directly trade on the quantifier raising idea that lies at the heart of Montague's original work. In this group Player places not only the storage methods we discussed in this chapter, but also Ambiguous Predicate Logic (Van Eijck and Jaspars, 1996), Quasi-Logical Form (Alshawi, 1992), and a logical formalism of his own that Player calls \mathcal{R} . Ambiguous Predicate Logic and \mathcal{R} are closely related, and both are interesting attempts to abstract away from the algorithmic details that drive the storage method and arrive at a simpler, more declarative, formulation of the underlying ideas.

The second group that Player distinguishes is the "holes and constraints group" of formalisms. This group includes Underspecified Discourse Representation Theory (see Reyle (1993)), Hole Semantics (see Bos (1996) and Bos (2001)), Minimal Recursion Semantics (see Copestake et al. (1995)) and the Constraint Language for Lambda Structures (see Egg et al. (1998) and Egg et al. (2001a)).

The intuition underlying the formalisms in this family is the one we met in the text: think about how the formulas of the underlying semantic representation language are put together, devise a constraint formalism for describing their structure, and specify a method for plugging together real semantic representations given the constraints. Underspecified Discourse Representations Structures, were the earliest such formalism; as their name suggests, they were designed not for first-order logic, but for Discourse Representation Structures (DRSs), the semantic representation formalism that lies at the heart of Discourse Representation Theory (Kamp and Reyle, 1993). Hole semantics was designed to generalise the approach pioneered by Reyle. The idea was to create a usable formalism that would be applicable to a variety of semantic representation formalism, and indeed hole semantics has been successfully adapted to both first-order logic (as in the text) and to DRSs (Bos, 2004), and has been used as part of a machine translation system Bos et al. (1996), Bos et al. (1998). Minimal Recursion Semantics (MRS) was designed to underspecify relatively flat semantic structures (that is, structures in which the degree of recursion has

been minimised). It is closely associated with the HPSG grammar formalism, and, like hole semantics, has been used in machine translation systems. The Constraint Language for Lambda Structures (CLLS) has been the subject of extensive formal investigation, and a lot is known about its computational and expressivity properties: important references include Duchier and Niehren (2000), Erk et al. (2003), Koller et al. (2000), Koller et al. (2003), Althaus et al. (2003), and Bodirsky et al. (2004).

The third group of underspecification formalisms that Player distinguishes has only a single member. This is an underspecified logic (Player calls it \mathcal{Q}) proposed in Van Eijck and Jaspars (1996), the same paper in which Ambiguous Predicate Logic was introduced. The distinguishing feature of \mathcal{Q} is that it offers an ambiguity connective: $\phi ? \psi$ is ambiguous between the reading ϕ and the reading ψ .

Player shows that although the formalisms in the three groups look very different, there is a sense in which they are all inter-reducible. On the basis of these reductions, Player argues that in spite of the apparent diversity of underspecification formalism, there is an underlying unity: all existing approaches have achieved roughly the same level of expressive power. Player also investigates a number of other issues in his thesis; for example he gives an account of inference in underspecified representation languages by means of a polynomial reduction to first-order logic.

Currently, the most active branch of research in underspecification revolves round the second group, the “holes and constraint group”. Player regards all the underspecification formalisms in this group as more-or-less obvious notational variants of one another. For Player’s purposes (establishing abstract general properties of a wide variety of underspecification formalisms) this perspective is useful, but it does have the drawback of masking the diversity of ideas that are currently being explored in this setting. For example, in the text we said that constraint solving was a key to understanding this kind of underspecification formalism. In fact, recent work on CLLS has moved on from viewing underspecification in terms of constraint solving towards identifying useful fragments of the underspecification language that can be viewed as graphs. This perspective opens the door to a new range of computational resources, namely graph algorithms. At the time of writing, the most efficient implementations of CLLS are graph-based: the solvability of an underspecification problem is defined in terms of certain cycles in the associated graph. The key references here are Althaus et al. (2003) and Koller (2004).

As the above remarks make clear, underspecification is currently

an active research area, and many different ideas are currently being explored. Probably the best advice we can give the reader interested in learning more is to consult both Player (2004) and Koller (2004). The first of these theses will give you a good birds-eye-view of the terrain, the second will give you detailed insight into the fine structure of the “holes and constraints” (and graphs!) family.

Another overview and analysis of underspecification you are likely to find useful is König and Reyle (1997). This surveys a number of underspecification formalisms, and compares their expressivity. It also addresses the topic of how inference can be directly performed on underspecified representations (a topic that we do not discuss in this book). Two other sources worth mentioning are Van Deemter and Peters (1996), a collection of papers, and Egg et al. (2001b), a special issue of the *Journal of Logic, Language and Information* devoted to underspecification.

One final remark. In this chapter we presented the USRs of hole semantics as certain kinds of (sorted) first-order formulas. This view of hole semantics comes from Bos (2001). The perspective is theoretically important (it justifies our use of lambda calculus as a glue language for USRs) but Bos also shows that it gives rise to a novel implementation technique: the use of model builders. Model building is discussed in Chapter 5; for present purposes it is enough to know that a first-order model builder takes a first-order formula as input and attempts to build a satisfying model. Bos’s experimental implementation gives a first-order model builder a hole semantics USR (that is, a certain kind of first-order formula) together with the first-order axioms for trees. The satisfying models produced as output are possible semantic representations. Current first-order model building technology is in its infancy, and this approach is currently very inefficient, but it will be interesting to see whether improvements are possible in the future.

Propositional Inference

In this chapter we turn to the second major theme of the book, namely:

How can we use logical representations of natural language expressions to automate the process of drawing inferences?

In Chapter 1 we defined three inference tasks of interest to computational semantics: the querying task, the consistency checking task, and the informativity checking task. Now, nothing more needs to be said about the querying task: it was the simplest task of all, and we dealt with it right away (recall that we implemented a first-order model checker). The consistency and informativity checking tasks, however, are a completely different kettle of fish. Both problems are undecidable—general algorithmic solutions don’t exist. Nonetheless, as we shall explain in this chapter and the next, there are some interesting partial solutions just waiting to be explored. In particular, using sophisticated theorem provers to perform negative checks for consistency and informativity, preferably backed up by model builders to provide partial positive checks, is an interesting way of exploring the role of inference in computational semantics.

This immediately raises a host of questions. What is a theorem prover? What is a model builder? Why distinguish negative and positive tests? And why the insistence on *sophisticated* inference tools? In this chapter we take our first steps towards providing some answers. We do so by introducing two theorem proving methods for propositional logic—that is, for the quantifier-free fragment of first-order logic.

4.1 From Models to Proofs

Our ultimate goal is to provide partial computational solutions to the consistency and informativity checking tasks for first-order logic with equality. Why ‘partial’? Because, as we discussed in Chapter 1, both these problems are undecidable: full computational solutions don’t ex-

ist. That's bad news, but we have some good news too: there's really only one task, for both informativity and consistency checking are definable in terms of validity.

Recall from Chapter 1 that when we say that a first-order formula ψ is uninformative with respect to first-order formulas ϕ_1, \dots, ϕ_n , we mean that

$$\phi_1, \dots, \phi_n \models \psi,$$

and that when we say that ψ is inconsistent with respect to ϕ_1, \dots, ϕ_n we mean (recall Exercise 1.2.7) that

$$\phi_1, \dots, \phi_n \models \neg\psi.$$

Furthermore, recall that (by the Semantic Deduction Theorem) both tasks can be reformulated in terms of the validity of single formulas: saying that ψ is uninformative with respect to ϕ_1, \dots, ϕ_n means that $\phi_1 \wedge \dots \wedge \phi_n \rightarrow \psi$ is valid, and saying that ψ is inconsistent with respect to ϕ_1, \dots, ϕ_n means that $\phi_1 \wedge \dots \wedge \phi_n \rightarrow \neg\psi$ is valid. So our quest for computational handles on the informativity and consistency checking tasks boils down to a single issue: are there computational techniques for determining whether a formula is valid?

So far so good—but if we are to make further progress with this problem, we have to re-conceptualise it. The definition of validity is semantic: it is defined in terms of models. Indeed, its definition is semantic in a very strong sense: a valid formula is one that is satisfied in *all* models. From a computational perspective, this definition is of limited interest: there are infinitely many models (and most of them are infinite), so we certainly cannot line them all up in a computer and check for validity by seeing if a formula is satisfied in them all! No—if we want a computational handle on validity checking, we need to find a way of viewing it as a concrete symbol manipulation task.

And this can be done. A branch of logic called *proof theory*, and a branch of computer science called *automated reasoning*, investigate logical validity from a purely syntactic perspective. Researchers in these fields have devised a number of *proof methods* for establishing validity, and implemented a wide range of *theorem provers* that turn these techniques into computational reality. The crucial thing about these techniques is that they only make use of the syntactic structure of formulas. To put it another way, these proof methods involve only symbol manipulation; models play no role. Of course, such methods should always be *justifiable* in model-theoretic terms. That is, if someone asks why some proposed proof method is really a way of establishing genuine logical validities—and not just some bizarre new way of playing with logical symbols—it should be possible to show that the method

is faithful to the semantic concept of validity, and later in the chapter we'll discuss what "being faithful to the semantic concept of validity" actually means. Nonetheless, the proof methods themselves are defined solely in terms of formula manipulation: they make no appeal to semantic concepts. Since we want a computational handle on validity, it is clearly sensible to try and make use of such techniques.

Many proof methods have been devised and studied: the best known include axiomatic (or Hilbert-style) systems, natural deduction systems, sequent calculi, tableau systems, and resolution systems. These proof methods were developed for different purposes and have different strengths and weaknesses. For example, axiomatic systems are excellent if one wants to study provability at an abstract level, but awful to actually use. Natural deduction systems, on the other hand, are easy for people to use (as their name suggests, they try to capture something of what is involved when a human being goes about proving something) but are not well suited for automated reasoning.

In this book we introduce the tableau and resolution methods. In our view, there is no better way for computational semanticists to get to grips with the basics of theorem proving than by learning about these methods. For a start, each is interesting in its own right. Moreover, the two methods are instructively different: working with them both will give the reader a sense of just how subtle (and difficult) theorem proving is. Finally, most of the sophisticated theorem provers the reader is likely to encounter (whether they be for first-order logic or something else) are likely to be either resolution or tableau based. Thus it is useful to have a basic grasp of both approaches.

We begin by discussing the tableau method. This method is conceptually simple and easy to understand, and (with the possible exception of natural deduction) is perhaps the proof method best adapted for pencil-and-paper calculation. This is because the method, although syntactic, transparently mirrors semantic intuitions—indeed, tableaus are often called *semantic tableaus*. Nonetheless, although tableaus offer an excellent human-oriented approach to inference, the basic mechanism is suitable for implementation, and efficient tableau-based theorem provers exist for many kinds of logic.

We then discuss resolution, a resolutely machine-oriented method. We shall present resolution as a two stage process. In the first stage the input formula is preprocessed into a special form called *conjunctive normal form*. In the second stage, a single rule, the *resolution rule*, is repeatedly applied to the result. Resolution is the most important method in contemporary automated theorem proving and (as readers of this book are probably aware) Prolog is based on it.

In this chapter we confine our discussion of the tableau and resolution methods to propositional logic. Recall from Chapter 1 and Appendix B that propositional languages are essentially a simple notation for the quantifier-free fragment of first-order languages. For example, instead of writing

$$(\text{DEAD(VINCENT)} \rightarrow \text{HAPPY(BUTCH)}) \wedge (\neg \text{DEAD(VINCENT)} \rightarrow \text{HAPPY(MIA)}),$$

which is a quantifier-free first-order formula, we would write something like

$$(p \rightarrow q) \wedge (\neg p \rightarrow r).$$

That is, in propositional logic we hide the internal structure of the atomic symbols, replacing them by *sentence symbols* such as p , q , and r . In propositional logic it is the syntactic configuration of the boolean connectives (\neg , \wedge , \vee , and \rightarrow) that is important.

Why start with propositional logic? Because it is a lot simpler than full first-order logic. For a start, propositional logic is decidable. That is, there are algorithms for determining whether an arbitrary propositional formula is valid or not (the method of truth tables, described in Appendix B is one such algorithm, and so are the tableau and resolution methods we shall develop in this chapter) whereas there is no algorithm for deciding whether or not an arbitrary first-order formula is valid. Moreover, the basic ideas of tableau and resolution shine through more clearly in propositional logic, for we don't have to deal with the tricky issue of how to cope with the quantifiers, an issue that will demand a great deal of our attention in the following chapter when we extend our discussion to first-order logic.

4.2 Propositional Tableaus

The key intuition underlying the tableau proof technique revolves around the following *semantic* question:

Suppose we are given a formula, and one of the truth values TRUE or FALSE. Is it possible to find a model in which the given formula has the given truth value?

The tableau method is essentially a *syntactic* way of answering this question. More precisely, a tableau proof is a *systematic* check, making use of only syntactic concepts, which lets us know whether or not it is possible to build a model in which some given formula is true or false.

Suppose that we had such a systematic check at our disposal. This would give us a way to test formulas for validity. After all, “valid” means “true in all models”, so a formula is valid if and only if it is not possible to falsify it in any model. Hence, a formula ϕ would be valid

if and only if the systematic method told us that there was no way to build a model that falsified ϕ . For this reason, the tableau method is often called a *refutation* proof method: we show that ϕ is valid by showing that all attempts to falsify it must fail.

So: what is a tableau system, and how do they systematically check for the existence of suitable models? We shall introduce the key ideas by presenting three examples of tableau proofs, taking care to point out the underlying semantic intuitions. We'll then make our discussion more precise.

Consider the formula $p \vee \neg p$. This is a validity—we certainly can't falsify it—but what would a systematic search for a falsification look like? Now, most readers probably know one way of conducting such a systematic search: use truth tables (if you don't know what a truth table is, read Appendix B). But truth tables aren't very appealing. For a start, while it's easy to fill out the truth table for $p \vee \neg p$, the truth table for a formula containing 8 different sentence symbols contains 256 lines, while the table for a formula containing 20 symbols contains 2^{20} lines, which is far too large for comfort. Moreover, the method of truth tables can't be extended to full first-order logic; we want a method that can.

So instead of truth tables we'll develop a number of *tableau expansion rules*. These rules will tell us how to make complex formulas true (or false) by breaking them down into their component formulas and giving the components the appropriate truth value. Let's see what sort of rules are needed, and how to use them, by constructing a tableau proof of $p \vee \neg p$.

We want to try and falsify $p \vee \neg p$, so let's introduce a piece of notation to express this goal. Writing $F\phi$, where ϕ is any formula, will mean that we want to make ϕ false. Similarly, writing $T\phi$ will mean that we want to make ϕ true. (T and F are called *signs*, and a formula preceded by a sign is called a *signed formula*.) Thus, as we are going to try to falsify $p \vee \neg p$, our initial goal is:

$$F(p \vee \neg p).$$

This rather trivial looking object is our first example of a tableau. Actually, when writing out tableaus by hand, it is handy to include a little extra book-keeping information, such as line numbers. So, in practice we'd tend to write the above tableau as:

$$1 \quad F(p \vee \neg p)$$

How do we proceed? Essentially we use the tableau expansion rules to smash the signed formula into smaller and smaller pieces until we

reach the atomic level. So, what expansion rule should we apply here? Obviously we need a rule that tells us how to *falsify a disjunction*. The required rule is clear: to make a disjunction false, falsify both disjuncts. So, applying this rule (let's call it F_V) we expand our one line tableau to a three line tableau as follows:

1	$F(p \vee \neg p)$	✓
2	Fp	1, F_V
3	$F\neg p$	1, F_V

Here, lines 2 and 3 are the extra information we have deduced by applying the F_V rule. The third column contains more book-keeping information: the ✓ symbol in line 1 records the fact that we've applied the appropriate rule to line 1, while the annotations 1, F_V in lines 2 and 3 record the fact that these lines were obtained from line 1 using rule F_V .

What next? In fact, there's only one more thing we can do. We've already applied a rule to line 1, so we've finished with that. Furthermore, line 2 tells us something about *atomic* information (namely that we need to make p false). This is simply a blunt fact, not something that can be further analysed. Thus only line 3 offers us the possibility of further progress; it tells us that we need to falsify the negation of p . So, we need an expansion rule that tells us how to do this. Again, the required rule is clear: to make the negation of a formula false, make the formula itself true. By applying this rule (let's call it F_{\neg}) we can expand our three line tableau to a four line tableau as follows:

1	$F(p \vee \neg p)$	✓
2	Fp	1, F_V
3	$F\neg p$	1, F_V , ✓
4	Tp	3, F_{\neg} .

Note that we have marked line 3 with a ✓ (thus recording that the applicable rule has been applied) and have indicated that line 4 was obtained from line 3 using F_{\neg} .

There are two important observations that must be made about this tableau. First, it is *rule-saturated*. That is, we cannot expand it any more. Either we've already applied the applicable rule (lines 1 and 3), or the line contains instructions about what to do with atomic information (lines 2 and 4). Second, the tableau is *closed*. That is, it contains contradictory instructions. The tableau tells us that if we want to falsify $p \vee \neg p$, we have to make p false (line 2) and p true (line 4). As this is impossible, and as it should be clear that the above tableau really does indicate all possible ways of falsifying $p \vee \neg p$, we conclude

that this formula is valid. This closed tableau is called a *tableau proof* of $p \vee \neg p$.

The previous example is a perfectly good tableau proof, but it's also one of the simplest the reader is ever likely to see. So let's consider a slightly more demanding task: testing $\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$ for validity.

Just as in the previous example, our initial goal is to try and falsify the given formula. Thus our initial tableau is:

$$1 \quad F \neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$$

Now we need to falsify an implication, so we need an expansion rule that tells us how to do this. Again the required rule is clear: to falsify an implication, make the antecedent true and the consequent false. Applying this rule (let's call it F_{\rightarrow}) yields the following tableau:

1	$F \neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	✓
2	$T \neg(q \wedge r)$	$1, F_{\rightarrow}$
3	$F(\neg q \vee \neg r)$	$1, F_{\rightarrow}$

Now, line 3 demands that we falsify a disjunction. We've already met the required expansion rule, namely F_{\vee} . Applying it to line 3 yields:

1	$F \neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	✓
2	$T \neg(q \wedge r)$	$1, F_{\rightarrow}$
3	$F(\neg q \vee \neg r)$	$1, F_{\rightarrow}, \checkmark$
4	$F \neg q$	$3, F_{\vee}$
5	$F \neg r$	$3, F_{\vee}$

At this point the alert reader should be saying “Hold on a minute! Why are we free to apply this rule to line 3? Sure, the rule fits—but look at line 2. There's a formula there that we need to make true. Don't we need to take care of that first?”

A sensible question, but the answer is: *no, we don't*. One of the pleasant things about propositional tableau is that we are free to apply applicable rules in any order we like. Yes, we could have applied the relevant rule to line 2 at this point, had we wanted to—but we're equally free to apply a rule to line 3, just as we did above. Propositional tableau rules tell us what we're *permitted* to do to expand a tableau; we're not forced to apply expansion rules in any particular order.

So let's move on. Note that both lines 4 and 5 ask us to falsify a negated formula. Again, we have already met the relevant rule, namely F_{\neg} . Applying it first to line 4, and then to line 5 (a completely arbitrary ordering choice) yields:

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	✓
2	$T\neg(q \wedge r)$	1, F_{\neg}
3	$F(\neg q \vee \neg r)$	1, F_{\neg}, \checkmark
4	$F\neg q$	3, F_{\vee}, \checkmark
5	$F\neg r$	3, F_{\vee}, \checkmark
6	Tq	4, F_{\neg}
7	Tr	5, F_{\neg}

Let's now deal with line 2. (In fact, there is nothing else we can do.) We need to make a negation true. The required rule is clear: to make the negation of a formula true, make the formula itself false. So, applying this rule (let's call it T_{\neg}) we can expand our tableau to obtain:

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	✓
2	$T\neg(q \wedge r)$	1, F_{\neg}, \checkmark
3	$F(\neg q \vee \neg r)$	1, F_{\neg}, \checkmark
4	$F\neg q$	3, F_{\vee}, \checkmark
5	$F\neg r$	3, F_{\vee}, \checkmark
6	Tq	4, F_{\neg}
7	Tr	5, F_{\neg}
8	$F(q \wedge r)$	2, T_{\neg}

Now we have to deal with line 8, which asks us to falsify a conjunction. Doing so leads us to the first real complication in the story we have been telling. The point is this: there's not just one way of making a conjunction false, there are two. Making either conjunct false will falsify the whole formula. As tableaus are meant to be *systematic* searches for falsifications, we're going to have to examine both possibilities. Thus the relevant expansion rule (let's call it F_{\wedge}) is going to yield two alternative ways of expanding the tableau, and we're going to have to keep track of both.

F_{\wedge} is our first example of a *disjunctive* (or *branching*) expansion rule; we'll encounter more such rules shortly. Because of such rules, tableaus *won't* generally consist of a single straight line down the page (that is, they're no longer going to consist of a single *branch*, to use the official terminology). Rather, they will be tree-like structures, possibly containing many branches. In the present case, what we get after applying F_{\wedge} is:

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	✓
2	$T\neg(q \wedge r)$	1, F_{\rightarrow} , ✓
3	$F(\neg q \vee \neg r)$	1, F_{\rightarrow} , ✓
4	$F\neg q$	3, F_V , ✓
5	$F\neg r$	3, F_V , ✓
6	Tq	4, F_{\neg}
7	Tr	5, F_{\neg}
8	$F(q \wedge r)$	2, T_{\neg} , ✓
9	Fq	8, F_{\wedge}
10	Fr	8, F_{\wedge}

That is, we have recorded two distinct possibilities: we must either falsify q , or falsify r .

But let's return to our tableau proof. What do we do next? Actually, we've finished: our two-branch tableau is rule-saturated, as the reader can easily check. So, after all that, is $\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$ a validity? Yes, it is. This is because the rule-saturated tableau we produced is *closed*, which means that *all the branches it contains are closed*. To see this, note that the left-hand branch is closed because it contains the contradictory instructions Fq (at line 9) and Tq (at line 6); whereas the right-hand branch is closed because it contains Fr (at line 10) and Tr (at line 7). This closed tableau is a tableau proof of $\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$.

Let's consider a final example to illustrate what happens if the formula we are working with is *not* a validity. Now, the formula $(p \wedge q) \rightarrow (r \vee s)$ is certainly not valid; what happens when we try and falsify it using the tableau method?

As usual, the first step simply states our goal; namely :

$$1 \quad F(p \wedge q) \rightarrow (r \vee s)$$

As we need to falsify an implication, we apply expansion rule F_{\rightarrow} :

1	$F(p \wedge q) \rightarrow (r \vee s)$	✓
2	$T(p \wedge q)$	1, F_{\rightarrow}
3	$F(r \vee s)$	1, F_{\rightarrow}

Now line 2 requires us to make a conjunction true. The expansion rule T_{\wedge} required is clear: we must make both conjuncts true. Applying T_{\wedge} yields:

1	$F(p \wedge q) \rightarrow (r \vee s)$	✓
2	$T(p \wedge q)$	1, F_{\rightarrow} , ✓
3	$F(r \vee s)$	1, F_{\rightarrow}
4	Tp	2, T_{\wedge}
5	Tq	2, T_{\wedge}

Now let's deal with line 3. The relevant rule is F_V . Applying it yields:

1	$F(p \wedge q) \rightarrow (r \vee s)$	✓
2	$T(p \wedge q)$	1, F_{\neg} , ✓
3	$F(r \vee s)$	1, F_{\neg} , ✓
4	Tp	2, T_{\wedge}
5	Tq	2, T_{\wedge}
6	Fr	3, F_V
7	Fs	3, F_V

But now the expansion process halts. This tableau is rule-saturated: we've applied the relevant rules to lines 1–3, while lines 4–7 simply stipulate what has to be done with atomic information. But note that there is a crucial difference between this tableau and the previous rule-saturated tableau we have seen: the single branch in this tableau is not closed, it's *open*. That is, it does *not* contain contradictory instructions. In fact, it gives us very sensible instructions indeed: lines 4–7 tell us to make p true, q true, r false, and s false. As the reader can easily check, doing this falsifies $(p \wedge q) \rightarrow (r \vee s)$, thus this formula is *not* a validity.

More generally, a rule-saturated tableau is called *open* iff it contains at least one open branch. If we obtain an open tableau when we try to falsify some formula, this means that the formula is *not* a validity. Moreover, just as in the previous example, every open branch on the open tableau actually contains an explicit falsification recipe for the formula: we falsify it by assigning truth values to the sentence symbols in the way the open branch stipulates. To put it another way, open branches of rule-saturated tableaus tell us how to build a propositional model that falsifies the formula we started with.

The reader should now have a fairly clear grasp of the main ideas and intuitions underlying tableau proofs, so let's now discuss the method more systematically. We begin by listing the eight main expansion rules, and classifying them into three types: *unary rules*, *conjunctive rules*, and *disjunctive rules*. This classification isn't vital, but it will help us keep the implementation neat.

For each boolean connective B , we have two tableau rules, T_B and F_B . T_B tells us how to make a formula with B as its main connective true, while F_B tells us how to make it false. The rules for negation, T_{\neg} and F_{\neg} , are the simplest:

$$\frac{T_{\neg}\phi}{F\phi} \quad \frac{F_{\neg}\phi}{T\phi}$$

These rules should be read from top to bottom. Given the examples discussed in the previous section, their meaning should be clear. In each rule, the signed formula above the horizontal line is the *input* to the rule, and the signed formula below it is the *output*. For example, T_{\neg} takes as input signed formulas of the form $T\neg\phi$, and returns as output signed formulas of the form $F\phi$. Similarly, F_{\neg} takes as input signed formulas of the form $F\neg\phi$ and returns as output signed formulas of the form $T\phi$. In what follows, we shall call these two rules our *unary* rules. This is simply shorthand for the fact that both rules return a single formula as output.

The rules for the binary connectives \wedge , \vee and \rightarrow are rather more interesting. Here are T_{\wedge} and F_{\wedge} , F_{\vee} and T_{\vee} , and F_{\rightarrow} and T_{\rightarrow} , respectively:

$$\begin{array}{ll} \frac{T(\phi \wedge \psi)}{\begin{array}{c} T\phi \\ T\psi \end{array}} & \frac{F(\phi \wedge \psi)}{\begin{array}{c} F\phi \\ | \\ F\psi \end{array}} \\[10mm] \frac{F(\phi \vee \psi)}{\begin{array}{c} F\phi \\ F\psi \end{array}} & \frac{T(\phi \vee \psi)}{\begin{array}{c} T\phi \\ | \\ T\psi \end{array}} \\[10mm] \frac{F(\phi \rightarrow \psi)}{\begin{array}{c} T\phi \\ F\psi \end{array}} & \frac{T(\phi \rightarrow \psi)}{\begin{array}{c} F\phi \\ | \\ T\psi \end{array}} \end{array}$$

Again, all six rules should be read from top to bottom, the top being the input to the rule, the bottom the output. We shall call the three rules in the left-hand column (that is, T_{\wedge} , F_{\vee} , and F_{\rightarrow}) *conjunctive rules*. The three rules on the right-hand side (that is, F_{\wedge} , T_{\vee} , and T_{\rightarrow}) are called *disjunctive rules*. Note that each of \wedge , \vee and \rightarrow gives rise to a pair of rules, one of which is conjunctive, the other disjunctive. Both conjunctive and disjunctive rules return two formulas as output—however, as we have already seen, they differ in the effect they have on tableau. In particular, use of a disjunctive rule splits the branches of the tableau containing the input formula into distinct branches, each of which records one of the two alternative output formula.

In the discussion that follows, we'll assume that we only have these eight rules at our disposal. However, it is easy to give expansion rules for other connectives, though such rules won't always fit into our three way classification. For example, it can be useful to have the expansion

rules for the bi-implication connective \leftrightarrow at our disposal. Here are the required rules:

$T(\phi \leftrightarrow \psi)$		$F(\phi \leftrightarrow \psi)$	
$T\phi$	$F\phi$	$T\phi$	$F\phi$
$T\psi$	$F\psi$	$F\psi$	$T\psi$

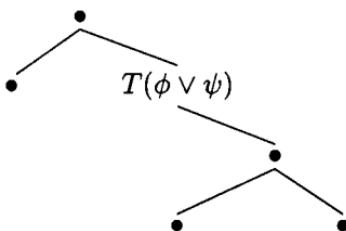
That is, for this connective we don't obtain a neat pair of rules, one of which is conjunctive, the other disjunctive, as we did for \wedge , \vee and \rightarrow . Both rules are disjunctive, for both force us to split tableau branches, but unlike the previous disjunctive rules, these rules yield four output formulas, two for each branch. Exercise 4.2.2 asks the reader to define tableau expansion rules for two other connectives.

Now that we know what our expansion rules are, let's make our previous discussion of tableau and tableau proofs a little more rigorous. A (propositional) *tableau* is simply a tree, each of whose nodes is a signed (propositional) formula. A *branch* of a tableau is simply a branch of such a tree—that is, a collection of nodes (that is, signed formula) that contains exactly one leaf node together with all the nodes which dominate it. An *initial tableau*—that is, the kind of tableau with which we start the tableau expansion process—is a tableau consisting of a single signed formula.

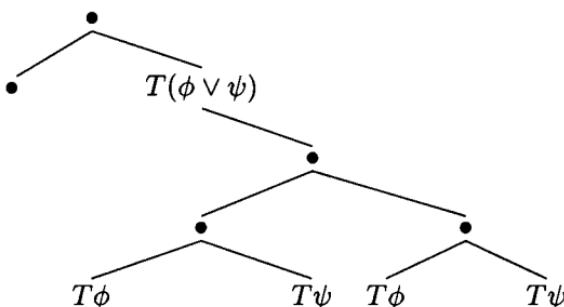
We carry out the tableau expansion process as follows. Given a tableau, we try to find a node in it that isn't a signed atomic formula, and to which we haven't already applied an expansion rule. Let's call such nodes *unexpanded nodes*. If there are no unexpanded nodes, we can't do anything: we have a *rule-saturated* tableau and are finished. So suppose there is at least one unexpanded node. This node has the form $S\phi$, where S is either T or F , and ϕ is a propositional formula. Moreover, as ϕ is not atomic, it has a main connective B , where B is one of the connectives \neg , \wedge , \vee , or \rightarrow . We can then read off the required rule: we have to apply rule S_B .

What happens when we apply a rule to a node? That depends on whether the rule is unary, disjunctive, or conjunctive. If the rule is unary, we extend every branch containing the input node by adding on the output formula of the rule as the new leaf node. If the rule is conjunctive, we extend every branch containing the input node by adding on, again at the leaf node, both output formula of the rule. Finally, if the rule is disjunctive, we extend every branch containing the input node to two distinct branches, one containing each of two choices of output formulas. Again, both the required additions are carried out at the leaf node of the original branches.

The basic idea of tableau expansion should be clear from the examples we discussed earlier, but there is one point worth emphasising. A signed formula may belong to several branches. (For example, the root node belongs to every branch of a tableau.) When we perform an expansion, we have to extend *every* branch on which the input formula lies in the appropriate way. For example, consider the following tree:



If we expand the indicated node $T(\phi \vee \psi)$, we obtain:



The tableau expansion process starts when we are given an initial tableau (typically, a single formula prefixed by F). We apply rules to the initial tableau, and then to the tableau obtained by earlier rule applications, and so on, until it is not possible to apply any more rules. As we discussed earlier, we can apply the rules in any order we like. The tableau expansion process stops when it is not possible to apply any more rules, that is, when we obtain a rule-saturated tableau.

We are now almost ready to say what a *tableau proof* is. First, a branch of a tableau is *closed* if it contains both $T\phi$ and $F\phi$, where ϕ is some formula. A branch that is not closed is called *open*. A tableau is closed if *every* branch it contains is closed, and open if it contains *at least one* open branch. Now for the key definition:

A propositional formula ϕ is tableau-provable if and only if it is possible to expand the initial tableau consisting of the single node $F\phi$ to a closed tableau. We use the notation $\vdash_t \phi$ to indicate that ϕ is tableau-provable. If a formula is tableau-provable, then we say that it is a tableau-theorem, or more simply, a theorem.

And that's the (propositional) tableau method. To conclude our discussion, one point is worth stressing: the tableau method really is a purely *syntactic* method. If we want to test whether ϕ is provable, we write down the symbols $F\phi$, and then try to build a closed tableau. The rules governing the expansion process are completely syntactical: when we expand a node, we know which rule to apply simply looking at its sign and main connective. Moreover, closure is a purely syntactic concept: it simply amounts to looking for items of the form $T\phi$ and $F\phi$ on some branch. Of course, as our presentation has tried to emphasise, the tableau method is driven by clear *semantic* ideas. Nonetheless, *using* the method doesn't require any semantic insight at all. Even if we didn't know what the signs T and F were meant to stand for—or indeed, what \neg , \wedge , \vee , and \rightarrow were meant to represent—we would still have a well defined way of manipulating logical formulas. It would probably be an exaggeration to claim that we could train a monkey to carry out tableau proofs—but as we shall soon see, it is easy to implement the method in Prolog.

Exercise 4.2.1 Give tableau proofs of the following formulas:

1. $\neg\neg p \rightarrow p$
2. $((p \rightarrow q) \rightarrow p) \rightarrow p$
3. $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$
4. $p \rightarrow (p \wedge (q \vee p))$
5. $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r))$.

Exercise 4.2.2 The connectives *nand* and *nor* are defined by the following truth tables:

<i>p</i>	<i>q</i>	<i>p nand q</i>	<i>p nor q</i>
TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	TRUE

Give the tableau expansion rules for both connectives.

Exercise 4.2.3 In Exercise 1.1.13 we introduced the special boolean constants \perp (always false) and \top (always true). Extend the tableau system so that it handles them. (Hint: add new rules for branch closure.)

4.3 Implementing Propositional Tableau

We shall represent a tableau in Prolog as a list containing lists of signed formulas. Signed formulas will be represented as Prolog terms with

functors f and t of arity 1. For example, the following is the Prolog representation of a tableau:

$$[[f(\text{imp}(p,q))]].$$

This represents the tableau consisting of just one branch, namely a branch containing the single signed formula $F(p \rightarrow q)$.

Here's a second example:

$$[[t(\text{and}(p,q)), f(\text{and}(p,r))]].$$

Again, this represents a tableau with just one branch. This time, however, the branch contains two signed formulas, namely $T(p \wedge q)$ and $F(p \wedge r)$.

Our Prolog implementation of the propositional tableau method consists of a small collection of predicates which manipulate such lists in accordance with the tableau expansion rules. For example, the list just given contains $t(\text{and}(p,q))$. Now, to make a conjunction true, we use expansion rule T_\wedge , which in this case tells us to add Tp and Tq to the given branch. It is easy to mimic the required expansion in Prolog: all we have to do is define a predicate which when given the previous list as input, returns the following one as output:

$$[[t(p), t(q), f(\text{and}(p,r))]].$$

Note two things. First, as expected, the output list contains $t(p)$ and $t(q)$. Second, it does *not* contain $t(\text{and}(p,q))$. Why is this? Because once the relevant expansion rule has been applied to some formula, that formula is no longer relevant and can be removed. That is, removal of signed formulas from lists is the Prolog analog of the \checkmark marking we used in our handwritten proofs.

Note that our new list contains $f(\text{and}(p,r))$, thus we can process it even further. Now, the rule for falsifying a conjunction is F_\wedge , a *disjunctive* rule. How are disjunctive expansions to be handled in Prolog? Again, the basic idea is very simple. To handle F_\wedge , for example, we need simply define a predicate which when given the previous list as input, returns the following one as output:

$$[[t(p), t(q), f(p)], [t(p), t(q), f(r)]].$$

Note that the list now contains two lists. It is thus the Prolog representation of a tableau containing two branches, namely the tableau with the signed formulas Tp , Tq and Fp on one (closed) branch, and the formulas Tp , Tq and Fr on the other.

Note that this approach to disjunctive rules differs from our handwritten approach. Our Prolog program makes two copies of the branch, one for each of the two disjunctive possibilities. This is somewhat heavy

handed—but it makes the code easy to understand. On the other hand, when writing out proofs by hand we want to cut down the writing needed as much as possible—hence our use of tree-like structures in which the information common to two branches is shared.

Summing up, our Prolog implementation is based round the representation of tableaus as lists containing lists of signed formulas. Each list of signed formula corresponds to a branch. We carry out expansions by manipulating these lists of signed formulas in the appropriate way. In particular, conjunctive (and unary) expansions will be carried out by taking the list representation of a branch, removing the relevant conjunctive (or unary) signed formula, and adding its component(s) to the list. Disjunctive expansions are carried out in much the same way, save that the process returns *two* lists, one containing each of the two possible components.

Let's go systematically through the program. The top-level predicate, `tprove/1` is defined as follows:

```
tprove(F) :-
    (
        closedTableau([[f(F)]]), !,
        write('Theorem.'), nl
    ;
        write('Not a theorem.'), nl
    ).
```

That is, when given a propositional formula `F` as input, `tprove/1` forms the initial tableau `[[f(F)]]` and then calls `closedTableau/1` to try and expand it to a closed tableau. If this succeeds, it declares that `F` is a theorem; if it fails, it tells us that formula `F` is not a theorem.

Here's the definition of `closedTableau/1`, the heart of the prover:

```
closedTableau([]).

closedTableau(OldTableau) :-
    expand(OldTableau,TempTableau),
    removeClosedBranches(TempTableau,NewTableau),
    closedTableau(NewTableau).
```

How does this work? The first clause defines the empty list to be closed. Why? Because whenever we encounter a closed branch in our tableau, we are going to discard it. This means that the empty list is a tableau from which all branches have been discarded—so if we are ever left with the empty list, we have successfully built a closed tableau and should halt. The second clause applies when the tableau built so far (that is, `OldTableau`) is not closed. In this case we first

use `expand/2` to build a temporary tableau, and then (via a call to `removeClosedBranches/2`) we weed out any closed branches to form `NewTableau`. We use this new tableau as the argument of the recursive call to `closedTableau/1`.

The remaining predicates fill in the details. Let's start with `expand/2`. This is a high level organisational predicate that works its way recursively through all the branches in the input tableau. Its first three clauses look for signed formulas to which a unary expansion, a conjunctive expansion, or a disjunctive expansion can be applied. The fourth clause makes the recursive call that deals with the next branch of the tableau if no more expansions can be carried out to the current branch:

```

expand([Branch|Tableau], [NewBranch|Tableau]) :-
    unaryExpansion(Branch, NewBranch), !.

expand([Branch|Tableau], [NewBranch|Tableau]) :-
    conjunctiveExpansion(Branch, NewBranch), !.

expand([Branch|Tableau], [NewBranch1, NewBranch2|Tableau]) :-
    disjunctiveExpansion(Branch, NewBranch1, NewBranch2), !.

expand([Branch|Rest], [Branch|Newrest]) :-
    expand(Rest, Newrest).

```

Now for the predicates that do the real work. Both the predicates `unaryExpansion/2` and `conjunctiveExpansion/2` have two arguments, namely an input branch, and an output branch. On the other hand, the predicate `disjunctiveExpansion/3` takes three arguments—an input branch and *two* output branches. All three predicates look for the occurrence of the appropriate type of signed formula (namely unary, conjunctive, and disjunctive respectively) as the first item in the input branch, then use the library predicate `removeFirst/3` to remove that occurrence, and finally build the required new branch (or branches, in the case of disjunctive formulas) out of the component (or components) of the signed formula:

```

unaryExpansion(Branch, [Component|Temp]) :-
    unary(SignedFormula, Component),
    removeFirst(SignedFormula, Branch, Temp).

conjunctiveExpansion(Branch, [Comp1, Comp2|Temp]) :-
    conjunctive(SignedFormula, Comp1, Comp2),
    removeFirst(SignedFormula, Branch, Temp).

```

```
disjunctiveExpansion/Branch, [Comp1|Temp], [Comp2|Temp]) :-  
    disjunctive(SignedFormula, Comp1, Comp2),  
    removeFirst(SignedFormula, Branch, Temp).
```

Of course, we have to say what conjunctive, disjunctive, and unary signed formulas actually are:

```
conjunctive(t(and(X,Y)), t(X), t(Y)).  
conjunctive(f(or(X,Y)), f(X), f(Y)).  
conjunctive(f(imp(X,Y)), t(X), f(Y)).
```

```
disjunctive(f(and(X,Y)), f(X), f(Y)).  
disjunctive(t(or(X,Y)), t(X), t(Y)).  
disjunctive(t(imp(X,Y)), f(X), t(Y)).
```

```
unary(t(not(X)), f(X)).  
unary(f(not(X)), t(X)).
```

Only one task remains—we need to define the predicate that removes closed branches from the tableau:

```
removeClosedBranches([], []).  
  
removeClosedBranches([Branch|Rest], Tableau) :-  
    closedBranch(Branch), !,  
    removeClosedBranches(Rest, Tableau).  
  
removeClosedBranches([Branch|Rest], [Branch|Tableau]) :-  
    removeClosedBranches(Rest, Tableau).
```

Note the Prolog cut (!) in the second clause. If we find a closed branch in a tableau, then we want to throw it away, and this is a decision that never needs to be reconsidered; the cut ensures it never will be.

Finally, note that closed branches are defined in the obvious way: a branch is closed if it contains $t(X)$ and $f(X)$ for some propositional formula X :

```
closedBranch(Branch) :-  
    memberList(t(X), Branch),  
    memberList(f(X), Branch).
```

Well, that's the propositional tableau prover. Time to test it. In the file `propTestSuite.pl` you will find entries of the form

```
formula(imp(p,not(not(p))), 'Theorem').
```

and

```
formula(imp(or(p,not(q)),or(p,q)), 'Not a theorem').
```

That is, the first argument of `formula/2` is a propositional formula, and the second records its status (that is, whether it is a theorem or a non-theorem). If you load the file `propTableau.pl` and give the command

```
?- tproveTestSuite.
```

you will generate output like

```
Input formula: imp(and(r,or(p,q)),or(and(r,p),and(r,q)))
Status: Theorem
Prover says: Theorem.
```

Before moving on, play with the prover. A good way to start is by running the test suite. Then try the following exercises.

Exercise 4.3.1 Try the prover out on some simple examples. Make sure you understand what is happening at each step. One way to do this is to use the standard Prolog `trace/0`, but a nicer way is to add the following code as a new second clause for the predicate `closedTableau/1`:

```
closedTableau(OldTableau):- nl, write(OldTableau), nl, fail.
```

Because this new clause always fails, it has no effect on the correctness of the predicate. However, before it fails it will write out the current state of the tableau, enabling you to follow the prover's progress easily.

Exercise 4.3.2 Try to find examples of theorems which the tableau prover can't handle, or can't handle fast. That is, try to find propositional formulas which you know to be valid but which the tableau prover either won't halt on, or takes a long time to halt. (The last two formulas in the test suite are like this, but try to find your own examples before looking at these.)

Exercise 4.3.3 Modify the code so that the tableau prover handles the bi-implication connective \leftrightarrow (use, say, `bimp/2` as the Prolog notation for this connective).

Exercise 4.3.4 Modify the code so that it handles the `nand` and `nor` connectives defined in Exercise 4.2.2.

Exercise 4.3.5 Building on the work of Exercise 4.2.3, extend the tableau prover so that it copes with \perp and \top .

Exercise 4.3.6 Reimplement propositional tableau in a way that is more faithful to our hand-written tree-based approach. That is, find a nice way of representing trees so that we don't need to duplicate branches when using disjunctive rules.

Exercise 4.3.7 Add a pretty print predicate to our implementation of propositional tableau, that shows the branches and the proof steps in a readable way.

Exercise 4.3.8 Implement the truth table method in Prolog (this method is described in Appendix B). Test your code using `propTestSuite.pl`. Can you find examples where the truth table method works better than the tableau method?

Programs for propositional tableau

`propTableau.pl`

The file that contains all the predicates for the implementation of our tableau-based theorem prover.

`propTestSuite.pl`

Test suite with problems to test our theorem provers.

`comsemPredicates.pl`

Contains some auxiliary predicates for list processing.

4.4 Propositional Resolution

We now examine a second technique for establishing the validity of propositional formulas: the resolution method. Like the tableau method it is purely symbolic (it is defined solely in terms of the syntactic structure of formulas) but it is different in two important respects. Firstly, the tableau method is based on the idea of multiple rules, two for each connective, which enable the input formula to be systematically dismantled; the resolution method, on the other hand, makes use of only a single rule (the resolution rule) which is repeatedly applied. Furthermore, tableau rules are directly applied to the input formula; in the resolution method (at least in its most common variants) the input formula is first converted to a special form (conjunctive normal form) and only then is the resolution rule applied.

Our discussion of resolution falls into two parts. We first discuss the preprocessing phase (the reduction to conjunctive normal form) and then the resolution phase proper.

Conjunctive Normal Form (CNF)

First some terminology:

A positive literal is a sentence symbol (for example, p, q, r, s, t, \dots). A negative literal is a negated sentence symbol (for example, $\neg p, \neg q, \neg r, \neg s, \neg t, \dots$). A literal is a positive literal or a negative literal. A clause is a disjunction of literals.

For example, $p \vee q \vee \neg r \vee s \vee \neg t$ is a clause, for each disjunct is a literal. Note that clauses are essentially ‘flat’ formulas. To be sure, the formula $p \vee q \vee \neg r \vee s \vee \neg t$ is shorthand for some bracketed formula, perhaps $p \vee (q \vee ((\neg r \vee s) \vee \neg t))$, or perhaps $(p \vee (q \vee \neg r)) \vee (s \vee \neg t)$. But, as we discussed in Exercise 1.2.3, such bracketings don’t matter semantically: disjunction is associative, hence all such variants are logically equivalent. To emphasise the flat nature of clauses (and the fact that we don’t need to worry about the bracketing) in the following discussion we shall generally write clauses using a list notation: for example, the clause $p \vee q \vee \neg r \vee s \vee \neg t$ will be written as $[p, q, \neg r, s, \neg t]$.

Now for the key semantic observation concerning clauses:

To make a clause true we have to make at least one of the literals it contains true (after all, a clause is a disjunction).

For example, to make $[p, q, \neg r, s, \neg t]$ true we have to make one of $p, q, \neg r, s$ or $\neg t$ true. This observation has an important special case. In resolution theorem proving it is usual to exploit the list notation to define a special clause, the empty clause, which we shall write as $[]$. As this notation makes clear, the empty clause contains no literals at all, hence it is impossible to make at least one of them true, and hence it is impossible to make the empty clause true. That is, the empty clause is essentially a different notation for the constant \perp (introduced in Exercise 1.1.13) which is always false. The empty clause plays an important role in resolution; as we shall see, resolution is essentially a determined effort to try and generate an empty clause.

But this is jumping ahead—we haven’t yet said what conjunctive normal forms are. Here’s the answer:

A formula is in conjunctive normal form (CNF) if and only if it is a conjunction of clauses.

For example

$$(p \vee q) \wedge (r \vee \neg p \vee s) \wedge (q \vee \neg s)$$

is in CNF, for it consists of three clauses (namely $p \vee q$, $r \vee \neg p \vee s$, and $q \vee \neg s$) conjoined together.

In what follows, we won’t usually write clauses using standard logical notation—instead we’ll extend the list notation just introduced

for clauses so that it covers formulas in CNF too. Consider the CNF formulas just given. First of all, we can replace each clause by its list representation to obtain

$$[p, q] \wedge [r, \neg p, s] \wedge [q, \neg s].$$

Having done this, it is natural to stop representing \wedge explicitly and to use a list of lists representation instead, namely:

$$[[p, q], [r, \neg p, s], [q, \neg s]].$$

Here's another example. The following is a list of lists representation of a formula in CNF:

$$[[p, \neg q], [r, \neg s], [], [q, \neg s, q]].$$

Note that one of its clauses is the empty clause. If we write this formula in standard notation (using \perp for the empty clause) we obtain

$$(p \vee \neg q) \wedge (r \vee \neg s) \wedge \perp \wedge (q \vee \neg s \vee q).$$

The list of lists notation for CNF formulas is very natural when discussing resolution.

Now for the key semantic observation concerning formulas in CNF:

For a formula in CNF to be true, all the clauses it contains (that is, all its conjuncts) must be true. Hence if a formula in CNF has the empty clause as one of its conjuncts, it can never be true.

This observation plays a crucial role in the formulation of the resolution method.

Well, now we know what CNF is—but there is an important question we have not addressed: how are we to transform propositional formulas into CNF? We need to be able to do this, for the approach to resolution introduced below assumes that the input formula is in this form.

One way to carry out the transformation to CNF is via the following algorithm. Given an input formula we first convert it into what is called *negation normal form* (NNF). Then, once we've converted the input into NNF, we reach CNF by repeatedly applying the distributive and associative laws. Let's go through this in detail.

A formula is in NNF if \vee and \wedge are the only binary boolean connectives it contains, and every occurrence of a negation symbol is applied to a sentence symbol. (To put it another way, a formula in NNF is built up out of literals using \vee and \wedge as the only binary connectives.) Now, it is easy to convert any propositional formula to NNF—all we need to do is keep applying the following rules:

Rewrite $\neg(\phi \wedge \psi)$ **as** $\neg\phi \vee \neg\psi$

Rewrite $\neg(\phi \vee \psi)$ **as** $\neg\phi \wedge \neg\psi$

Rewrite $\neg(\phi \rightarrow \psi)$ **as** $\phi \wedge \neg\psi$

Rewrite $\phi \rightarrow \psi$ **as** $\neg\phi \vee \psi$

Rewrite $\neg\neg\phi$ **as** ϕ .

The first and second rules (the De Morgan laws) drive negations deeper into the formula (that is towards the sentence symbol level), the third and fourth rules eliminate occurrences of \rightarrow , and the fifth rule eliminates nested negations. It is easy to see that when these rules have been applied to the input formula as often as possible, the only connectives left will be \wedge , \vee and \neg , and all remaining negations will be applied to sentence symbols. In short, repeated application of these rules will convert the input into NNF.

But having reached NNF, how do we get to CNF? As we said above, by repeatedly applying the distributive and associative laws. Here are the required rules:

Rewrite $\theta \vee (\phi \wedge \psi)$ **as** $(\theta \vee \phi) \wedge (\theta \vee \psi)$

Rewrite $(\phi \wedge \psi) \vee \theta$ **as** $(\phi \vee \theta) \wedge (\psi \vee \theta)$

Rewrite $(\phi \wedge \psi) \wedge \theta$ **as** $\theta \wedge (\phi \wedge \psi)$

Rewrite $(\phi \vee \psi) \vee \theta$ **as** $\theta \vee (\phi \vee \psi)$.

The first two rules are the distribution rules: the first distributes \vee over \wedge from the left, the second distributes \vee over \wedge from the right. Note their effect: they drive occurrences of \vee deeper into the formula, and lift occurrences of \wedge —and that's exactly what is needed to get CNF. The last two rules are the associativity rules. Their role is to allow brackets to be moved around so that the distribution rules can be applied.

And that's that. By using this two-stage process (reduction to NNF, followed by applications of the distributive and associative rule) we can convert any propositional formula into CNF. Let's look at an example. We shall convert

$$(\neg p \rightarrow q) \rightarrow (\neg r \rightarrow s)$$

into CNF. The first stage of the process is to put the formula in NNF. So we eliminate the main \rightarrow to obtain:

$$\neg(\neg p \rightarrow q) \vee (\neg r \rightarrow s).$$

We next eliminate the lefthand \rightarrow to obtain:

$$(\neg p \wedge \neg q) \vee (\neg r \rightarrow s).$$

We then eliminate the righthand \rightarrow to obtain:

$$(\neg p \wedge \neg q) \vee (\neg\neg r \vee s).$$

Eliminating the double negation yields:

$$(\neg p \wedge \neg q) \vee (r \vee s).$$

This formula is in NNF, and we can now apply the second distribution law to obtain:

$$(\neg p \vee (r \vee s)) \wedge (\neg q \vee (r \vee s)).$$

We're there—this formula is in CNF. To emphasise this, let's write it in our list of lists notation:

$$[[\neg p, r, s], [\neg q, r, s]].$$

A word of warning. This is the algorithm we shall implement to convert formulas to CNF—but we didn't choose this algorithm because it's the best (it's not)—we chose it because it is the simplest to justify. The algorithm just outlined has the following property: if ψ is the formula in CNF produced by this algorithm from some input formula ϕ , then ϕ and ψ are logically equivalent, that is, satisfied in exactly the same models (the reader is asked to prove this in Exercise 4.4.2). In short, the approach to preprocessing used here converts the input formula into a logically equivalent formula in CNF, and this obviously makes good semantic sense.

However, this algorithm can be extremely slow on some input. This is because sometimes the CNF formula it gives rise to may be a *lot* bigger than the input formula (to put it more technically, there can be an exponential blowup in the size of the input formula). Where does the problem lie? With the distribution laws. Repeated distributions can swiftly create very large formulas indeed, as the reader is asked to show in Exercise 4.4.3. A number of more sophisticated algorithms which avoid the exponential blowup are known. We won't discuss them here, but the reader should know about them, and references to these methods are given in the Notes at the end of the chapter.

Is that all? Not quite—we have to make one final refinement. We said above that the approach to resolution presented below assumes that the input formula is in CNF. Actually, this is not quite right—the method really assumes that the input is in what we shall call *set CNF*. What does this mean? Simply this: a formula is in set CNF if none of its clauses contains a repeated literal, and no clause occurs more than once. For example,

$$[[p, q, r, \neg s], [p, \neg q, p, \neg r]]$$

is in CNF, but it is not in set CNF. Why not? Because the second clause contains two occurrences of p . And

$$[[t, \neg r], [p, q, \neg r], [t, \neg r]]$$

is not in set CNF either, because the clause $[t, \neg r]$ occurs twice. On the other hand,

$$[[p, \neg r, s], [\neg s, \neg p], [q, \neg r]]$$

is in set CNF—no clause contains multiple copies of any literal, and no clause occurs more than once. To put it another way: each inner level list can be viewed as a set of literals, and the outer level list can be viewed as a set of clauses. In what follows we shall often refer to formulas written in set CNF as *clause sets*.

Any formula in CNF can be converted to an equivalent clause set: all we have to do is throw out repeated literals in clauses, and repeated clauses. It should be clear that discarding these kinds of repetitions results in a logically equivalent formula; we are merely jettisoning redundant disjuncts and conjuncts.

In the approach to resolution discussed below, it will be important that we work with formulas in set CNF (that is, clause sets) and not merely CNF. We'll see why later.

Exercise 4.4.1 Convert the *negation* of the following formulas to CNF:

1. $\neg\neg p \rightarrow p$
2. $((p \rightarrow q) \rightarrow p) \rightarrow p$
3. $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$
4. $p \rightarrow (p \wedge (q \vee p))$
5. $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r)).$

Exercise 4.4.2 Show that if ψ is the formula in CNF produced from some input formula ϕ by the algorithm described in the text, then ϕ and ψ are logically equivalent (that is, satisfied in exactly the same models).

Exercise 4.4.3 Try to find a family of formulas which shows that the algorithm described in the text for converting formulas to CNF can lead to an exponential blowup in the size of the input formula.

Exercise 4.4.4 The rules given above for reduction to NNF only cover the \neg , \wedge , \vee , and \rightarrow connectives. Add rules that enable formulas containing \leftrightarrow to be reduced to NNF too. If you allow occurrences of \leftrightarrow in the input, is the length of the output NNF formula always of roughly the same size as the length of the input?

The Resolution Rule

We are now ready to discuss the resolution rule. First some terminology. Suppose we have two clauses, say C and C' , and C contains a positive literal (say r) and C' contains its negation (that is, $\neg r$). Then the

literals r and $\neg r$ are called a *complementary pair* and clauses C and C' are called *complementary clauses*.

The resolution proof method is based around repeated use of the following rule, the *binary resolution rule*:

Given two clauses of the form

$$[p_1, \dots, p_n, r, p_{n+1}, \dots, p_m] \text{ and } [q_1, \dots, q_j, \neg r, q_{j+1}, \dots, q_k]$$

as input, deduce

$$[p_1, \dots, p_n, p_{n+1}, \dots, p_m, q_1, \dots, q_j, q_{j+1}, \dots, q_k]$$

as output.

To spell this out, as input the resolution rule takes two complementary clauses. Complementary clauses must contain at least one pair of complementary literals, and we have written r and $\neg r$ to indicate such a pair. This pair—the *resolvents* as they are usually called—is the complementary pair we use when we apply the resolution rule. And what happens when we apply the rule? Quite simply, we discard the resolvents, and merge what remains of the two clauses. For example, if we apply the rule to the clauses

$$[p, \neg q, r] \text{ and } [s, q, t]$$

we obtain

$$[p, r, s, t].$$

It should be reasonably clear that the resolution rule is semantically sensible. After all, if we know that both $[p, \neg q, r]$ and $[s, q, t]$ are true then (as both of these expressions are disjunctions) we know that at least one of p , $\neg q$ and r must be true, and that at least one of s , q , and t must be true too. But as $\neg q$ and q can't both be true, we can be certain that at least one of p , r , s or t must be true, and this is exactly what $[p, r, s, t]$ (the output of the rule) asserts.

Note that there may well be several ways to apply the resolution rule to the same pair of clauses. For example, consider the clauses $[p, s, \neg t, r]$ and $[q, t, \neg s, r]$. There are two complementary pairs here: s and $\neg s$, and $\neg t$ and t . If we choose s and $\neg s$ as our resolvents, then applying the resolution rule yields

$$[p, \neg t, r, q, t, r].$$

On the other hand, if we choose $\neg t$ and t as our resolvents, then applying the resolution rule yields

$$[p, s, r, q, \neg s, r].$$

It is also worth noting that even if the two clauses input to the binary resolution rule are in set form, this does not guarantee that the

output clause will be too. For consider the example just given. Neither $[p, s, \neg t, r]$ nor $[q, t, \neg s, r]$ contains repeated literals, but $[p, s, r, q, \neg s, r]$ contains a repeated r . As we shall see, for the resolution method described below to function properly, it is essential that we always work with clauses in set CNF. Thus in general we will have to post-process the clauses output from the resolution rule to ensure that they are sets. Let's be a little more precise about this:

If C is a clause, then $\text{set}(C)$ is simply C itself if C is already in set form. Otherwise, $\text{set}(C)$ is the clause C' obtained from C by discarding literals so that each literal in C occurs in C' exactly once.

For example, $\text{set}([p, q],) = [p, q]$ and $\text{set}([p, q, \neg r, p],) = [p, q, \neg r]$. Thus when we say that we have to post-process the output from the resolution rule, we mean that instead of working directly with the clause C that the rule gives to us, we always work with $\text{set}(C)$ instead.

Well, the preliminaries are now out of the way, so it's time to turn to the central question: how do we actually use the resolution rule in theorem proving?

Like the tableau method, the resolution method is a refutation method. That is, just as in the tableau method, if we want to show that ϕ is valid, we give $\neg\phi$ as input, and try to generate a contradiction (which in the setting of resolution means that we try to generate the empty clause). However (unlike the tableau method), resolution is a two stage process. First we have to convert the input formula $\neg\phi$ into set CNF; we'll call the result of this conversion our *clause set*. We then check to see our clause set contains the empty clause, for if it does, we are finished: we have obtained a contradiction and can declare that ϕ has been proved. But if this is not the case, we have to move onto the second phase: the resolution phase proper.

What does this involve? The following:

1. Pick two complementary clauses from our clause set, and two complementary literals r and $\neg r$ in the clauses picked, and apply the resolution rule using r and $\neg r$ as the resolvents (if there are no complementary pairs, halt and declare that ϕ has *not* been proved).
2. If the clause C output by the resolution rule is the empty clause, halt and declare that ϕ has been proved.
3. Otherwise, form $\text{set}(C)$. If $\text{set}(C)$ is already in our clause set, we discard it. Otherwise we add it to our clause set, and go back to Step 1 of the process.

In short, we simply keep cycling through these three steps, applying the

resolution rule and (when it yields some new non-empty clause) adding the output to the clause set. We halt if at any stage the resolution rule yields the empty clause (in which case we say that ϕ has been proved) or if we have applied resolution to all possible resolvents in all complementary pairs (in which case we say that ϕ has *not* been proved).

Let's look at some examples. We'll first use this method to prove $p \rightarrow p$. As we said above, resolution is a refutation method, so we negate this formula, obtaining $\neg(p \rightarrow p)$, and convert this to set CNF. Doing so yields

$$[[p], [\neg p]].$$

Can we refute this? Well, it does not contain the empty clause, so we must enter the resolution phase. We choose a complementary pair of clauses and a pair of resolvents in these clauses (easy—there's only one possible choice!) and apply the binary resolution rule. Resolving $[p]$ with $[\neg p]$ yields $[]$, so we have generated the empty clause and refuted the negated input. We halt and declare that we have proved $p \rightarrow p$.

Here's a second example. We'll use the method to prove

$$(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p).$$

First the preprocessing phase. We negate this formula, and convert it to set CNF. Doing so yields:

$$[[\neg p], [q], [\neg q, p]].$$

Can we refute this? Again, it does not contain the empty clause, so we move onto the resolution phase proper. Let's first resolve $[\neg p]$ against $[\neg q, p]$. Doing so yields $[\neg q]$. This clause is already in set form, and it is not already present in our clause set, so we add it to obtain

$$[[\neg p], [q], [\neg q, p], [\neg q]].$$

Resolving $[\neg q]$ against $[q]$ then yields $[]$, so we halt and declare that $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$ has been proved.

Let's now prove $(p \vee p) \rightarrow (p \wedge p)$. This example will make clear why we have to work in set CNF. We'll do a little experiment: we'll negate the formula, but instead of converting into set CNF, we'll only convert into ordinary CNF. If we do this we obtain

$$[[p, p], [\neg p, \neg p]].$$

But now consider what happens when we apply the resolution rule: we obtain

$$[[p, \neg p][p, p], [\neg p, \neg p]].$$

And it is not hard to see that whatever we do we are never going to generate the empty clause—all we can do is generate $[p, \neg p]$, $[p, p]$, and

$[\neg p, \neg p]$ clauses. And this is due to the fact that we did not take care to convert the original clauses into sets. For if we had done this, our original clause set would have been

$$[[p], [\neg p]],$$

and we could then have generated the empty clause in a single resolution step.

Well, that's the approach to propositional resolution that we will implement. But before doing so, let's discuss another question: does the resolution process just described make semantic sense? We have already given an informal justification of the resolution rule itself, but the use of this rule is only a part (albeit a central part) of the resolution method. Does everything really hang together the way it should? Yes, it does, for everything we do to the negation of the input formula *preserves satisfiability in a model*.

An operation which takes as input a propositional formula ψ and returns as output a propositional formula ψ' preserves satisfiability in a model if whenever the input ψ is satisfied in some model M then so is the output ψ' too. (Of course, as we are working with propositional logic, in this chapter models are simply assignments of truth values to sentence symbols; see Appendix B.)

Now, it is quite easy to see that everything we do to the negation of the input formula in the course of a resolution proof preserves satisfiability in a model. Let's think this through. Suppose we are trying to show that ϕ is valid. We negate ϕ , forming $\neg\phi$, and from then on everything we do preserves satisfiability. We first convert $\neg\phi$ into set CNF. This part of the process certainly preserves satisfiability. In fact, the way we did it, it preserves even more semantic structure: the set CNF formula we obtain is actually logically equivalent to $\neg\phi$ (that is, it is satisfied in exactly the same models). So what about the second phase, the applications of the resolution rule? Do these rule applications preserve satisfiability in a model too?

The resolution rule takes as input two clauses

$$[p_1, \dots, p_n, r, p_{n+1}, \dots, p_m] \text{ and } [q_1, \dots, q_j, \neg r, q_{j+1}, \dots, q_k]$$

and returns as output a clause

$$[p_1, \dots, p_n, p_{n+1}, \dots, p_m, q_1, \dots, q_j, q_{j+1}, \dots, q_k]$$

(the resolvents are r and $\neg r$). Now, suppose that both input clauses are satisfied in some model M . This means that at least one literal in each of the two input clauses must be true in M . But only one of r and $\neg r$ can be true in M , so at least one other literal (either from the first

literal or from the second) must be true in M as well. But all literals from the input clauses (apart from the resolvents r and $\neg r$) are in the output clause, hence at least one literal in the output is true in M , and hence the output (being a disjunction of literals) is true in M too.

Thus everything we do to $\neg\phi$ preserves satisfiability in a model. It follows that:

If $\neg\phi$ is satisfied in some model M , then all clause sets that we obtain in the course of carrying out resolution must be true in M too.

Now consider what this must mean if we generate the empty clause during the resolution process. A clause set containing the empty clause cannot be satisfied in any model M whatsoever. Hence, as the resolution process preserves satisfiability in a model, it follows that $\neg\phi$ cannot be satisfied in any model M either. From this it follows that the input formula ϕ must be satisfied in *all* models; that is, ϕ is indeed valid.

In short, the resolution procedure just described is a purely syntactic method that makes good semantic sense—it is a genuine proof method. So, parallelling what we did in the tableau case, we shall make the following definition:

A propositional formula ϕ is resolution-provable if and only if it is possible to apply the resolution method described above to refute $\neg\phi$ (that is, generate a clause set containing the empty clause). We use the notation $\vdash_r \phi$ to indicate that ϕ is resolution-provable.

Exercise 4.4.5 Give resolution proofs of the following formulas (recall that you were asked to convert the negations of these formulas into CNF in Exercise 4.4.1):

1. $\neg\neg p \rightarrow p$
2. $((p \rightarrow q) \rightarrow p) \rightarrow p$
3. $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$
4. $p \rightarrow (p \wedge (q \vee p))$
5. $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r)).$

Compare these proofs with the tableau proofs requested in Exercise 4.2.1.

Exercise 4.4.6 Suppose we are carrying out a resolution proof with a clause set that contains the clause $[r, p]$ and also the clause $[p]$. Can you explain why the clause $[r, p]$ is redundant? (Hint: think semantically.) Can you extend this observation into a simple strategy for eliminating certain types of redundant clauses?

Exercise 4.4.7 Suppose we are trying to prove some formula, and that the set CNF representation of its negation contains a clause in which some literal occurs both positively and negatively (that is, it contains a clause of the

form $[\dots, p, \dots, \neg p \dots]$). Explain why such clauses are redundant and can be eliminated (Hint: think semantically).

4.5 Implementing Propositional Resolution

We are now ready to implement propositional resolution. Unsurprisingly, our work falls into two phases: first we must implement the reduction to set CNF, and then we must implement the resolution phase proper.

Converting formulas to set CNF

Here's the high level predicate `cnf/2` which carries out the conversion:

```
cnf(Formula, SetCNF) :-  
    nnf(Formula, NNF),  
    nnf2cnf([[NNF]], [], CNF),  
    setCnf(CNF, SetCNF).
```

The input to this predicate, `Formula`, is a propositional formula written in our standard (Prolog) logical notation. For example:

```
imp(and(p, imp(q, r)), s).
```

The output, `SetCNF`, is the result written in (Prolog) list of lists notation, of converting the input into set CNF. For example, the input just given would convert to:

```
[[s, not(p), q], [s, not(p), not(r)]].
```

As the code makes clear, `cnf/2` performs the conversion in three steps. First it converts the input formula into a formula in NNF. It then converts the NNF formula into one in CNF. Finally, it converts the CNF into the output set CNF formula by throwing away redundant literals and clauses. Let's deal with each part in turn.

Recall that a formula in NNF only has as connectives \wedge , \vee and \neg , and that all occurrences of \neg must be next to sentence symbols. So, as we discussed above, we convert an arbitrary propositional formula into NNF by using the De Morgan laws to push the negations deeper into the formula, by re-expressing implications and negated implications in terms of \wedge , \vee and \neg , and by eliminating double negations. It is easy to capture this rewriting process as a recursive Prolog predicate. First of all, here are the clauses that push the negations deeper into the formula:

```
nnf(not(and(F1, F2)), or(N1, N2)) :-  
    nnf(not(F1), N1),  
    nnf(not(F2), N2).
```

```

nnf(and(F1,F2),and(N1,N2)):-  

  nnf(F1,N1),  

  nnf(F2,N2).  
  

nnf(not(or(F1,F2)),and(N1,N2)):-  

  nnf(not(F1),N1),  

  nnf(not(F2),N2).  
  

nnf(or(F1,F2),or(N1,N2)):-  

  nnf(F1,N1),  

  nnf(F2,N2).

```

The first and the third of these clauses capture the effect of the De Morgan laws. The second and the fourth clauses permit the recursion to work its way down to subformulas in the cases where no negations are encountered.

The clauses for eliminating negated implications, implications, and double negations are straightforward:

```

nnf(not(imp(F1,F2)),and(N1,N2)):-  

  nnf(F1,N1),  

  nnf(not(F2),N2).  
  

nnf(imp(F1,F2),or(N1,N2)):-  

  nnf(not(F1),N1),  

  nnf(F2,N2).  
  

nnf(not(not(F1)),N1):-  

  nnf(F1,N1).

```

Finally, here's what happens when we reach the level of literals:

```

nnf(F1,F1):-  

  literal(F1).  
  

literal(not(F)):- atomic(F).  

literal(F):- atomic(F).

```

Now for the next step—we need to convert our NNF formula into one in CNF by repeatedly applying the distributive and associative laws. Actually, at this stage we do something else as well. As we said above, the input to `cnf/2` is a formula in our standard (Prolog) notation whereas the output is in a list of lists notation. Where do we shift from one representation to another? Here at the second stage. Let's take a look at the code:

```
nnf2cnf([],_,[]).
```

```

nnf2cnf([[]|Tcon],Lit,[Lit|NewTcon]) :-
  !,
  nnf2cnf(Tcon,[],NewTcon).

nnf2cnf([[and(F1,F2)|Tdis]|Tcon],Lits,Output) :-
  !,
  appendLists(Tdis,Lits,Full),
  nnf2cnf([[F1|Full],[F2|Full]|Tcon],[],Output).

nnf2cnf([[or(F1,F2)|Tdis]|Tcon],Lits,Output) :-
  !,
  nnf2cnf([[F1,F2|Tdis]|Tcon],Lits,Output).

nnf2cnf([[Lit|Tdis]|Tcon],Lits,Output) :-
  nnf2cnf([Tdis|Tcon],[Lit|Lits],Output).

```

This is the trickiest part of the CNF conversion code—probably the best way to get to grips with it is to try out Exercise 4.5.2 right away.

Once we have the actual CNF, the hard work is done. All that remains is to strip out duplicate literals and clauses. We do this with the help of an accumulator and the `removeDuplicates/2` predicate defined in the file `comsemPredicates.pl`. Here's what we do:

```

setCnf(Cnf1,Cnf2) :-
  setCnf(Cnf1,[],Cnf2).

setCnf([X1|L1],L2,L3) :-
  removeDuplicates(X1,X2),
  setCnf(L1,[X2|L2],L3).

setCnf([],Cnf1,Cnf2) :-
  removeDuplicates(Cnf1,Cnf2).

```

That is, the first clause initialises the accumulator to the empty list, the second clause strips duplicate literals from each clause and places the resulting set clauses on the accumulator, and the third then weeds duplicate clauses from the accumulator list and returns the output.

We have included a little test suite for trying out the converter. The code that runs the test suite is:

```

cnfTestSuite :-
  formulaClause(Formula,Cnf),
  format('`nInput formula: ~p',[Formula]),
  format('`nKnown cnf: ~p',[Cnf]),
  cnf(Formula,CNF),
  format('`nComputed cnf: ~p`n',[CNF]),
  fail.

```

```
cnfTestSuite.
```

The test suite itself is in file `cnfTestSuite.pl`.

Exercise 4.5.1 Can Prolog cuts safely be added to the definition of `nnf/2`? If they can be, add them.

Exercise 4.5.2 Add the following clause to the start of `nnf2cnf/3`

```
nnf2cnf(X,Y,Z) :- nl, write(X), write(Y), write(Z), fail.
```

Because this clause always fails, it has no effect on the correctness of `nnf2cnf/3`. But before failing it writes out the arguments given to `nnf2cnf/3`, thus enabling you to visualise what is going on. For example, if you make the query

```
cnf(or(p, and(q, r)), C).
```

you will receive a listing of exactly how `nnf2cnf/3` was called by `cnf/2` during the conversion process. Try out the CNF converter with this extra clause in place.

Exercise 4.5.3 The third clause of `nnf2cnf/3` deals with conjunctions by duplicating disjuncts (thus applying the distribution rule); the duplicates are given as the Prolog variable `Full`. This is not efficient—all the operations that have to be performed for the information in `Full` will be doubled (and doubled again if the distributive rule is fired again, and so on). Deal with this by extending `nnf2cnf/3` with a memory feature that records all its operations, and before applying a new rule, first checks whether it has already done these operations. Use a dynamic predicate and `assert/1` and `retract/1` to implement this in Prolog.

Exercise 4.5.4 Extend the definition of `nnf/2` to deal with the bi-implication connective \leftrightarrow (use, say, `bimp/2` as the Prolog notation for this connective).

Performing Resolution

With the preliminaries out of the way, we are ready to implement the predicates that do the real work of propositional resolution. In fact, the required code is fairly straightforward: all we have to do is ensure that we apply the resolution rule systematically, that we keep working with sets of clauses and sets of literals, and that we detect the empty clause if we generate it.

Here is our main predicate, `rprove/1`:

```
rprove(Formula) :-  
  cnf(not(Formula), CNF),  
  refute(CNF).
```

This predicate takes as its argument the formula we are trying to prove. The formula is input in our standard (Prolog) logical notation, and is immediately negated (as resolution is a refutation method) and given to `cnf/2`. This converts it to set CNF in list of lists notation. We then try to refute this clause set by giving it as an argument to `refute/2`.

As you can see, `refute/2` encapsulates the entire resolution process into a few lines of code:

```
refute(C):-  
    memberList([],C).  
  
refute(C):-  
    \+ memberList([],C),  
    resolveList(C,[],Output),  
    unionSets(Output,C,NewC),  
    \+ NewC = C,  
    refute(NewC).
```

Let's go through this carefully, as this is the most important predicate in the program. The first Prolog clause checks whether the empty clause is in the input clause set `C`: if it is, we have succeeded in proving the input formula. If the empty clause is not present, we enter the resolution phase. That is, we systematically start trying all possible ways of resolving the clauses in `C` with each other; the call to `resolveList/3` does this for us. The second argument of `resolveList/3` is an accumulator (initialised to the empty list) and the output of all these resolutions is returned as the third argument `Output`, a list of clauses; the lower level predicates called by `resolveList/3` guarantee that all the clauses in `Output` are returned in set form. We then form `NewC` by adding the clauses returned in `Output` to the clauses in `C`. We do this with the help of the predicate `unionSets/3`, which is defined in `comsemPredicates.pl`, to ensure that `NewC` is in set CNF. Finally, we check whether this last round of resolutions has produced any new clauses: that is, we check whether `NewC` is different from `C`. If new clauses have been produced, we recursively call `refute/1` with `NewC` as its argument. On the other hand, if there are no new clauses then `refute/1` fails, and the input formula cannot be proved.

Only three more predicates are required to make this all work, and they are all quite simple. Let's first look at `resolveList/3`, which is called by `refute/1` to systematically try out all possible resolutions on the clauses in an input clause set:

```
resolveList([],Acc,Acc).  
  
resolveList([Clause|List],Acc1,Acc3):-
```

```
resolveClauseList(List, Clause, Acc1, Acc2),
  resolveList(List, Acc2, Acc3).
```

The first argument of this predicate is a list of clauses. The second clause in the definition recursively takes the clause at the head of the list, and tries resolving it against all the remaining clauses on the list. The results are stored in an accumulator. The first clause in the definition halts the accumulation process when the empty set is reached (that is, when all the clauses in the input have undergone the resolution process).

So how do we resolve a clause against a list of clauses? As follows:

```
resolveClauseList([], _, Acc, Acc).
```

```
resolveClauseList([H|L], Clause, Acc1, Acc3) :-
  resolve(Clause, H, Result),
  unionSets([Result], Acc1, Acc2),
  resolveClauseList(L, Clause, Acc2, Acc3).
```

```
resolveClauseList([H|L], Clause, Acc1, Acc2) :-
  \+ resolve(Clause, H, _),
  resolveClauseList(L, Clause, Acc1, Acc2).
```

Again, this predicate makes use of accumulators, and the task of the first clause of the definition is to halt the accumulation process when the empty list is reached. The second and third Prolog clause are jointly responsible for resolving `Clause` against all the clauses in the list of clause `[H|L]`. The second predicate handles the case that arises when it is possible to successfully call `resolve/3` to resolve `Clause` against a clause on the list: it stores the result in an accumulator, and then recursively carries on trying to resolve `Clause` against the next item on the list. The third clause deals with the case that occurs when `resolve/3` does *not* return a result (that is, when it is not possible to resolve `Clause` against a clause on the list). It simply recursively carries on trying to resolve `Clause` against the next item on the list.

All that we now need is predicate which can resolve two clauses together to produce a new clause. Here is what we use:

```
resolve(Clause1, Clause2, NewClause) :-
  selectFromList(Lit, Clause1, Temp1),
  selectFromList(not(Lit), Clause2, Temp2),
  unionSets(Temp1, Temp2, NewClause).

resolve(Clause1, Clause2, NewClause) :-
  selectFromList(not(Lit), Clause1, Temp1),
```

```
selectFromList(Lit,Clause2,Temp2),
unionSets(Temp1,Temp2,NewClause).
```

This is straightforward: we try to find a pair of complementary literals in the two input clauses, we then remove these literals (using the `selectFromList/3` predicate defined in `comsemPredicates.pl`) and combine the remaining information. Because we carry out the combination using `unionSets/3`, we ensure that the output clause `NewClause` is in set form.

Well, that's the propositional resolution prover. It's time to start playing with it. Like the tableau prover, this can also be tried out on the test suite that you will find in the file `propTestSuite.pl`. To run the test suite, simply load the file `propResolution.pl` and enter the command `rproveTestSuite`.

Exercise 4.5.5 Try the prover out on some simple examples. Make sure you understand what is happening at each step. A nice way to do this is to add the following code as a new first clause for the predicate `refute/1`:

```
refute(C) :- nl, write(C), nl, fail.
```

Because this new clause always fails, it does not effect the correctness of the predicate. However, before it fails it will write out the current clause set, enabling you to follow the prover's progress easily.

Exercise 4.5.6 Try to find examples of theorems which the resolution prover can't handle, or can't handle fast. That is, try to find propositional formulas which you know to be valid but which the resolution prover either won't halt on, or takes a long time to halt. (The last two formulas in the test suite are like this, but try to find your own examples before looking at these.)

4.6 Theoretical Remarks

We conclude our discussion of propositional inference with some theoretical remarks. First, we shall introduce three concepts that every computational semanticist should have at least a nodding acquaintance with: soundness, completeness, and decidability. Second, we shall link these concepts to the consistency and informativity checking tasks defined in Chapter 1. Finally, we make some remarks on the computational complexity of propositional theorem proving.

Soundness

Both the tableau system and the resolution system discussed in this chapter are *sound* proof systems for propositional logic. What does this mean?

Programs for propositional resolution

`propResolution.pl`

The file that contains all the predicates for the implementation of our resolution-based theorem prover.

`cnf.pl`

File with the definitions for translating to nnf, cnf, and set cnf.

`propTestsuite.pl`

Testsuite with problems to test both propositional theorem provers.

`comsemPredicates.pl`

Contains some auxiliary predicates for list processing.

As we remarked at the start of the chapter, although syntactic proof methods only *use* syntactic information, they have to be *justifiable* in semantic terms. That is, when someone proposes a new proof method, we need guarantees that the proposal is semantically sensible.

First and foremost, a proof system must be sound. Soundness is essentially a ‘no garbage’ condition: if the proof system says some formula is provable, then that formula should be valid. In a nutshell, sound proof methods are faithful to the semantic concept of validity.

Our tableau system is sound. That is, for any propositional formula ϕ we have that

$$\text{if } \vdash_t \phi \text{ then } \models \phi.$$

So tableau proofs will never lead us astray—and this shouldn’t really come as a surprise. After all, we developed our tableau expansion rules by thinking in overtly semantic terms. For example, we asked “How do we go about making a conjunction true?” and gave the (obviously sensible) answer “By making both conjuncts true”, and it is clear that all our tableau expansion rules are semantically sensible. Now, the tableau method is a refutation method: to prove ϕ we form $F\phi$ and apply the appropriate tableau rules. If all branches of this tableau turn out to be closed (that is, if each branch ends up containing a pair of signed formulas of the form $T\psi$ and $F\psi$) then there can be no way to falsify the initial formula: for otherwise (using the fact that the tableau rules are semantically sensible) it would follow that there were models in which

some formula ψ was both true and false, which is impossible. It follows that the original formula ϕ must indeed be a validity.

Our resolution system is also sound. That is, for any propositional formula ϕ we have that

$$\text{if } \vdash_r \phi \text{ then } \models \phi.$$

We've pretty much proved this already. Resolution is a refutation method, so to prove ϕ we convert $\neg\phi$ to set CNF and repeatedly apply the binary resolution rule. Now, as we discussed at the end of Section 4.4, both the conversion to set CNF and applications of the binary resolution rule preserve satisfiability in a model. Hence, if we succeed in deriving the empty clause from $\neg\phi$, it follows that $\neg\phi$ *cannot* have a model, from which it follows that ϕ must indeed be a validity.

Completeness

What about completeness? This is a more interesting demand: a complete proof system is one which is capable of proving *all* valid formulas. That is, if ϕ is any valid formula whatsoever, then it must be possible to give a proof of ϕ .

Our tableau system is complete. That is, for any propositional formula ϕ we have that

$$\text{if } \models \phi \text{ then } \vdash_t \phi.$$

This means that no propositional validity lies beyond the reach of the tableau proof method: if a formula ϕ is valid, then by forming the initial tableau $F\phi$ and applying tableau rules it is possible to construct a closed tableau. Incidentally, it is easy to give examples of tableau systems that are sound but *not* complete—simply throw away some of the expansion rules! For example, if we discard the rule $F\neg$, we still have a *sound* tableau system, but we don't have enough power left to prove all propositional validities, as we can no longer have the rule we need for coping with any negated formulas that we need to falsify. Note that because the tableau method is both sound and complete we have that:

$$\vdash_t \phi \text{ iff } \models \phi.$$

That is, because our tableau system is both sound and complete there is a perfect match between the syntactic concept of tableau-provability and the semantic concept of propositional validity.

Our resolution system is also complete. That is, for any propositional formula ϕ we have that

$$\text{if } \models \phi \text{ then } \vdash_r \phi.$$

This means that no propositional validity lies beyond the reach of the

resolution proof method: if a formula ϕ is valid, then by converting it to set CNF and applying binary resolution it is possible to generate the empty clause. It is easy to give an example of a resolution system that is sound but not complete—simply change the method described in the text so that we only convert into ordinary CNF, not set CNF. As we saw in Section 4.4, if we do this we can't prove all validities (for example, we can't prove $(p \vee p) \rightarrow (p \wedge p)$). Note that because the resolution method is both sound and complete we have that:

$$\vdash_r \phi \text{ iff } \models \phi.$$

That is, as with the tableau method, there is a perfect match between a syntactic concept (this time, resolution-provability) and the semantic concept of propositional validity.

It is not particularly difficult to prove the completeness of either the tableau or resolution systems we have discussed. There are many clear and instructive completeness proofs in the literature, and we encourage mathematically inclined readers to consult the references cited in the Notes for further information.

Decidability

Soundness and completeness are properties of proof systems—we talked of the soundness and completeness of our tableau and resolution system. In a sense, decidability is a more abstract property, for it is not a property of any particular proof system, rather it is a property of the set of valid formulas. Here's the key question: is it possible, at least in principle (that is, ignoring practical constraints on the amount of memory and processing time available) to devise an algorithm that will take a propositional formula as input, and halt after a finite number of steps, and correctly tell us whether the input formula is valid or not? If it is possible to devise such an algorithm, then propositional validity is *decidable*. If it is not possible to devise such an algorithm, then propositional validity is *undecidable*. (Actually, we are usually more easy going with our terminology, and talk about propositional logic being decidable—but when we say things like this we mean that the question of determining whether an arbitrary formula of propositional logic is valid is algorithmically solvable.)

Well then—is propositional validity decidable? Yes, for it is not hard to see that both the tableau and the resolution methods discussed in the text are algorithms for determining the validity of propositional formulas. Now, as is clear from our discussions of soundness and completeness, both the tableau and the resolution methods are capable of delivering *correct* verdicts about validity (after all, soundness plus com-

pleteness basically amounts to an ironclad guarantee of correctness). So, to establish that they are algorithms in the full sense of the word, it only remains to show that both methods halt with their verdict after a finite number of steps, no matter what formula we give them as input. Do they?

Well, it is not hard to see that the propositional tableau method is (at least in principle) guaranteed to halt on all possible inputs. To see why, note that every tableau expansion rule takes a signed formula and returns a *finite* number of signed formulas (in fact, no rule returns more than four signed formula). Moreover, crucially, each of the formulas returned contains fewer connectives than the input formula. Thus, as we never have to apply a rule to the same formula twice, after every expansion the collection of unexpanded formulas remains finite, and moreover, any new unexpanded formulas we obtain as a result of rule applications are simpler: they contain fewer connectives. Thus—no matter which sequence of expansions we choose to make—we will achieve rule-saturation after a finite number of steps, for eventually we will produce output containing no connectives at all. Making this argument fully precise is not difficult and Exercise 4.6.2 asks the reader to spell out the details. The upshot is this: the propositional tableau method is (in principle) guaranteed to halt on any input and (by soundness and completeness) guaranteed to deliver a correct verdict on the validity or non-validity of the input formula. Hence the tableau method is an algorithm for deciding propositional validity.

The propositional resolution method is also (at least, in principle) guaranteed to halt on all possible inputs. To see why, first note that conversion to CNF is a well-defined process that is guaranteed to terminate after a finite number of steps. Furthermore, we cannot go on applying the binary resolution rule forever. Why not? Essentially because the rule manipulates *sets*. We start with some fixed set of literals, and the binary resolution rule does not produce new symbols, it simply rearranges the symbols we started with into new clauses. The worst that can happen is that the process leads to some huge clauses—but it can't build any clause containing more than the number of literals with which we started. Nor can the resolution rule build an infinite number of clauses: as we work with sets of clauses, no repetition is allowed. So eventually the propositional resolution method must halt (we ask the reader to flesh out this argument sketch in Exercise 4.6.3). The upshot is this: the propositional resolution method is (in principle) guaranteed to halt on any input and (by soundness and completeness) guaranteed to deliver a correct verdict on the validity or non-validity of the input formula. Hence resolution is another algorithm for deciding

propositional validity.

Two comments. First, as we remarked at the beginning of the chapter, there is another well known algorithm for deciding propositional logic: the truth table method is a direct (though often cumbersome) way of determining whether a propositional formula is valid or not. Second, note that we hedged our claim that the tableau and resolution methods halted with the words “in principle”. As we shall shortly see, while both methods do indeed terminate “in principle”, what happens in practice is often another matter. But before discussing this, let us link the concepts we have been discussing to the consistency and informativity checking tasks.

Consistency and Informativity Checking

We now know that propositional logic is decidable, and that both the tableau and resolution methods are sound and complete for propositional validity. What are the ramifications of this for the consistency and informativity checking tasks? In essence this: it tells us that theorem proving constitutes (in principle) a complete computational solution to these tasks for propositional logic.

This should be fairly clear. When we say that a propositional formula ψ is uninformative with respect to propositional formulas ϕ_1, \dots, ϕ_n , we mean that

$$\phi_1, \dots, \phi_n \models \psi,$$

and when we say that ψ is inconsistent with respect to ϕ_1, \dots, ϕ_n we mean that

$$\phi_1, \dots, \phi_n \models \neg\psi.$$

Both tasks can be reformulated (by appealing to the Semantic Deduction Theorem) in terms of the validity of single formulas: ψ is uninformative with respect to ϕ_1, \dots, ϕ_n means that $\phi_1 \wedge \dots \wedge \phi_n \rightarrow \psi$ is valid, and saying that ψ is inconsistent with respect to ϕ_1, \dots, ϕ_n means that $\phi_1 \wedge \dots \wedge \phi_n \rightarrow \neg\psi$ is valid.

But then to solve either task computationally, all we need is a theorem prover for propositional logic. For assume we choose a semantically correct theorem prover (that is, one that embodies a sound and complete proof method) and assume the theorem prover has been correctly implemented (and in particular, that in principle it terminates on all input). Then, as testing for informativity or consistency simply amounts to testing the validity of certain formulas, the theorem prover is a computational solution to both problems.

Now, this observation is not all that interesting in its own right, for it's clear that propositional logic is not enough for natural language

semantics. Rather, we mention it because of the sharp contrast with what we shall learn in the following chapter: in the setting of full first-order logic, theorem proving does *not* give us a full solution to the consistency and informativity checking tasks. As we shall see, it is possible to extend both the tableau and resolution provers to (sound and complete) systems for first-order logic, but while both provers will be correct, they *won't* offer a full computational solution to the consistency and informativity checking tasks. Why not? Because first-order logic is undecidable, so neither prover (and indeed, no prover at all, no matter how good) will terminate on all input. This is a fundamental fact of logical life, and we'll have to learn to live with it.

Computational Complexity

Is the tableau method an efficient way of determining propositional validity? Is the resolution method? The honest answer is this: nobody knows for sure whether there is *any* efficient way of performing propositional inference, and the general consensus is that methods that perform well on all input are unlikely to exist. To put it another way, it is widely believed that the task of determining propositional validity, though decidable, is intrinsically (extremely) difficult.

If you are new to proof theory and theorem proving it may be hard to believe that determining propositional validity is really so hard. For a start, it is easy to find formulas which show that tableau or resolution rules can be a lot better than blindly filling out a complete truth table; for example, the truth table for

$$(q \rightarrow (r \vee (p \wedge \neg t \rightarrow \neg \neg s))) \vee \neg(q \rightarrow (r \vee (p \wedge \neg t \rightarrow \neg \neg s))).$$

has 32 rows, whereas the tableau method gets this example right in a single step. And it is not hard to devise simple heuristics for constructing more compact tableau and resolution proofs. For example, in tableau proofs it is a good idea to apply as many conjunctive expansion rules as possible before applying disjunctive expansion rules, and in resolution proofs it can be useful to jettison redundant clauses (recall Exercises 4.4.6 and 4.4.7). Such considerations may mislead you into thinking that determining propositional validity really isn't all that demanding.

It is important that you lose this impression as swiftly as possible: it is illusory. There are some formulas for which it is extremely hard to prove validity. A case in point are the formulas that describe the *pigeon hole principle*. Stated informally, this principle tells us that if you have fewer pigeon holes than pigeons, and you put every pigeon in a pigeon hole, then there must be at least one pigeon hole with more

than one pigeon. (In more mathematical language, it tells us that if we have a function f from a set X containing $n + 1$ elements to a set Y containing n elements, then there must exist distinct elements x and x' of X such that $f(x) = f(x')$.) Obvious though this principle is, our theorem provers find it extremely hard to prove.

Our propositional test suite contains three pigeon hole principle formulas: the formula for three pigeons and two holes, the formula for four pigeons and three holes, and the formula for five pigeons and four holes. Both the tableau and resolution provers successfully prove the formula for three pigeons and two holes—but if you add the extra lines suggested above for printing out the tableau and clause sets that the provers generate in the course of proofs (recall Exercises 4.3.1 and 4.5.5), then you are in for a shock: the tableau generated is very large, as is the clause set that the resolution prover needs to generate before it finds an empty clause. If we try out the four pigeons and three holes problem, neither prover succeeds: the tableau prover simply does not terminate (at least, we've never had the patience to wait for it to terminate) and the resolution prover goes to sleep for a long time and then reports a stack overflow. And as for five pigeons and four holes...

Now, this is partly due to the inefficiency of our implementations. Both provers were written to illustrate basic principles of theorem proving and neither can lay the slightest claim to sophistication. But still, even if we were working with the most sophisticated theorem prover available, eventually it would fail. Perhaps all we'd need to do is work with a formula for more pigeons and pigeon holes, or perhaps we'd need to try formulas from another hard class (and plenty of hard classes of formulas are known) but *every* propositional theorem prover will eventually be killed by some input.

Why is that? Because determining propositional validity is the classic *co-NP complete problem*. (The problem of determining the *satisfiability* of a propositional formula—indeed, even of a propositional formula in CNF—is the classic example of an *NP-complete problem*. Determining propositional validity is what complexity theorists call the *complementary* problem to determining propositional satisfiability, hence the terminology co-NP complete.) Now, co-NP complete problems are decidable, but it is widely believed that no matter what method we use, on some input they will require at least 2^n steps to solve, where n is the number of symbols in the input. This means we are facing a combinatorial explosion, for 2^n can be astronomical in size even for small values of n . Both the resolution and the tableau method have been shown to require this many steps on some input. And most complexity theorists believe that this is not an accident: it is standardly conjectured that

there is *no* algorithm for determining propositional validity that runs efficiently (that is, in polynomial time) on all input, though nobody has succeeded in proving this.

Exercise 4.6.1 Is it true that for all propositional formulas ϕ we have that $\vdash_r \phi$ if and only if $\vdash_t \phi$? Explain.

Exercise 4.6.2 Spell out in detail the argument sketched in the text that the propositional tableau method must terminate on all input.

Exercise 4.6.3 Spell out in detail the argument sketched in the text that the propositional resolution method must terminate on all input.

Notes

Proof theory is a rich and fascinating subject. The reader can gain a good overview of various types of proof systems (such as natural deduction systems, sequent calculi, axiomatic systems, tableau systems, and resolution) and how they relate to each other, by consulting Sundholm (1983). Another useful, more computationally oriented, overview of proof theory is Gallier (1986).

For tableau systems, the classic source is Smullyan (1995); our discussion of signed tableau is based on Smullyan's treatment. A good treatment of *unsigned* tableau can be found in Bell and Machover (1977); the reader interested in finding out more about the underlying technicalities (including how to prove that the propositional tableau method is complete) will find all that is needed in these two books. Our implementation of propositional tableau was heavily influenced by the implementation of unsigned tableau given in Fitting (1996). The reader will find it instructive to compare the two implementations (apart from anything else, it's a nice way of finding out about unsigned tableau).

Our discussion of propositional resolution was loosely based on the presentation in Leitsch (1997). We attempted to keep our discussion informal and example driven—Leitsch's account, on the other hand, is extremely precise, and the reader who wants a solid introduction to resolution should look at this book. This is also a good source for finding out more about reductions to CNF. The algorithm we presented in the text is not the best, merely the easiest to understand. Leitsch discusses both this basic algorithm and a cleverer approach that avoids the risk of exponential blowup. Another interesting introduction to propositional resolution can be found in Fitting (1996). Fitting presents resolution in a rather different way: instead of viewing it as a two-step process (the standard approach) he interleaves the reduction to normal form with applications of the resolution rule.

The analysis of the complexity of proof methods is an active field of research. Perhaps the best starting point is Urquhart (1995). This contains a good discussion of why the tableau method sometimes leads to combinatorial explosion. Another interesting (and very readable) paper is D'Agostino (1992), which shows that in certain cases the tableau method performs *worse* than the truth table method. In the text we mentioned that pigeon hole principle problems can be extremely hard; such formulas were introduced in Haken (1985). For a discussion of NP complete and co-NP complete problems, see Garey and Johnson (1979) or Papadimitriou (1994).

There is currently a great deal of interest in propositional theorem proving, albeit in a disguised form: SAT solving. The SAT problem is this: given a propositional formula in CNF, does it have a model or not? An amazing variety of tasks can be encoded as SAT problems (in essence, because it is an NP-complete problem). That is, it is often possible to encode an important practical task (about task scheduling, say) as a formula of propositional logic, and then to solve the problem by giving the formula to a SAT solver.

Nowadays there are two main types of SAT solvers: complete ones and incomplete ones. Actually, we've already seen a complete SAT solver: our resolution prover. Recall that to prove a formula ϕ , we formed $\neg\phi$, converted it to CNF, and then handed it to our prover. The task of the resolution component was to check whether the CNF formula could be satisfied or not—we declared that ϕ was proved if it could *not*. In this way, any complete SAT solver can be regarded as a theorem prover for propositional logic.

So complete SAT solvers have much in common with ideas we have already met. But one difference should be mentioned. Current state-of-the-art complete SAT solvers usually make use of an algorithm, the DPLL procedure (the Davis, Putnam, Logemann and Loveland procedure), that is rather different from the resolution method given in the text. The DPLL procedure revolves around using “unit resolution”, that is, resolution between an atom and a disjunctive clause; it was first defined in Davis et al. (1962). Be warned that the DPLL procedure is sometimes referred to as the Davis Putnam algorithm, but in fact the original Davis Putnam algorithm, defined in Davis and Putnam (1960), is quite different.

Incomplete SAT solvers, by way of contrast, make an initial guess of the truth values of the atomic formulas, and then use simple statistical techniques to refine that guess until a satisfying assignment is found (see, for example, Gu (1992) and Selman et al. (1992)). They are not complete because they are not systematic, and cannot report

unsatisfiability (if the formula is unsatisfiable they may not halt, even though propositional logic is decidable). The advantage of incomplete SAT solvers is that they can handle formulas containing an extremely large number of propositional symbols—and if the formula is satisfiable, they will probably be able to demonstrate this.

We mention SAT solvers here for two reasons. First, it is our belief that they are likely to play a useful role in computational semantics, just as they have in other domains. Secondly, in the following chapter we shall make use of a relatively new technology called model building. Some first-order model builders work by converting a first-order formula to a propositional one (this can be done when working with finite models) and then making use of a SAT solver. For a wide-ranging discussion of the SAT problem, see Cook and Mitchell (1997).

Finally, although it takes us far from the concerns of the present book, it is worth noting that proof theory arguably has deeper connections with natural language semantics than our discussion in this chapter might suggest. There is an interesting philosophical tradition which claims that meaning shouldn't be explained in terms of truth conditions, but in terms of assertability conditions. (Roughly speaking, on this view the meaning of an utterance is the reason we have for holding it, not the situations in which it is true.) Semanticists in this tradition tend to regard proofs as the primary semantic objects. At first sight this may appear to be a rather strange view, but it has a lot to recommend it. A good introduction to this line of thought is Sundholm (1986). Moreover, as Ranta (1994) demonstrates, the approach is of relevance to computational semantics.

First-Order Inference

In this chapter we continue our discussion of inference, but we move from the (relatively) simple setting of propositional logic to full first-order logic. This step is far from trivial. For a start, the consistency and informativity checking tasks for first-order logic are undecidable, thus we are tackling problems for which full computational solutions do not exist. Moreover, we will swiftly come face-to-face with a tough practical issue. As we shall see, it is rather easy to extend our propositional tableau system to a (sound and complete) proof system for first-order logic. Unfortunately, while this system is conceptually simple and elegant, it is a computational nightmare. This difficulty (unsurprisingly) is directly related to the treatment of quantifiers, and resolving this problem will lead us to the concept of *unification*, the cornerstone of automated reasoning for first-order logic. Unification is the key to a computationally realistic treatment of the quantifiers, and with its help we will convert our propositional tableau and resolution provers into full-fledged provers for first-order logic.

Those are the topics we discuss in the first part of the chapter, and the reader who works through this material will gain a firm grasp of the basic issues involved in implementing first-order tableau and resolution systems and will also (we hope) develop a healthy respect for the art of first-order theorem proving. In the second part of the chapter we go on to show that this respect is not misplaced. First, we show that neither of our theorem provers are serious inference tools—bluntly, both are toys. Second, we show that the issue of undecidability leads to difficulties relevant to the practice of computational semantics. Theorem proving alone is not enough to give us the kind of grip we would like to have on the consistency and informativity checking tasks. What are we to do?

We make two moves, one conceptual, the other practical. On the conceptual level, we explain why it is necessary to make a distinction

between *negative* and *positive* checks for consistency and informativity checking. On the practical level, we show how sophisticated off-the-shelf theorem provers can be used to provide negative checks for consistency and informativity, and how the relatively new technology of model builders can be used to provide (partial) positive checks for consistency and informativity. That is, we abandon the do-it-yourself approach to automated reasoning. Instead, we show the reader how the powerful tools developed by the automated reasoning community over the past 40 years can be integrated into our architecture for computational semantics.

We have a lot of ground to cover. Let's get started.

5.1 A First-Order Tableau System

We can extend our propositional tableau system to a system for first-order logic by adding four new tableau rules. Conceptually, the extension is very simple. Our new tableau rules will allow us to get rid of quantifiers by substituting suitable terms for bound variables. In effect, they let us reduce first-order formulas to propositional ones.

What kinds of rules enable us to do this? Let's consider two examples. Suppose a tableau contains the signed formula $T\forall x \text{KILLER}(x)$, and suppose we are working with a first-order language containing the constants JULES and BUTCH and a 1-place function symbol FATHER. Then it is legitimate to extend the tableau by adding on any of the following signed formulas: $T\text{KILLER}(\text{JULES})$, $T\text{KILLER}(\text{BUTCH})$, and

$$T\text{KILLER}(\text{FATHER}(\text{FATHER}(\text{FATHER}(\text{FATHER}(\text{JULES}))))).$$

(After all, if everyone is a killer, every term picks out a killer.) More generally, given any universal formula prefixed by the sign T , we are free to take a copy of it, throw away the universal quantifier on the copy, substitute any term for the newly freed variable in the matrix, prefix the new formula by T , and extend the tableau with the result.

Analogous tableau extensions are legitimate when we are told that an existential sentence is false. For example, suppose a tableau contains $F\exists x \text{SHOOT}(\text{JULES}, x)$. Then we can deduce that $F\text{SHOOT}(\text{JULES}, \text{BUTCH})$, that $F\text{SHOOT}(\text{JULES}, \text{FATHER}(\text{JULES}))$, and so on. (After all, if it's false that Jules shoots someone, then it's false that Jules shoots any person we care to name.)

Such examples lead us to formulate the following two tableau rules, $T\forall$ and $F\exists$:

$$\frac{T\forall x\phi}{T\phi(\tau)}$$

$$\frac{F\exists x\phi}{F\phi(\tau)}$$

Here $\phi(\tau)$ denotes the result of replacing the variable bound by the quantifier by some closed term τ . (Recall that a closed term is a term that does not contain any variables.) We call these two rules *universal rules*.

These rules clearly trade on the semantic intuitions underlying the tableau method. Note, however, that they differ from the propositional tableau rules in two important respects. First, each rule licenses as many tableau extensions as there are closed terms in the language. Second, in general we will have to apply a universal rule more than once to a particular occurrence of a signed formula. Here's a simple example. We shall give a tableau proof of $\forall x \text{DIE}(x) \rightarrow \text{DIE}(\text{MIA}) \wedge \text{DIE}(\text{ZED})$.

1	$F(\forall x \text{DIE}(x) \rightarrow \text{DIE}(\text{MIA}) \wedge \text{DIE}(\text{ZED}))$	
2	$T \forall x \text{DIE}(x)$	$1, F_{\rightarrow}$
3	$F(\text{DIE}(\text{MIA}) \wedge \text{DIE}(\text{ZED}))$	$1, F_{\rightarrow}$
4	$T \text{DIE}(\text{MIA})$	$2, T_{\forall}$
5	$T \text{DIE}(\text{ZED})$	$2, T_{\forall}$
6	$F \text{DIE}(\text{MIA})$	$3, F_{\wedge}$
7	$F \text{DIE}(\text{ZED})$	$3, F_{\wedge}$

Note the way we had to apply T_{\forall} twice to line 2: once to get $T \text{DIE}(\text{MIA})$, and once to get $T \text{DIE}(\text{ZED})$. Thus we have lost one of the pleasant properties of propositional tableaus: it is no longer true that we are through with an occurrence of a signed formula once we've applied a rule to it.

What sort of tableau rules are needed to deal with signed formulas of the form $T \exists x \phi$ or $F \forall x \phi$? This is a more subtle matter. Suppose a tableau contains the signed formula $T \exists x \text{KILLER}(x)$. It is *not* legitimate on the basis of this information to deduce that $T \text{KILLER}(\text{JULES})$, or that $T \text{KILLER}(\text{BUTCH})$, or indeed to deduce that $T \text{KILLER}(\text{closed-term})$ for *any* closed term of the language we are working with. *Someone* is a killer—but we don't know who. So how are we to eliminate the quantifier?

Actually, the solution is straightforward: we invent a brand new name for the entity whose existence is asserted, and eliminate the quantifier by substituting this new name. Such brand new names are called *parameters*, and by using parameters we can deal with true existential statements, and false universal ones too. For example, if a tableau contains $T \exists x \text{KILLER}(x)$, we give this killer (whoever he or she is) a new name (that is, we choose a new parameter, say c) and extend the tableau by asserting $T \text{KILLER}(c)$. Similarly, given the signed

formula $F\forall x \text{RELIGIOUS}(x)$, we christen the unbeliever (whoever he or she is) with a new name (say c_8) and extend the tableau by asserting $\text{FRELIGIOUS}(c_8)$.

Let's make these ideas precise. Suppose we are working in a first-order language over some vocabulary V . Let PAR be a (countably infinite) set of new constant symbols, that is, constant symbols that *don't* belong to V . We'll call these new constant symbols parameters, and reserve the symbols c, c_1, c_2, \dots , and so on, for them. From now on, when we want to do tableau proofs, we *won't* work in our original language (that is, the language built over the vocabulary V). Rather we'll work in the first-order language whose vocabulary consists of all the original vocabulary V , and in addition, all the new constant symbols in PAR .

Given these ideas, it's easy to define the rules F_\forall and T_\exists :

$$\frac{F\forall x\phi}{F\phi(c)} \qquad \frac{T\exists x\phi}{T\phi(c)}$$

Here $\phi(c)$ denotes the result of substituting a parameter c that we haven't used so far on that branch of the tableau, for the newly freed variable in the matrix. We call these two rules *existential rules*.

Two points about these rules must be grasped. First, when we use the existential rules, it is absolutely vital that we substitute parameters that haven't been used so far (at least, on that branch of the tableau). To see why, suppose a tableau contains both $T\exists x \text{KILLER}(x)$ and $T\exists x \text{RELIGIOUS}(x)$ on one of its branches. Suppose we first apply T_\exists to $T\exists x \text{KILLER}(x)$ using the parameter c_5 (which, let us suppose, hasn't been previously used) to name the killer. That is, we extend the tableau with $T\text{KILLER}(c_5)$. Now, suppose that at some later stage we apply T_\exists to the occurrence $T\exists x \text{RELIGIOUS}(x)$ on that same branch. It would be outrageous to re-use the parameter c_5 . If we did this (that is, if we 'deduced' that $\text{TRELIGIOUS}(c_5)$) we would in effect be claiming that there is a single individual (namely the one named by c_5) who is both a killer and religious. This simply doesn't follow from the given information. All we know is that there is at least one killer, and at least one religious person. They may well be different people, hence we need to assign each of them a fresh new name. In short, once we've used an existential rule to replace a quantifier by a parameter, that parameter becomes 'old', and cannot later be re-used on the same branch. (As tableau branches are independent of each other—each records a separate attempt to falsify the original formula—it does not matter if the same parameter is re-used on different branches.)

There is a second important point that the reader should note: the

definition of the existential rules has consequences for the universal rules. When carrying out first-order tableau proofs we no longer work in the original first-order language, but in the original language enriched with an infinite collection PAR of new constants. Now, constant symbols—including parameters—are closed terms, so we should be free to substitute parameters when we use a universal rule. In fact, it's *crucial* that we be able to do this, as the following example will make clear. We will show that

$$\exists x \forall y \text{SHOOT}(x,y) \rightarrow \forall y \exists x \text{SHOOT}(x,y)$$

is a theorem. (This is a rather pretty example. It's simple, but puts all four quantifier rules to work.)

1	$F(\exists x \forall y \text{SHOOT}(x,y) \rightarrow \forall y \exists x \text{SHOOT}(x,y))$	
2	$T \exists x \forall y \text{SHOOT}(x,y)$	1, F_{\rightarrow}
3	$F \forall y \exists x \text{SHOOT}(x,y)$	1, F_{\rightarrow}
4	$T \forall y \text{SHOOT}(c_1, y)$	2, T_{\exists}
5	$F \exists x \text{SHOOT}(x, c_2)$	3, F_{\forall}
6	$T \text{SHOOT}(c_1, c_2)$	4, T_{\forall}
7	$F \text{SHOOT}(c_1, c_2)$	5, F_{\exists}

The key point to notice about this proof is the way the existential and universal rules interact. In particular, note the way we used the existential rules to introduce the new parameters (c_1 in line 4 and c_2 in line 5) and then used the universal rules to make further use of these symbols. It should be clear from this example that it is vital that the universal rules have access to the parameters.

Well, that's our first-order tableau system. Now, our goal is to get a computational grip on the consistency and informativity checking tasks for first-order logic. Given that this is our goal, how useful will this tableau system be to us?

We need to be careful how we answer this. The first (and most fundamental) point that should be made is that this first-order tableau system does *not* give rise to an algorithm for determining which first-order formulas are valid. An algorithm is a recipe which, when given an instance of a problem to solve, halts after a finite number of steps with the correct answer. *There is no algorithm at all for determining the validity of arbitrary first-order formulas.* That is, first-order validity is *undecidable*. It is certainly possible to implement various kinds of proof system for first-order logic (for example, tableau and resolution systems) and we shall do so in this chapter, but no implementation of any system is guaranteed to terminate on all possible input. It is worth emphasising that the tableau system just described *is* (sound

and) complete. That is, if a formula ϕ is valid, then it is possible to construct a (finite) closed tableau that has $F\phi$ as its root node. Thus the proof system just described is a genuine syntactic analysis of the semantic concept of first-order validity, and indeed a rather natural one. However, this analysis does not yield an algorithm for determining which first-order formulas are valid: if a first-formula that is not valid is given as input, then in some cases the tableau construction process never halts.

Fair enough. The tableau system can't possibly serve as a full solution to the consistency and informativity checking tasks, for nothing can do this, so let's ask a more modest question. Many first-order theorem provers are useful practical tools. Proof search won't always terminate, but these provers work well on a wide range of input formulas, and can be used as components of larger systems quite satisfactorily. So the question we should next pose is: how good is the tableau system just described as a practical basis for automated first-order theorem proving? This question has a clear answer: *it's terrible*. Although our tableau system is conceptually simple, it's a computational nightmare.

The problem lies with the universal rules. They offer us a vast menu of substitutable terms. If $T\forall x\phi$ belongs to a tableau, then we are free to extend it by first adding $\phi(c_1)$ then $\phi(c_2)$ then $\phi(c_3)$, ..., and so on. (We could have done this in the previous tableau, for example, starting at line 4.) Most such extensions will be completely pointless. In the examples given above, it was intuitively clear which substitutions were sensible; we used our sense of what was relevant to guide our choice of substitutions. Unfortunately, computers lack our intuitions. If we want a reasonably practical implementation, we need to find a method choosing substitutions that doesn't depend on human insight.

Here's what we shall do. First, we'll change the universal rules slightly: we'll never substitute closed terms, we'll *always* substitute free variables instead. In a sense, we are not going to make a real choice of substitutions at all—we are going to use free variables as 'dummies' that will enable us to delay making this decision. In this way, we will gradually build up a whole system of 'substitution equations', a set of constraints on variable values that contain a great deal of information about the terms in the tableau we are building. Crucially, there is an algorithm for solving such constraint sets, the famous *unification algorithm*. We will use unification to look for solutions to the constraints that lead to branch closure.

That's an outline of our strategy. An awful lot of detail remains to be filled in. For a start, blending unification with tableaus is going to force us to rethink the existential rules, and there are many other details

that will require careful attention. Nonetheless, unification is the key to further progress with tableau theorem proving (and as we shall see, it is utterly fundamental to first-order resolution theorem proving) so let's examine this concept in some detail.

5.2 Unification

Unification is the process of carrying out substitutions on two terms so that they become identical. The reader who has made it this far will certainly have an informal idea of what substitutions are, and indeed, since we are working with Prolog, a good working knowledge of what unification involves in practice. However Prolog's default version of unification is *not* suitable for theorem proving purposes. Given two terms, Prolog usually does not make what is known as the occurs check, rather it just rushes ahead and tries to unify them. (Don't worry if you don't know what the occurs check is; we will explain it later in this section.) This is fine (and certainly efficient) if the terms are unifiable, but it can lead to non-terminating computations if they are not. In this section we shall explain what unification is, and what it means to apply unification to terms, to formulas, and indeed to entire tableaus. We'll then point out that there is an in-built Prolog mechanism that does exactly what we want.

Suppose we have chosen the first-order language we are going to work with. Then a *substitution* is a function that maps the set of variables to the terms of this language. We use the notation $x\sigma$, (rather than $\sigma(x)$) to denote the value of x under the substitution σ .

We are most interested in *finite substitutions*. These are substitutions which only assign new terms to a finite number of variables; the rest they leave alone. That is, if σ is a finite substitution, then for all but a finite number of variables, $x\sigma = x$.

The simplest finite substitution is the one which does not assign new terms to *any* of the variables. This substitution is called the *identity substitution*, and we denote it by $\{\}$. We also have a special notation for other finite substitutions, namely:

$$\{x_1/\tau_1, \dots, x_n/\tau_n\}.$$

Here x_1, \dots, x_n are distinct variables, τ_1, \dots, τ_n are terms, and $\tau_i \neq x_i$ for any i from 1 to n . The notation x_i/τ_i means that the variable x_i is mapped to τ_i .

As we have defined them, substitutions only act on variables. But we will need to carry out substitutions on terms, formulas, and indeed, on entire tableaus. Let's see how to recursively define these concepts.

Substitutions on terms. Let σ be a substitution and τ a term. Then:

1. If τ is a variable x , then $\tau\sigma = x\sigma$, if this is defined. Otherwise $\tau\sigma$ is undefined.
2. If τ is a constant, then $\tau\sigma = \tau$.
3. If τ is $f(\tau_1, \dots, \tau_n)$, then $[f(\tau_1, \dots, \tau_n)]\sigma = f(\tau_1\sigma, \dots, \tau_n\sigma)$.

Note that in the clause for f we used the notation $[f(\tau_1, \dots, \tau_n)]\sigma$ for the application of σ to the complex term; when we apply a substitution to a complex term, it's useful to have brackets to indicate precisely the symbols to which the substitution applies. We shall now extend substitution to formulas, and will use the square bracket notation here too. We'll also need another piece of notation. If σ is a substitution, then by σ_x we mean the substitution that is exactly like σ except that $x\sigma_x = x$ (in other words, this substitution does *not* affect x).

We can now recursively define $\phi\sigma$, the result of applying the substitution σ to the formula ϕ .

Substitutions on formulas.

1. If $R(\tau_1, \dots, \tau_n)$ is an atomic formula, then $[R(\tau_1, \dots, \tau_n)]\sigma = R(\tau_1\sigma, \dots, \tau_n\sigma)$. (Here R is an n -place relation symbol.);
2. $[\neg\phi]\sigma = \neg[\phi\sigma]$;
3. $[\phi \wedge \psi]\sigma = [\phi\sigma] \wedge [\psi\sigma]$, $[\phi \vee \psi]\sigma = [\phi\sigma] \vee [\psi\sigma]$, and
 $[\phi \rightarrow \psi]\sigma = [\phi\sigma] \rightarrow [\psi\sigma]$;
4. $[\forall x\phi]\sigma = \forall x[\phi\sigma_x]$, and $[\exists x\phi]\sigma = \exists x[\phi\sigma_x]$.

Finally, we define substitution for entire tableaus:

Substitutions on signed tableaus. If σ is a substitution and T is a signed tableau, then $T\sigma$ is the signed tableau obtained by replacing every signed formula of the form $T\phi$ in T by $T[\phi\sigma]$, and every signed formula of the form $F\phi$ in T by $F[\phi\sigma]$.

With these definitions out of the way, let us return to our main task: understanding unification.

Because substitutions are functions, we can *functionally compose* them in the usual way. That is, if σ_1 and σ_2 are substitutions, then we can define a new substitution $\sigma_1\sigma_2$, the composition of σ_1 and σ_2 . For every variable x , we define $x(\sigma_1\sigma_2)$ to be $(x\sigma_1)\sigma_2$. That is, $\sigma_1\sigma_2$ first carries out the substitution σ_1 and then carries out the substitution σ_2 .

There is one more concept we need to discuss before we can define unification. Let's approach it via an example. Suppose we want to make the terms $f(c, y, w)$ and $f(x, y, g(z))$ identical (where c is a constant, w , x , y , and z are variables, f is a 3-place function symbol, and g is

a 1-place function symbol). Let σ_1 be the substitution $\{x/c, w/g(z)\}$. Applying σ_1 to these terms has the desired result, for $[f(c, y, w)]\sigma_1 = f(x, y, g(z))\sigma_1 = f(c, y, g(z))$.

But there are other ways of making these terms identical. For example, let σ_2 be the finite substitution $\{x/c, w/g(z), y/h(u, x)\}$. Applying σ_2 to either term yields $f(c, h(u, x), g(z))$. Nonetheless, clearly σ_1 is a more general solution to the problem: σ_2 does too much work. For suppose we apply σ_1 to some term. Then, if we want to (or need to) we are always free to later give y the value $h(u, x)$. To do so we need simply apply the substitution $\{y/h(u, x)\}$, and this two-step process gives us the same effect we would have achieved by directly applying σ_2 . But of course, we might not want to take this further step. (Perhaps mapping y to $h(u, x)$ is incompatible with other substitutions we need to make.) If we use σ_2 to solve the problem, we are over-committing ourselves.

Such considerations motivate the following definition. A substitution σ_1 is said to be *more general* than a substitution σ_2 if and only if there is some substitution θ such that $\sigma_2 = \sigma_1\theta$. That is, σ_1 is more general than σ_2 if we can get the effect of σ_2 by first carrying out σ_1 and then making a further substitution θ . Thus, reverting to our motivating example, $\{x/c, w/g(z)\}$ is more general than $\{x/c, w/g(z), y/h(u, x)\}$ because there is a substitution θ (namely $\{y/h(u, x)\}$) such that $\{x/c, w/g(z), y/h(u, x)\} = \{x/c, w/g(z)\}\theta$.

Incidentally, note that under this (standard) definition of ‘more general than’, each substitution σ is more general than itself. This is because we can get the effect of σ by first carrying out σ and then carrying out the identity substitution $\{\}$. Thus the ‘more general than’ relation is reflexive. It is also transitive. That is, if σ_1 is more general than σ_2 , and σ_2 is more general than σ_3 , then σ_1 is more general than σ_3 . The reader may like to try proving this.

We are ready for the key definition.

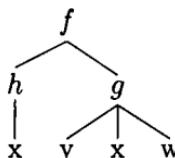
Unification. Let τ_1 and τ_2 be terms. A substitution σ is a *unifier* for τ_1 and τ_2 if and only if $\tau_1\sigma = \tau_2\sigma$. Terms τ_1 and τ_2 are said to be *unifiable* if and only if they have a unifier. A substitution σ is a *most general unifier* (or *MGU*) for two terms if and only if it is a unifier for these terms, and is more general than any other unifier for those terms.

Now we know what unification is—but what is involved in computing unifiers? Ideally, what we want is an algorithm which will take as input two terms and determine whether or not they are unifiable. If the terms are unifiable, it should return their unifier as output. If they are not unifiable, it should halt and tell us so.

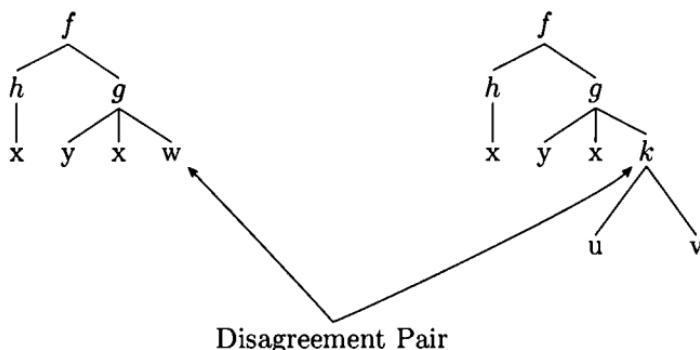
Such algorithms exist. Let’s consider one of the more straightforward

ones in some detail.

To appreciate what this algorithm does, we really need to think of terms as trees (in much the same way as we thought of formulas as trees when we discussed hole semantics in Chapter 3). For example, consider the term $f(h(x), g(y, x, w))$. Its analysis tree—that is, the tree showing how it is built up out of sub-terms—looks like this:



When are two terms different? For our purposes, the following answer is the most useful: two terms are different if and only if their analysis trees contain at least one *disagreement pair*. What's a disagreement pair? Here's an example.



Intuitively, the pair of terms $(w, k(u, v))$ is a disagreement pair for these terms because they are distinct terms that occupy ‘corresponding places’ in the two analysis trees. What is meant by ‘corresponding places’? Simply the nodes that one reaches by following the same sequence of transitions from the root in the two trees. For example, if we follow the transition sequence {second daughter, third daughter} from the root of the first tree we arrive at the node labelled w , and if we follow the same transition sequence in the second tree we arrive at the node labelled k .

Disagreement pairs make terms different, thus unification algorithms should try to eliminate disagreement pairs. When are disagreement pairs eliminable, and how can they be eliminated?

First, suppose that two terms τ_1 and τ_2 are different because there is a disagreement pair (d_1, d_2) such that *neither* d_1 nor d_2 is a variable.

Then there is nothing we can do. No substitution can help us out. The terms τ_1 and τ_2 are not unifiable.

Now for the tricky question. Suppose that one of these terms (d_1 say) is a variable. Are the two terms unifiable? The answer is ‘yes’, *provided that the variable d_1 does not occur in d_2* . Think about it. Suppose d_1 is a variable, say x , and that x *doesn’t* occur in d_2 . Then we can eliminate this disagreement pair very easily: we simply need to replace x by d_2 (That is, we need simply perform the substitution $\{x/d_2\}$.) To give a concrete example, the disagreement pair $(w, k(u, v))$ shown above is of this form. We eliminate it by replacing w by $k(u, v)$.

On the other hand, if d_1 is a variable (say x) and x *does* occur in d_2 , then unification is impossible. For example, suppose that d_2 is $f(x)$. Then, no matter what value we choose for x , we will *never* render these two terms identical—we’ll always have that extra function symbol f to reckon with.

Let’s make these observations a little more systematic. Suppose we have found a disagreement pair (d_1, d_2) . We will call this pair a *simple* disagreement pair if and only if at least one of the terms d_1 or d_2 is a variable that does not occur in the other. (It could happen, of course, that *both* d_1 and d_2 are variables that don’t occur in the other—for example if $d_1 = x$ and $d_2 = y$. That’s fine. As long as there’s at least one we’re happy.) Simple disagreement pairs are the ones we can repair. We do so by carrying out what we shall call the *relevant repair*. If d_1 is a variable that does not occur in d_2 , then the relevant repair is the substitution $\{d_1/d_2\}$. If d_2 is a variable that does not occur in d_1 , then the relevant repair is the substitution $\{d_2/d_1\}$. If both d_1 and d_2 are variables that don’t occur in the other, then there are two substitutions that will repair the problem, namely $\{d_1/d_2\}$ and $\{d_2/d_1\}$. We’ll arbitrarily stipulate that in such cases the relevant repair is $\{d_1/d_2\}$.

On the other hand, if the disagreement pair we have found is *not* simple, there’s nothing we can do. Either we have that neither d_1 nor d_2 is a variable, or one of them is a variable that occurs in the other term.

We now know which disagreement pairs are eliminable, and how to carry out the elimination. And this means we are only one small step away from an algorithm for solving the unification problem. To unify two terms, simply try and eliminate *all* the disagreement pairs! This idea immediately suggests the following non-deterministic algorithm:

input terms τ_1 and τ_2

let $\sigma := \{\}$

```

while  $\tau_1\sigma \neq \tau_2\sigma$ 
    choose a disagreement pair  $(d_1, d_2)$  for  $\tau_1\sigma, \tau_2\sigma$ 
    if  $(d_1, d_2)$  is not simple then
        write  $\tau_1$  and  $\tau_2$  are not unifiable and HALT
    else
        let  $\sigma := \sigma \cup \rho$ , where  $\rho$  is the relevant repair
    endif
endwhile

```

This is a genuine computational solution of the unification problem. No matter which two terms it is given as input, it will halt after finitely many steps. When it halts, it will either have told us that the terms are not unifiable (and if it says this, it's right!) or it will have found out how to build the MGU of the two terms. These claims are not obvious; they require proof. The reader interested in finding out more should consult the references cited in the Notes.

In fact, the algorithm has one additional property: the MGUs it produces are *idempotent*. That is, if σ is an MGU produced by this algorithm, then $\sigma\sigma = \sigma$. Why on earth is such an abstract looking property interesting? The answer is: idempotent MGUs provide us with an easy way of solving the *simultaneous unification problem*, and this is the problem that will be important when working with tableaus.

We will be using unification to try and close tableau branches. That is, we will look for branches containing pairs of atomic formulas $T(R(\tau_1, \dots, \tau_n))$ and $F(R(\tau'_1, \dots, \tau'_n))$. If we can find a substitution σ that makes τ_1 identical to τ'_1 , τ_2 identical to τ'_2 , ..., τ_n identical to τ'_n , then by applying σ we obtain a branch containing contradictory formulas, that is, a closed branch. Thus we need to solve the problem of finding a single substitution that makes n pairs of terms identical; this is called the simultaneous unification problem.

Now, this problem may look harder than the ordinary unification problem, but actually it's not. It can be solved as follows. To find a substitution that identifies n pairs of terms, first find an *idempotent* MGU σ_1 for τ_1 and τ'_1 . We can do this using the above algorithm. Next, find an *idempotent* MGU σ_2 for $\tau_2\sigma_1$ and $\tau'_2\sigma_1$. Again we can do this using the above algorithm. Next, find an *idempotent* MGU σ_3 for $\tau_3\sigma_1\sigma_2$ and $\tau'_3\sigma_1\sigma_2$ In fact, all we need to do is keep 'chaining together' the solutions to each individual pair. The substitution $\sigma = \sigma_1\sigma_2\dots\sigma_{n-1}\sigma_n$ that is obtained in this way is a simultaneous MGU for the n pairs of terms. For this 'chaining together' method to work, the substitutions constructed at each step must be idempotent, and

as the above algorithm yields idempotent MGUs, it really does deliver everything we shall need for tableau theorem proving.

The basic concepts should now be clear, so let's turn to a more practical issue: how can we use Prolog to unify terms? Now, it is important to realise that we cannot simply use the Prolog's default unification mechanism (that is, the `=/2` predicate). Usually Prolog does not bother making the *occurs check*. That is, given a disagreement pair (d_1, d_2) , one of which is a variable, Prolog does *not* standardly check whether this variable occurs in the other term, but will go straight ahead and attempt to carry out what it thinks the required repair is. This can lead it to attempt to unify terms that aren't unifiable, which can lead to stack overflows and other undesirable behaviour.

Now, we are interested in using unification as part of a first-order theorem prover. We really need to know whether or not two terms are unifiable, and we certainly *don't* want Prolog to mess things up with its "Hey, just go for it!" behaviour. Fortunately, any standard Prolog offers the following alternative:

`unify_with_occurs_check/2`.

As its name suggests, this predicate carries out the full version of term unification we discussed above, including the occurs check. When we implement our first-order tableau and resolution systems, we shall perform all the unifications we require with the help of this predicate.

Exercise 5.2.1 Try unifying `f(X)` and `X` with the version of Prolog you're using. First use `=/2` and then use `unify_with_occurs_check/2`. (If your Prolog does not support the `unify_with_occurs_check/2` predicate, do the next exercise.)

Exercise 5.2.2 Although `unify_with_occurs_check/2` gives us a standard solution to handling the occurs check, it is an interesting exercise to try writing your own version. Moreover, some Prolog implementations don't support it. Try defining your own version of this important predicate (if you get stuck, the Notes at the end of the chapter tell you where you can find answers).

5.3 Free-Variable Tableaus

Let's see how we can use unification to define a more computationally realistic signed tableau proof system.

As we discussed earlier, the problem with our first attempt at a first-order tableau system lay with the universal rules. By substituting free variables, and gradually building up a system of constraints which we will solve using unification, we hope to bypass the need for human

insight. Let's work through this idea in detail, and see where it leads. As a first step, here are our new universal rules:

$$\frac{T\forall x\phi}{T\phi(v)} \qquad \frac{F\exists x\phi}{F\phi(v)}$$

Here $\phi(v)$ denotes the result of replacing all instances of the variable that the quantifier bound by a new variable v that does not occur bound anywhere in the tableau. (This restriction is simply to prevent any ‘accidental bindings’ taking place. In fact, every time we apply these rules, we’re going to substitute a previously unused variable.)

So far so good—but a moment’s thought will convince the reader that our new strategy could lead to serious problems with the existential rules. Recall that the basic idea behind the existential rules was to invent a brand new name for the entity asserted to exist (we called these new names ‘parameters’) and to eliminate the quantifier by substituting these parameters. Unification threatens to undercut this strategy: the substitutions it makes may undo all our careful choices of new names.

Here’s a concrete example. Consider the formula

$$\exists y(\neg R(x, y) \wedge R(x, x)).$$

This formula is satisfiable (it can be satisfied in a two-element model in which one point is not R -related to the other element but is R -related to itself). Now, if we eliminate the existential quantifier using our new free-variable tableau rule we get

$$\neg R(x, w) \wedge R(x, x).$$

And now we have a problem. The tableau rule for \wedge allows us to break this formula down into $\neg R(x, w)$ and $R(x, x)$, so both these formulas will be sitting on the same tableau branch. If we then apply unification we will get $\neg R(x, x)$ and $R(x, x)$, and of course the branch will close. That is, we started with a satisfiable formula, and our new method has turned it into something unsatisfiable!

In short, we seem to be in a dilemma. The use of free variables in the universal rules is essentially a delaying device; we don’t want to make a real choice of what to substitute, we want unification to sort it all out for us. But if we delay in this way, how can we guarantee that unification will respect the ‘new names’ condition that is crucial to the existential rules?

A very clever—and very simple—idea allows us a way round this problem. We are going to substitute *structured* terms when we use the existential rules; the term structure itself will ensure that unification cannot spoil anything. To be more precise, we are to use what are known

as *Skolem terms*. Without further ado, here are the new existential rules:

$$\frac{F \forall x \phi}{\overline{F \phi(s(x_1, \dots, x_n))}} \qquad \frac{T \exists x \phi}{\overline{T \phi(s(x_1, \dots, x_n))}}$$

Here $\phi(s(x_1, \dots, x_n))$ denotes the result of replacing the variable bound by the quantifier by $s(x_1, \dots, x_n)$. In the substituted expression, s is a new *Skolem function symbol*, and x_1, \dots, x_n are all the free variables in ϕ distinct from x . If there are no free variables in ϕ (apart from x) then s is a new *Skolem constant*. What does this mean, and why does it work?

The basic idea is a straightforward generalization of what we did earlier with parameters. Once again, instead of working with the first-order language built over the original vocabulary V , we are going to choose a set of new symbols SKO consisting of a countably infinite set of Skolem constants (these are essentially the same as our earlier parameters) and for every natural number n a countably infinite set of Skolem function symbols of arity n . When carrying out tableau proofs, we won't work in the original first-order language, but in the first-order language whose vocabulary consists of all the original vocabulary V plus all the new symbols in SKO .

The symbols in SKO enable us to manufacture new names. Crucially, however, because we now have Skolem *function* symbols at our disposal, we can do something that we couldn't do with parameters: we can 'build in newness' in a way that will survive the unification operation. Look at the term the existential rules demands we substitute: $s(x_1, \dots, x_n)$, where x_1, \dots, x_n are all the free variables in ϕ distinct from x . Now recall our discussion of the occurs check. Quite simply, $s(x_1, \dots, x_n)$ *cannot* unify with any of x_1, \dots, x_n . Our new 'Skolem structured' term really will be new.

Let's go back to our example. Consider again the formula

$$\exists y (\neg R(x, y) \wedge R(x, x)).$$

This time when we eliminate the existential quantifier we get

$$\neg R(x, s(x)) \wedge R(x, x).$$

And now everything is fine. Sure, the tableau rule for \wedge will allow us to break this formula down into $\neg R(x, s(x))$ and $R(x, x)$. But as $s(x)$ cannot unify with x this is unproblematic. We'll never pull a contradiction out of this pair of formulas.

So we now have the new quantifier rules we need. Only one task remains: bringing unification into play.

We want unification to be the ‘intelligence’ guiding the proof search. We’re going to use our quantifier rules essentially ‘blindly’: we’re simply going to build up a set of constraints and hope that unification can do something useful with them. In particular, we hope that unification will be able to close branches for us. How could it do this for us? Let’s consider an example. We shall redo our proof of

$$\exists x \forall y \text{SHOOT}(x,y) \rightarrow \forall y \exists x \text{SHOOT}(x,y)$$

using our new quantifier rules and unification. Here are the first seven steps of the construction.

1	$F(\exists x \forall y \text{SHOOT}(x,y) \rightarrow \forall y \exists x \text{SHOOT}(x,y))$	
2	$T \exists x \forall y \text{SHOOT}(x,y)$	1, F_{\rightarrow}
3	$F \forall y \exists x \text{SHOOT}(x,y)$	1, F_{\rightarrow}
4	$T \forall y \text{SHOOT}(s_1, y)$	2, T_{\exists}
5	$F \exists x \text{SHOOT}(x, s_2)$	3, F_{\forall}
6	$T \text{SHOOT}(s_1, v_1)$	4, T_{\forall}
7	$F \text{SHOOT}(v_2, s_2)$	5, F_{\exists}

The first 5 lines are essentially identical with the previous version, save that we have used the Skolem constants s_1 and s_2 (rather than the parameters c_1 and c_2) in lines 4 and 5. (Note that we don’t need to use Skolem functions as neither formula contains free variables.) The real difference occurs in lines 6 and 7. In both lines we have ‘blindly’ instantiated in new variables, namely v_1 and v_2 . So we don’t (yet) have closure.

But closure is easy to get. Consider the substitution $\{v_1/s_2, v_2/s_1\}$. If we apply this to the tableau (recall that we defined the concept of applying a substitution to a tableau in the previous section) we get the following tableau:

1	$F(\exists x \forall y \text{SHOOT}(x,y) \rightarrow \forall y \exists x \text{SHOOT}(x,y))$	
2	$T \exists x \forall y \text{SHOOT}(x,y)$	1, F_{\rightarrow}
3	$F \forall y \exists x \text{SHOOT}(x,y)$	1, F_{\rightarrow}
4	$T \forall y \text{SHOOT}(s_1, y)$	2, T_{\exists}
5	$F \exists x \text{SHOOT}(x, s_2)$	3, F_{\forall}
6	$T \text{SHOOT}(s_1, s_2)$	4, T_{\forall}
7	$F \text{SHOOT}(s_1, s_2)$	5, F_{\exists}

This tableau is closed.

This example motivates the addition of the following *Atomic MGU Closure Rule*:

Suppose that \mathcal{T} is a tableau formed from some initial tableau \mathcal{I} , and that some branch of \mathcal{T} contains a pair of atomic formulas of the form

$T\phi$ and $F\psi$. Then $T\sigma$, where σ is an MGU of ϕ and ψ , is also a tableau formed from the initial tableau \mathcal{I}

This rule is rather different from the other rules we've seen: it's not an extension rule, rather it's a transformation rule. It tells us that if we have a tableau formed from an initial set \mathcal{I} , and we transform it by applying a substitution having certain properties, we obtain another tableau for the same initial set.

The basic idea guiding the choice of transformation should be clear: we want to apply substitutions that could lead to branch closure. One point, however, may be puzzling. Obviously we are interested in pairs of formulas of the form $T\phi$ and $F\psi$, but why have we restricted our attention to *atomic* formulas? The answer is: *simplicity*. It is possible to formulate a more general rule, but this would have to be stated carefully (we would need to avoid 'accidental capture' of variables and checking for accidental capture would be computationally expensive). By restricting our attention to atomic formulas we avoid these difficulties.

And that's our free-variable tableau proof system. Frankly, it's not nearly as nice as our previous system if one wants to prove things by hand—playing with Skolem functions swiftly gets unwieldy, and thinking in terms of unification is cumbersome. But it wasn't designed with the needs of humans in mind, it was designed for automation. And, as we shall now see, for this purpose it is really rather good.

5.4 Implementing Free-Variable Tableaus

We shall now present an implementation of the free-variable signed tableau proof system. The implementation is an extension of the propositional tableau implementation—but it can't be described as a straightforward extension. While the basic ideas underlying signed free-variable tableau make it possible to devise practical implementations, there is more detail to take care of than in the propositional case. Moreover, the fact remains that first-order logic is undecidable, so it seems sensible to defuse the (very real) threat of non-terminating tableau constructions, and this requires a little care.

The following implementation is based on the Fitting (1996) implementation of unsigned free-variable tableaus. Roughly speaking, we have taken our signed propositional tableau implementation as the starting point, and extended it to a first-order system by adopting many of Fitting's techniques and design choices; see the Notes at the end of the chapter for further remarks.

Before examining the main body of the code, let's take a quick look

at the main supporting routines we shall use. Obviously we need predicates to handle substitution. The workhorse is the `substitute/4` predicate (supplied in `comsemPredicates.pl`). The `substitute/4` predicate takes a term, a variable, and a formula as its first three arguments, and returns in its fourth argument the result of substituting the term for each free occurrence of the variable in the formula. With this predicate at our disposal, it is straightforward to define what we mean by instances of quantified formulas:

```

instance(t(all(X,F)),Term,t(NewF)):-  
    substitute(Term,X,F,NewF).  
  

instance(f(some(X,F)),Term,f(NewF)):-  
    substitute(Term,X,F,NewF).  
  

instance(t(some(X,F)),Term,t(NewF)):-  
    substitute(Term,X,F,NewF).  
  

instance(f(all(X,F)),Term,f(NewF)):-  
    substitute(Term,X,F,NewF).

```

To handle the existential rule correctly, we need to be able to generate new Skolem function symbols on demand. The following predicate does this. (The predicates it calls are defined in `comsemPredicates.pl`.)

```

skolemFunction(VarList,SkolemTerm):-  
    newFunctionCounter(N),  
    compose(SkolemTerm,fun,[N|VarList]).
```

We shall also need to know what free variables a formula contains. We associate this information explicitly with each formula. The following predicate does this.

```
notatedFormula(n(Free,Formula),Free,Formula).
```

We are now ready to discuss the main code. As in the propositional case, the outermost predicate is called `closedTableau/2`. This recursively attempts to rule-saturate the input tableau with the aid of the `expand/4` predicate. Moreover, as in the propositional case, the base clause of `closedTableau/2` tests for closure via a predicate called `removeClosedBranches/2`.

But there is an important difference. The first-order version of `closedTableau/2` has a second argument, a number called `Qdepth` (which can be read as ‘quantification depth’). This number is the maximum number of times that we are allowed to apply universal rules in the course of constructing a tableau. This is the mechanism which wards off the threat of non-terminating tableau constructions.

```

closedTableau([],_Qdepth):- !.

closedTableau(OldTableau,Qdepth):-
    expand(OldTableau,Qdepth,TempTableau,NewQdepth), !,
    removeClosedBranches(TempTableau,NewTableau),
    closedTableau(NewTableau,NewQdepth).

```

The `removeClosedBranches/2` predicate tests for branch closure using unification and is identical to the one used in the propositional case. It uses the `closedBranch/1` predicate tests for branch closure using unification. This attempts to find a pair of signed formulas of the form `t(X)` and `f(X)`, checks whether they are atomic formulas (using `basicFormula/1`, as defined in `comsemPredicates.pl`), and then uses the `unify_with_occurs_check/2` predicate discussed in the previous section to test for branch closure.

```

closedBranch/Branch):-
    memberList(n_,t(X)),Branch),
    memberList(n_,f(Y)),Branch),
    basicFormula(X),
    basicFormula(Y),
    unify_with_occurs_check(X,Y).

```

Like its propositional cousin, the `expand/4` predicate is a high level organisational predicate which works its way recursively through the branches of the input tableau, and tries to apply the various kinds of expansion. Its two extra arguments keep track of the input and output quantification depths. Quantification depth is unaffected by all expansions save universal expansions. Each universal expansion uses up one of our predetermined quota of expansions and thus decreases the quantification depth by 1.

In addition, `expand/4` performs a more interesting task. Note the use of `appendLists/3` (in the clause handling universal expansions) to glue the new branch back onto the *end* of the tableau. Why do this? Essentially, it's an attempt to ensure that our predetermined quota of universal expansions are 'spread around fairly'. It would be rather silly to use up our entire quota on a single branch. By using append to 'rotate' the branches on the tableau, we ensure that every branch receives its fair share of universal expansions.

```

expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
    unaryExpansion(Branch,NewBranch).

expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
    conjunctiveExpansion(Branch,NewBranch).

```

```

expand([Branch|Tableau], QD, [NewBranch|Tableau], QD) :-
    existentialExpansion(Branch, NewBranch).

expand([Branch|Tableau], OldQD, NewTableau, NewQD) :-
    universalExpansion(Branch, OldQD, NewBranch, NewQD),
    appendLists(Tableau, [NewBranch], NewTableau).

expand([Branch|Tableau], QD, [NewBranch1, NewBranch2|Tableau], QD) :-
    disjunctiveExpansion(Branch, NewBranch1, NewBranch2).

expand([Branch|Rest], OldQD, [Branch|Newrest], NewQD) :-
    expand(Rest, OldQD, Newrest, NewQD).

```

Now for the predicates that actually carry out the expansions. The expansion predicates for unary, conjunctive, and disjunctive formulas are essentially the same as in the propositional case. The only difference is that we have to translate a notated formula into a signed formula to calculate its components, and then translate the components back again into notated components.

```

unaryExpansion(Branch, [NotatedComponent|Temp]) :-
    unary(SignedFormula, Component),
    notatedFormula(NotatedFormula, Free, SignedFormula),
    removeFirst(NotatedFormula, Branch, Temp),
    notatedFormula(NotatedComponent, Free, Component).

conjunctiveExpansion(Branch, [NotatedComp1, NotatedComp2|Temp]) :-
    conjunctive(SignedFormula, Comp1, Comp2),
    notatedFormula(NotatedFormula, Free, SignedFormula),
    removeFirst(NotatedFormula, Branch, Temp),
    notatedFormula(NotatedComp1, Free, Comp1),
    notatedFormula(NotatedComp2, Free, Comp2).

disjunctiveExpansion(Branch, [NotComp1|Temp], [NotComp2|Temp]) :-
    disjunctive(SignedFormula, Comp1, Comp2),
    notatedFormula(NotatedFormula, Free, SignedFormula),
    removeFirst(NotatedFormula, Branch, Temp),
    notatedFormula(NotComp1, Free, Comp1),
    notatedFormula(NotComp2, Free, Comp2).

```

Needless to say, the most interesting predicates are those for the quantifiers: `existentialExpansion/2` and `universalExpansion/4`. Note the use of `skolemFunction/2` to instantiate a new Skolem term in `existentialExpansion/2`. In `universalExpansion/4`, note that quantification depth is decremented, a new variable V is ‘blindly’ instantiated, and then—on the very last line—note the way `appendLists/3`

is used to replace the universal formula we have been working with back onto the *end* of the branch. Why do this? Well, we need to replace the formula because (as we have already discussed) we will often have to re-use universal formulas. But then it makes very good sense to replace the formula at the *end* of the branch. If we leave it where it is, we run the risk of using up our entire quota of universal rule applications on this one formula. It is far more sensible to ensure fairness by ‘rotating’ the universal formulas on the branch, much as we rotated the branches of the tableau earlier.

```

existentialExpansion/Branch, [NotatedInstance|Temp]) :-  

    notatedFormula(NotatedFormula, Free, SignedFormula),  

    existential(SignedFormula),  

    removeFirst(NotatedFormula, Branch, Temp),  

    skolemFunction(Free, Term),  

    instance(SignedFormula, Term, Instance),  

    notatedFormula(NotatedInstance, Free, Instance).  
  

universalExpansion/Branch, OldQD, New, NewQD) :-  

    OldQD > 0, NewQD is OldQD - 1,  

    memberList(NotatedFormula, Branch),  

    notatedFormula(NotatedFormula, Free, SignedFormula),  

    universal(SignedFormula),  

    removeFirst(NotatedFormula, Branch, Temp),  

    instance(SignedFormula, V, Instance),  

    notatedFormula(NotatedInstance, [V|Free], Instance),  

    appendLists([NotatedInstance|Temp], [NotatedFormula], New).

```

There is one tricky point in the definition of the universal expansion predicate: the use of `memberList/2` in the second line. At first sight this seems superfluous (note that we don’t have it in the other expansion predicates). But it is needed. Recall that there are two types of signed formulas to which the universal rule needs to be applied: true universal statements, and false existential statements. If we leave out the call to `memberList/2`, then `universal/1` will pick a formula having the first of these forms, strip off its quantifier, add its matrix to the beginning of the branch, and put the universal formula back at the end of the branch (via `removeFirst/3` and `appendLists/3`). That’s fine. Unfortunately, if there is a formula of the other form on the branch as well, it will never be touched. Adding the call to `memberList/2` at the beginning of the clause overcomes this problem: it lets us find the first formula to which we need to apply the rule, whether it is a true universal or a false existential statement.

Now it only remains to spell out what kinds of signed formula we are working with.

```

conjunctive(t(and(X,Y)),t(X),t(Y)).
conjunctive(f(or(X,Y)),f(X),f(Y)).
conjunctive(f(imp(X,Y)),t(X),f(Y)).

disjunctive(f(and(X,Y)),f(X),f(Y)).
disjunctive(t(or(X,Y)),t(X),t(Y)).
disjunctive(t(imp(X,Y)),f(X),t(Y)).

unary(t(not(X)),f(X)).
unary(f(not(X)),t(X)).

universal(t(all(_,_))). 
universal(f(some(_,_))). 

existential(t(some(_,_))). 
existential(f(all(_,_))). 

```

As with the propositional implementation, it's nice to have a predicate that translates the formula into a notated, signed formula, and calls the `closedTableau/2` predicate. We call this predicate `tprove/2`, as in our implementation for propositional tableaus. Its second additional argument is the value of Q-depth.

```

tprove(X,Qdepth) :-
    notatedFormula(NotatedFormula, [], f(X)),
    closedTableau([[NotatedFormula]], Qdepth).

```

We conclude with a warning. Remember that this program can only construct those tableaus which require fewer applications of the universal rules than the user-imposed `Qdepth` limit. Thus if checking a formula with `tprove/2` fails, this most emphatically does *not* mean “Not a first-order tableau theorem”! The formula in question may well be a tableau theorem—but the value of `Qdepth` may be too small to yield a proof. (Incidentally, while trying this program out on the kinds of formulas found in introductory logic books, we usually had `Qdepth` set to 25.) On the other hand, if the program tells us that ϕ is a first-order tableau theorem, this is not open to question. If a closed tableau has been formed from the initial tableau $F\phi$, then ϕ is provable, and that's that.

Exercise 5.4.1 The file `freeVarTabl.pl` is the main file for the implementation of the free-variable tableau prover. There is also a test suite with first-order problems called `folTestSuite.pl`. Examine both files and run the test suite, using the `tproveTestSuite/0` predicate.

Exercise 5.4.2 Add a pretty print predicate to our implementation of first-

order tableaus, that shows the branches and the proof steps in a readable way.

Exercise 5.4.3 Change the definition of `tprove/1` in `freeVarTabl.pl` in such a way that it iterates on Q-depth values until it finds a proof. Do you think this is effectively the same as removing the check on Q-depth in `universalExpansion/4`?

Programs for Free-Variable Tableaus

`freeVarTabl.pl`

The file that contains the code for free-variable tableaus, using ideas from Melvin Fitting's implementation.

`folTestSuite.pl`

A test suite with first-order formulas (non-theorems as well as theorems).

5.5 First-Order Resolution

Our next goal is to turn our propositional resolution prover into a sound and complete first-order theorem prover. We are in for a pleasant surprise. Given what we now know about unification, it will be a relatively simple task to make this extension. Roughly speaking, first-order resolution boils down to repeatedly applying the propositional resolution rule that we studied in the previous chapter while simultaneously carrying out unification.

Recall that we presented propositional resolution as a two-step process: first there was the reduction to (set) conjunctive normal form, and then there was the resolution phase proper. First-order resolution inherits this two-step structure. We shall examine each stage in turn, indicating how each differs from its propositional counterpart.

Clause normal form

In the propositional setting, we first preprocessed the input formula into (set) conjunctive normal form. In the first-order setting we carry out a more sophisticated form of this preprocessing: we convert the input formula into what is known as (set) *clause normal form*. This preprocessing accomplishes two things. First, just as in the proposi-

tional case, the input formula is converted to conjunctive normal form. Second, along the way all the quantifiers are eliminated.

Given an input formula, we convert it to clause normal form by carrying out the following four steps:

1. Put the formula into negation normal form (NNF).
2. Skolemise away any existential quantifiers.
3. Discard any universal quantifiers.
4. Put the resulting (quantifier-free) formula into (set) conjunctive normal form.

Let's take a closer look at each step. First of all, we learned in the previous chapter how to convert a propositional formula to NNF: simply apply the following rules:

Rewrite $\neg(\phi \wedge \psi)$ as $\neg\phi \vee \neg\psi$

Rewrite $\neg(\phi \vee \psi)$ as $\neg\phi \wedge \neg\psi$

Rewrite $\neg(\phi \rightarrow \psi)$ as $\phi \wedge \neg\psi$

Rewrite $\phi \rightarrow \psi$ as $\neg\phi \vee \psi$

Rewrite $\neg\neg\phi$ as ϕ .

Clearly these rules are needed for first-order formulas too. In addition, however, we will need the following two rules which let us push negations inwards past quantifiers:

Rewrite $\neg\forall x\phi$ as $\exists x\neg\phi$

Rewrite $\neg\exists x\phi$ as $\forall x\neg\phi$.

That's all that needs to be said about the reduction of first-order formulas to NNF; it's a routine extension of the propositional case. The next step, however, is far more interesting: this is where we skolemise away the existential quantifiers.

In fact, we have already used Skolem functions and Skolem terms to eliminate existential quantifiers (recall we used them in free-variable tableaus to build structured terms that would block erroneous unifications), but the way we are going to use them now is even more simple and direct. Model theoretically, skolemisation hinges on the following insight: for every first-order formula ϕ containing existential quantifiers, there is a formula ϕ^s that contains no existential quantifiers such that ϕ is satisfiable if and only if ϕ^s is too. To put it informally, we can always convert a formula ϕ to a formula ϕ^s that contains no existential quantifiers without doing semantic damage.

Consider the formula $\exists xP(x)$. It's easy to skolemise here: choose a new constant s and form $P(s)$. Clearly, if one of these formulas is satisfiable, then so is the other. Or consider the formula $M(a) \rightarrow \exists x\exists yR(x,y)$.

Again it's easy to skolemise: simply choose new constants s and c and form $M(a) \rightarrow R(s, c)$. Once more, it should be clear that if one of these formulas is satisfiable then so is the other.

Fine, but these examples are rather too easy. The interesting question is: how do we skolemise away existential quantifiers when they are under the scope of universal quantifiers?

As follows. When we skolemise away an existential quantifier that is under the scope of universal quantifiers $\forall x_1, \dots, \forall x_n$, then instead of using simple Skolem constants (like s or c) we should use a Skolem term of the form $s(x_1, \dots, x_n)$. Intuitively, such a term says that the value we need assign to the existentially quantified variable y to satisfy the formula will in general depend on the values we have assigned to the universally quantified variables x_1, \dots, x_n .

For example, consider the formula $\forall x \exists y R(x, y)$. According to what we just said, $\forall x R(x, s(x))$ should be a suitable skolemisation. And a moment's thought shows that this makes sense. For a start, if $\forall x R(x, s(x))$ is true in some model, then $\forall x \exists y R(x, y)$ is true in that same model too. On the other hand, suppose that $\forall x \exists y R(x, y)$ is true in some model. Now, this formula says that for each x there is some y such $R(x, y)$, but this means that it is possible to define a function that maps each element in the model to an R -related element. But the fact that such a function exists is precisely what the formula $\forall x R(x, s(x))$ claims.

Here's a slightly more complex example. Consider the formula

$$\forall x \forall y (R(x, y) \rightarrow \exists z (R(x, z) \wedge R(z, y))).$$

Now, in this example the existential quantifier is under the scope of the quantifiers $\forall x$ and $\forall y$. Hence, given what we said above, we can skolemise as follows:

$$\forall x \forall y (R(x, y) \rightarrow R(x, s(x, y)) \wedge R(s(x, y), y)).$$

We leave the reader to verify that this formula is indeed satisfiable if and only if the original existentially quantified version is too.

So much for skolemisation—what about the next step, dropping the universal quantifiers? This may sound a little drastic; what could justify such a move? Actually, it is rather simple. By this stage, the formulas we are preprocessing are in a rather restricted form: all negations occur next to atomic symbols, and the only boolean connectives are \wedge and \vee . Here's a typical example:

$$\forall x M(x, s(x)) \wedge \forall z \forall w (\neg M(z, w) \vee N(z)) \wedge \neg N(c).$$

Now, suppose we move all the universal quantifiers to the front of the formula. That is, consider the formula:

$$\forall x \forall z \forall w (M(x, s(x)) \wedge (\neg M(z, w) \vee N(z)) \wedge \neg N(c)).$$

It is clear that this is logically equivalent to the preceding one. More generally, given any formula in this restricted form, we can move all the universal quantifiers it contains to the front of the formula without changing anything semantically. But given this, there is then little point in actually explicitly writing the universal quantifiers. We might as well just make the convention that all variables are to be taken as universally quantified—that is, we drop the quantifiers, thus making the universal quantification an implicit part of the notation.

What about the last step, the conversion to conjunctive normal form? Here again, we are back on familiar territory: we simply do what we did in the propositional case (that is, apply the distributive laws and so on). Nothing more needs to be said here.

Well, that's the algorithm. Let's work through an example. Let's put

$$\neg((\forall x \exists y R(x,y) \wedge \forall z \forall w (R(z,w) \rightarrow P(z))) \rightarrow \forall u P(u))$$

in clause normal form.

The first step is to convert this formula to NNF. We can do this by using the rewriting rules given earlier:

$$\text{Step 1 } \neg((\forall x \exists y R(x,y) \wedge \forall z \forall w (R(z,w) \rightarrow P(z))) \rightarrow \forall u P(u)).$$

$$\text{Step 2 } \forall x \exists y R(x,y) \wedge \forall z \forall w (R(z,w) \rightarrow P(z)) \wedge \neg \forall u P(u).$$

$$\text{Step 3 } \forall x \exists y R(x,y) \wedge \forall z \forall w (R(z,w) \rightarrow P(z)) \wedge \exists u \neg P(u).$$

$$\text{Step 4 } \forall x \exists y R(x,y) \wedge \forall z \forall w (\neg R(z,w) \vee P(z)) \wedge \exists u \neg P(u).$$

Next, we skolemise away the existential quantifiers:

$$\text{Step 5 } \forall x R(x, s(x)) \wedge \forall z \forall w (\neg R(z,w) \vee P(z)) \wedge \neg P(c).$$

Now it's time to drop the universal quantifiers:

$$\text{Step 7 } R(x, s(x)) \wedge (\neg R(z,w) \vee P(z)) \wedge \neg P(c).$$

This formula is already in conjunctive normal form, so we have no further work to do. But to finish up neatly, let's put it in our list of lists notation:

$$\text{Step 8 } [[R(x, s(x))], [\neg R(z,w), P(z)], [\neg P(c)]].$$

And that's all there is to the preprocessing. Time to turn to the resolution phase proper.

The resolution phase

After the conversion of the input formula to clause normal form, we are ready to begin first-order resolution. As we promised earlier, this

pretty much boils down to repeated application of the resolution rule we learned about in the previous chapter, coupled with unification. Let's consider an example.

Consider the following two clauses:

$$[P(x), Q(x)] \text{ and } [\neg P(a), R(z)].$$

It should be intuitively clear that parts of these two clauses disagree with each. Recall the convention that in clause normal form all variables are implicitly universally quantified. Thus the $P(x)$ in the left-hand clause is really making the claim that $\forall x P(x)$. On the other hand, in the right-hand clause we have the claim that $\neg P(a)$. However, we can't apply the resolution rule to exploit this disagreement: the resolution rule demands a complementary pair of literals (that is, a pair of literals where one is the negation of the other) and $P(x)$ and $\neg P(a)$ do not have precisely this form.

But of course, they *almost* have this form, and if we unified x with a we would have the complementary pair of literals we required. So let's do this. Unifying x with a yields

$$[P(a), Q(a)] \text{ and } [\neg P(a), R(z)].$$

Applying the resolution rule then gives us

$$[Q(a), R(z)].$$

In short, by first unifying and then resolving we can get our hands on the complementary literal that drive the resolution process.

Sometimes we need to relabel variables before we unify. For example, consider the following pair of clauses.

$$[P(x), Q(x)] \text{ and } [\neg P(a), R(x)].$$

As in the previous example, if we unified a with x we could then apply the propositional resolution rule. Note, however, that in this case unification would also affect the x in $R(x)$, which we *don't* want (the two clauses are independent pieces of information, and unification needs to respect this). So we first relabel this variable, thus obtaining

$$[P(x), Q(x)] \text{ and } [\neg P(a), R(y)].$$

We can now unify a with x , obtaining:

$$[P(a), Q(a)] \text{ and } [\neg P(a), R(y)].$$

Resolution then yields:

$$[Q(a), R(y)].$$

In fact, when implementing resolution in Prolog we won't have to worry about this. As we use Prolog variables to represent first-order variables, Prolog will take care of the variable book-keeping automatically.

And that's the basic idea of first-order resolution. However, there is one complication we need to examine in more detail: we need to know about *non-redundant factors*.

A non-redundant factor F of a clause C is a clause obtained by applying unification to the literals *within* C as often as possible. That is, a non-redundant factor of C identifies as many literals as possible within C by unification. It is crucial to note (see the following example) that a clause may have more than one non-redundant factor, and we need them all. Incidentally, if none of the literals within a clause C can be unified, then C itself is its own unique non-redundant factor.

Why are non-redundant factors important? Actually, it's related to something we saw in the propositional case. Recall that when performing propositional resolution we need to work with clauses that are *sets*. For example, if we have the non-set clauses $[p, p]$ and $[\neg p, \neg p]$, resolution yields $[p, \neg p]$ and then we are stuck. Our input clauses were too fat: we should first thin them down to sets (here $[p]$ and $[\neg p]$) and only then apply resolution (which immediately yields the empty clause).

Now, in the first-order case, we must continue to work with set clauses just as before—but that alone is not enough. So to speak, a first-order clause can be too fat not merely because it contains repeated items, but because it contains items that can be identified by unification. In a nutshell, non-redundant factors of a clause are the clauses produced by thinning out via unification.

Let's look at an example. In what follows, A is a one-place predicate, B is a two-place predicate, w, x, y , and z are variables, m and n are constants, and f is a one-place function symbol.

Consider the following clause

$$[A(m), A(y), B(n, x), B(y, z), \neg C(w), \neg C(f(z))].$$

Note that we can apply unifications to the literals within this clause, thereby simplifying it. Here's one way of doing it: unify $A(y)$ with $A(m)$, and unify $\neg C(w)$ with $\neg C(f(z))$. We then throw away the repeated items (recall that clauses must be thought of as *sets*, that is with no repeated information), to get the following clause:

$$[A(m), B(n, x), B(m, z), \neg C(f(z))].$$

Note that we cannot reduce this clause anymore by unifying the literals it contains. This clause is a *non-redundant factor* of the original clause. The unifications have got rid of all the unnecessary information. However, note there was another way we could have reduced the original clause. We could have unified $B(n, x)$ with $B(y, z)$, and $\neg C(w)$ with

$\neg C(f(z))$ to get the following clause:

$$[A(m), A(n), B(n, x), \neg C(f(x))].$$

Once again, we cannot reduce this clause anymore by unifying the literals it contains. This clause is also a non-redundant factor of the original clause. The unifications have got rid of all unnecessary information, but this time in a different way.

Note that there are no more non-redundant factors of the original clause: the two we have just calculated are (up to α -equivalence, which is all we care about) the only two factors. And we need to hold onto *both* these factors, or sometimes we will not find proofs.

A first-order theorem prover should have access to the non-redundant factors of the clauses it is working with. So at the start of the proof, we won't simply convert the input formula into a set of clauses—once we've done the conversion to clause form we must immediately calculate the non-redundant factor clauses and add these as well. Moreover, whenever we apply the resolution rule, we don't simply return the resulting clause—we also calculate its non-redundant factors and add these too.

5.6 Implementing First-Order Resolution

It is extremely straightforward to turn the approach to first-order resolution sketched above into Prolog code. Unsurprisingly, the work divides naturally into two steps. First, we extend the code (from the previous chapter) for converting propositional formulas to conjunctive normal form so that it handles the conversion of first-order formulas to clause normal form. We then combine our propositional resolution code with our unification predicates, thus obtaining a resolution theorem prover for first-order logic.

Reduction to clause form

Here's the main clause of our program for converting first-order formulas to clause form:

```
cnf(Formula,SetCNF) :-
    nnf(Formula,NNF),
    skolemise(NNF,Skolemised,[]),
    cnf([[Skolemised]],[],CNF),
    setCnf(CNF,SetCNF).
```

Save for the call to the `skolemise/3` predicate, this is exactly the same as the propositional case. So the next question is: how is `skolemise/3` defined? Here are the clauses for universally quantified

formulas:

```
skolemise(all(X,F),N,Vars) :-  
    skolemise(F,N,[X|Vars]).
```

Recall that when skolemising we need to know which universal quantifiers outscope the existential quantifier we are eliminating. Hence, as we recursively strip off the universal quantifiers, we take care to push the variables they bind onto the list `Vars`. This keeps track of the way the universal quantifiers are embedded one within another, and the clause for existentially quantified formulas puts this information to good use:

```
skolemise(some(X,F1),N,Vars) :-  
    skolemFunction(Vars,SkolemTerm),  
    substitute(SkolemTerm,X,F1,F2),  
    skolemise(F2,N,Vars).
```

We've met `skolemFunction/2` before; we used it in the code for the free-variable tableau prover. This predicate simply makes a Skolem function that has all the listed variables as arguments (that is, the variables in `Vars`). We then hand the result to `substitute/4` to build the skolemised formula (recall that `substitute/4` is one of the predicates in `comsemPredicates.pl`) and carry on recursively skolemising.

Well, those were the two key clauses. The remaining ones merely complete the recursion in the expected way:

```
skolemise(and(F1,F2),and(N1,N2),Vars) :-  
    skolemise(F1,N1,Vars),  
    skolemise(F2,N2,Vars).
```

```
skolemise(or(F1,F2),or(N1,N2),Vars) :-  
    skolemise(F1,N1,Vars),  
    skolemise(F2,N2,Vars).
```

```
skolemise(F,F,_) :-  
    literal(F).
```

Now that we've dealt with `skolemise/3` we've covered the main change required to adapt our propositional conjunctive normal form converter to a first-order clause normal form converter. The only other change worth mentioning is that we have also to add the following clauses to the definition of `nnf/2` to enable negations to be driven inwards past quantifiers:

```
nnf(not(all(X,F)),some(X,N)) :-  
    nnf(not(F),N).
```

```

nnf(all(X,F),all(X,N)):-  

  nnf(F,N).  
  

nnf(not(some(X,F)),all(X,N)):-  

  nnf(not(F),N).  
  

nnf(some(X,F),some(X,N)):-  

  nnf(F,N).

```

Apart from these differences, the code is pretty much identical to that used in the propositional case.

Implementing the resolution phase

When it comes to implementing the resolution phase, matters are even more pleasant. Remarkably little needs to be done to our propositional code to make it work for full first-order logic.

Here's the main clause of the program:

```
rprove(Formula):-  

  cnf(not(Formula),CNF),  

  nonRedundantFactors(CNF,NRF),  

  refute(NRF).
```

The only difference between this and the analogous clause in the propositional prover is the presence of the call to `nonRedundantFactors/2`.

Another clause that must be changed is `resolve/3`. Here's the new version:

```

resolve(Clause1,Clause2,NewClause):-  

  selectFromList(Lit1,Clause1,Temp1),  

  selectFromList(not(Lit2),Clause2,Temp2),  

  unify_with_occurs_check(Lit1,Lit2),  

  unionSets(Temp1,Temp2,NewClause).  
  

resolve(Clause1,Clause2,NewClause):-  

  selectFromList(not(Lit1),Clause1,Temp1),  

  selectFromList(Lit2,Clause2,Temp2),  

  unify_with_occurs_check(Lit1,Lit2),  

  unionSets(Temp1,Temp2,NewClause).

```

Once again, the change involved is minimal: this is just the propositional code, with the calls to `unify_with_occurs_check/2` added. But though this change is simple, it is crucial: this is where we combine applications of the resolution rule with unification.

In fact, there aren't really any complex changes to the code. Yes, we need to define `nonRedundantFactors/2`, but this is straightforward. We go through the list of clauses, compute the non-redundant factors

for that clause (using a subsidiary predicate `nonRedFact/2`), and then append the results into one big list of clauses:

```
nonRedundantFactors([],[]).

nonRedundantFactors([C1|L1],L4) :-
    findall(C2,nonRedFact(C1,C2),L3),
    nonRedundantFactors(L1,L2),
    appendLists(L3,L2,L4).
```

For the definition of the predicate `nonRedFact/2` we need to take some care. Once again, we can't use Prolog's standard unification, and therefore we'll use unification with the occurs check to check whether two literals in the clause can be unified. This is coded as follows:

```
nonRedFact([],[]).

nonRedFact([X|C1],C2) :-
    memberList(Y,C1),
    unify_with_occurs_check(X,Y),
    nonRedFact(C1,C2).

nonRedFact([X|C1],[X|C2]) :-
    nonRedFact(C1,C2).
```

And that's all there is to it. We have gone from a propositional to a first-order resolution prover and hardly noticed the transition!

Exercise 5.6.1 The file `foResolution.pl` is the main file for the implementation of the first-order resolution prover. Use the `rproveTestSuite/0` predicate to test the prover on the test suite.

Exercise 5.6.2 Test the `nonRedundantFactors/2` predicate. For example, test it on a list containing the single clause:

```
[[a(m),a(Y),b(n,X),b(Y,Z),not(c(W)),not(c(f(Z)))]].
```

Explain the result you get. Then try the next exercise.

Exercise 5.6.3 If you have tried the previous exercise, you will have noticed that `nonRedundantFactors/2` over-generates new clauses. This is no surprise given its definition! Here is a suggestion to improve it by including a call to a predicate that discards clauses that are *subsumed* by others. (A clause C_1 subsumes a clause C_2 if and only if there is a substitution σ such that $C_1\sigma = C_2$. We say that C_2 is subsumed by C_1 .)

```
nonRedundantFactors([],[]).

nonRedundantFactors([C1|L1],L5) :-
    findall(C2,nonRedFact(C1,C2),L3),
    nonRedundantFactors(L1,L4),
    subsume(C1,L4,L5).
```

```
nonRedundantFactors(L1,L2),
appendLists(L3,L2,L4),
subsume(L4,L5).
```

Implement a subsumption check by giving a definition for `subsume/2`. Ensure you use unification with the occurs check.

Exercise 5.6.4 Our resolution prover contains no mechanism for preventing non-terminating proof search (recall that our tableau prover made use of Q-depth to prevent this). Add a mechanism that guarantees termination.

Programs for First-order Resolution

`foResolution.pl`

The file that contains the code for the first-order resolution prover.

`cnfFOL.pl`

Definition of translation to clause normal form.

`folTestSuite.pl`

A test suite with first-order formulas (non-theorems as well as theorems).

5.7 Off-the-Shelf Theorem Provers

By rights, we ought to be at the end of the chapter. We've seen why naive approaches to first-order theorem proving (notably, the simple and intuitive tableau system we started with) are inadequate from a computational perspective, and understand why the use of unification to guide the process of instantiating variables is important. Building on this insight, we've adapted our propositional provers to deal with first-order logic.

But pleasant though this is, it's nowhere near enough. Our goal is to develop useful computational tools that will help us get to grips with the consistency and informativity checking tasks. Now, our first-order tableau and resolution provers are certainly computational tools, but are they truly *useful*? The only honest answer is: *only for very simple problems*. Although both provers use the basic technique needed to make first-order theorem proving at all feasible (namely, unification)

they don't incorporate any of the sophisticated optimisations needed to make first-order theorem provers genuinely practical.

The following example should dispel any illusions the reader may have. Here is a well-known problem from the 1980s called Schubert's Steamroller:

Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. There are also some grains, and grains are plants. Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore, there is an animal that likes to eat a grain-eating animal.

This discourse expresses a logically valid argument. That is, if the first six sentences are true, then the seventh sentence ("Therefore, there is an animal that likes to eat a grain-eating animal") must be true too. In short, the seventh sentence is a logical consequence of the first six and hence (once we have represented these sentences in first-order logic) a first-order theorem prover ought to be able to prove this.

Now, in the first-order test suite you will find this problem transcribed into our first-order notation. Try both the tableau and the resolution provers on this problem. As you will see, neither can handle it. Both provers embark on a lengthy proof search—and neither comes back. You will find yourself looking at a computer terminal that seems to have gone to sleep, and eventually you will have to give up and abort execution. Both provers have been steamrollered.

In short, one problem with our provers is *efficiency*: they are far too easy to crush. Moreover, that's not all that's wrong with them. There is also a second problem: neither handles equality. For natural language applications, this is a severe limitation. Even the limited grammar we use in this book gives rise to semantic representations containing the = symbol. For example, the semantic representation we build for Vincent is not Butch is

```
not(eq(vincent, butch)).
```

And if we developed the grammar to cover phenomena such as pronominal anaphora (where we might need to say that the entity denoted by some pronoun is the same as some other previously mentioned entity) we would swiftly find ourselves making even heavier use of the equality symbol.

Now, it is worth reminding the reader that equality really is a special

symbol. That is, as we said in Chapter 1, although syntactically $=$ is just a binary relation symbol, semantically it is special: it is always used to talk about the relation of equality between individuals in a model.

What are the consequences of this for theorem proving? The key point is that the equality symbol legitimises certain additional inferences. To give a simple example, given that $x = y$ and $y = z$ it is legitimate to infer that $x = z$. Note that we *cannot* make the analogous inference for arbitrary binary relation symbols R . Given that Rxy and Ryz it does *not* follow that Rxz . To see this, note if R is interpreted in some model as the relation “stands immediately to the right of”, and three distinct individuals are assigned to the variables x , y and z , then this inference would be incorrect.

In short, because the equality symbol is interpreted in the same special way in each model, it licenses additional patterns of inference. Hence if a first-order theorem prover does not contain additional mechanisms for coping with equality, it will *not* fully cope with its logic.

Now, it is not that difficult to devise additional rules (or other mechanisms) which enable equality to be dealt with by first-order theorem provers; see the Notes for references. However, if we use these rules naively (or worse, if we simply add axioms, as we ask the reader to do in Exercise 6.6.10 in the following chapter) the already sluggish performance of our provers will slow to a crawl. Integrating equality reasoning into first-order theorem proving *efficiently* is a non-trivial task, one well beyond the scope of this book.

All in all, it seems we have a serious problem on our hands: our theorem provers are not the useful inference tools we hoped they would be. So what are we to do? Actually, the answer is clear. The problem with both our provers is that they are naive: they don't handle ordinary first-order logic efficiently, and they can't be made to handle first-order logic with equality without a further degradation in performance. Very well then: if the problem is naivety, the solution is sophistication! Why don't we put some truly sophisticated provers to work instead?

Nowadays, many extremely sophisticated theorem provers are freely available on the internet. In the forty years since the pioneering work of Robinson (who introduced the resolution rule and the idea of unification) there has been intensive work on automated first-order theorem proving, and performance has improved dramatically. This improvement has not been driven simply by making use of more sophisticated implementation techniques (though such techniques have certainly played an important role), it has also been driven by the deeper insight that has been obtained into the mathematical structure of res-

olution and tableau proofs, coupled with the development of optimisation techniques to exploit these insights. (A discussion of such techniques lies well beyond the scope of this book, but the Notes contain pointers to references where the reader can find out more.) Moreover, over the last decade there has been a development of special relevance to natural language semantics: there have been substantial improvements in the ability of first-order theorem provers to handle equality reasoning efficiently. All in all, home-brewed theorem provers (like our two little toys) can't begin to achieve the high levels of performance that are nowadays regarded as commonplace by the automated reasoning community. Luckily, they don't have to. All this sophistication is out there on the internet, just waiting to be used.

So, let's put them to use. Now, many theorem provers are available; which should we choose? In this book we shall make use of the provers Otter and Bliksem. In the Notes we'll say a little more about why we picked these two provers, for now we'll just say that both provers are resolution based, both handle equality, and that the performance of both is in a completely different league to that of our two baby provers.

Of course, a practical issue immediately raises its head. In this book we have been constructing a (Prolog based) architecture for reference and inference in natural language. Suddenly we're proposing to add software produced elsewhere to this architecture. Is this so easy to do? As we shall now show, yes it really is. We are going to treat both Otter and Bliksem as black boxes. That is, we're simply going to devise interfaces that let us glue the capabilities they provide into our (Prolog based) architecture.

Both Otter and Bliksem use a different notation for first-order logic from the one used in this book (and indeed, they use different notations from each other). Moreover, they both work by reading a file, which contains certain initialisation options followed by a list of the formulas to be proved.

But the reader won't need to know about any of this: we shall simply provide a couple of small Prolog programs which take a first-order formula written in our notation, convert it to Otter or Bliksem-style notation, and simultaneously supply the initialisation information that Otter and Bliksem need. Here's a simple example. Consider the formula

$$\forall x \text{DANCE}(x) \rightarrow \neg \exists y \neg \text{DANCE}(y).$$

Now, our Prolog notation for this formula is

```
imp(all(X,dance(X)),not(some(Y,not(dance(Y))))).
```

This little formula is valid, and both our tableau and resolution-based

provers are quite capable of proving this. But suppose we wanted to use Otter or Bliksem to prove it. What would we have to do?

The first step is to get the formula in the appropriate notation and supply prover initialisation information. The Prolog predicates which do this for us are `fol2otter/2` and `fol2bliksem/2`. If we make the query

```
?- fol2otter(imp(all(X,dance(X)),not(some(Y,not(dance(Y))))),  
            user).
```

we get the result (note that we used `user` as second argument to specify that Prolog pipes the result to standard output):

```
set(auto).  
assign(max_seconds,30).  
clear(print_proofs).  
set(prolog_style_variables).  
formula_list(usable).  
((all A dance(A)) -> -(exists B -(dance(B)))).  
end_of_list.
```

That is, we get back a set of instructions which sets Otter in automatic mode, assigns it a maximum time of 30 seconds for the problem, and insists that it uses a Prolog-style representation of first-order variables. Following this comes the list of all the formulas that Otter has to prove and (needless to say) in this case there is only one, namely our input formula transcribed in Otter notation. As you can see from this example, Otter notation is very natural, and not too different from our own.

What about Bliksem? Well, if we make the query

```
?- fol2bliksem(imp(all(X,dance(X)),not(some(Y,not(dance(Y))))),  
               user).
```

we get the result

```
Auto.  
(( [ A ]dance(A)) -> !( < B >!dance(B))).
```

This is even simpler. We simply get an instruction that sets Bliksem in auto mode, followed by the input formula transcribed into Bliksem notation.

Actually, strictly speaking the reader does not even need to know about `fol2otter/2` and `fol2bliksem/2`, for the theorem provers Otter and Bliksem can be called directly by consulting `callInference.pl`. This loads the file `inferenceEngines.pl`, which is where we specify which theorem provers we would like to use. As we would like to work with Otter and Bliksem, we specify the following:

```
inferenceEngines([otter,bliksem]).
```

Now, `callInference.pl` offers several interface predicates to off-the-shelf provers, but the predicate that we are interested in for now is `callTP(Problem,Result,Prover)`, which succeeds if there is a theorem prover `Prover` (on the list of inference engines that we selected) that finds `Result` for the first-order formula `Problem` (where problem is written in the notation of this book). Here `Result` will be instantiated with the Prolog atom `proof` if indeed a proof is found, and with `unknown` otherwise.

For example, if we want to use it to prove the formula

$$\forall x \text{DANCE}(x) \rightarrow \neg \exists y \neg \text{DANCE}(y)$$

we would make the following query:

```
?- callTP(imp(all(X,dance(X)),not(some(Y,not(dance(Y))))),P,E).
```

This will give the problem both to Otter and Bliksem, and will return the result of the first prover that managed to find a solution for the problem. For instance, we could get the result

`P = proof`

`E = otter`

which tells us that Otter has succeeded in proving the input formula.

What happens if we give an invalid formula as input? For example, what happens if we give them $\forall x \text{DANCE}(x)$, which is clearly not true in all models? Let's try. Suppose we make the following query:

```
?- callTP(all(X,dance(X)),P,E).
```

As before, `callTP` will feed the input formula to all the theorem provers it knows about (namely Otter and Bliksem) and wait for a result. In this case we will get the result

`P = unknown`

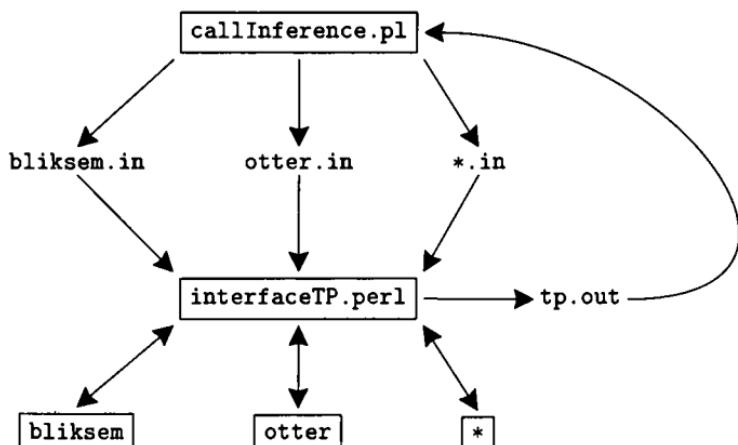
`E = unknown`

Both Otter and Bliksem will have tried (and failed) to prove this formula.

It is crucial that the reader understands that the answer `unknown` returned here really is the correct response. As it happens, we know that the input formula is *not* valid. But theorem provers don't prove invalidity, all they can try and do is prove validity. If a theorem prover fails to prove something, this can be for two quite distinct reasons. Sometimes, the formula may be valid, but extremely complex to prove, and in such cases the prover can easily use up all its pre-allocated resources (for example, the Q-depth resources in the case of our free-variable tableau prover, or some preset time limit) without finding a

proof (even though a proof *does* exist). Sometimes, however, as in the previous example, the formula really is not valid, hence no proof exists, and so the prover (quite reasonably) cannot find a proof. That's the trouble with undecidable formalisms like first-order logic: in general, if we don't get a result, we're left in the dark as to *why* we didn't get a result. In the following section we shall discuss a partial solution to this problem.

But before doing this, let's take a quick look at the architecture used in `callInference.pl` to control the theorem provers. You don't need an in-depth understanding of the code, but if you want to extend the system by adding new provers (as Exercise 5.7.1 asks you to do) you will need to know something of how it all fits together. The following diagram shows the key points:



As you can see, `callInference.pl` creates input files for the selected theorem provers (here Bliksem and Otter, and an unnamed prover `*`, a placeholder for the theorem provers we hope the reader will add). These files have the extension `.in`. Note that `callInference.pl` calls a Perl program

`interfaceTP.perl`

that starts the selected theorem provers with the created input files, reads the output, and then writes the result in Prolog format in the file `tp.out`. This file is then read by `callInference.pl` and the circle is closed.

Why do we use Perl here? For at least three reasons. First, our goal is to coordinate the input and output of various programs (namely the Prolog program `callInference.pl`, and Otter and Bliksem). Perl is

superb at this task (it's what is known as a *glue language*, or *scripting language*), so rather than make life miserable for ourselves by hacking it in Prolog, we used a language in which these kinds of task are easy. Second, as we have already mentioned, each inference engine uses its own internal format. Prolog is not much good at fiddling around with text which is not in Prolog format—Perl, on the other hand, is in its element with such tasks. Thirdly (and most interestingly) Perl allows us to simulate parallelism. When using Otter and Bliksem together, we don't want to first use one and then the other: we want to unleash both of them on the problem simultaneously (apart from anything else, it's interesting to see which one solves the problem first). Perl offers some basic tools for this kind of programming, and we've used them to build a rudimentary form of parallelism into our architecture. As we shall see, these parallelism capabilities will become very important in the following section when we introduce model builders into our inference architecture.

Time to play. If you load up the file `callInference.pl` you can experiment with Otter and Bliksem on various formulas. In fact, one of the options that is available is to run these theorem provers on the first-order test suite; this can be done by giving the command

```
?- tpTestSuite.
```

This runs all selected provers on all the problems in the test suite, including Schubert's Steamroller. Neither prover so much as blinks. In the 1980s the Steamroller was considered a difficult problem (and as we have seen it's still a tough nut for a naive prover to crack) but for sophisticated provers such as Otter or Bliksem it's a triviality. The Steamroller has been steamrollered.

Exercise 5.7.1 Extend the arsenal of inference engines with the theorem prover Spass. Download Spass from the web. Extend the Prolog code of `callInference.pl` and the Perl script `interfaceTP.perl` to include Spass. You will need a new module `fol2spass.pl` that converts first-order formulas in Prolog to the syntax used by Spass. Use `tpTestSuite/0` to test your work.

5.8 Model Building

Once again the reader might be forgiven for thinking that at last we have reached our goal. After all, we have learned a little about two theorem proving techniques, have seen that naive theorem provers are not efficient enough to cope with even relatively simple problems (such as the Steamroller), and have learned that sophisticated theorem provers such as Otter and Bliksem can be bolted onto our inference architec-

ture as black boxes. So haven't we accomplished what we set out to do? Don't we now have everything we need for coping with the consistency and informativity checking tasks for first-order logic with equality? No, we don't. And indeed, we never will. For now we must face up to a fundamental fact about first-order logic: it is undecidable.

Let's be a little more precise about what this actually means, and explore its ramifications for what we have set out to do in this chapter. When we say that first-order logic is undecidable, we mean the following: it is *impossible* to write a computer program that takes as input an arbitrary first-order formula, and that is guaranteed to halt after a finite amount of time and (correctly) tell us whether or not that formula is valid. This impossibility does not arise because of practical constraints (for example, the fact that physical computers have only a finite amount of memory) it is a consequence of *Church's Thesis*, the standard mathematical model of computation. That is, if the standard model of computation is correct (and most computer scientists believe it is) then even if we ignore all the practical constraints that a physical computer would be subject to, it will never be possible to build a computer that could correctly decide (in finite time) whether an arbitrary first-order formula is valid or not. Now, recall from Chapter 1 that both the consistency and informativity checking tasks are basically just reformulations of the validity checking task. It follows that both these tasks are undecidable too.

Put like that, this all sounds very abstract. However, as we shall now see, the limitations imposed by the undecidability of first-order logic are real, and have ramifications for what we are trying to do. Let's first think about what it means for the consistency checking task.

Suppose we are trying to check whether the last sentence of a discourse is consistent with the preceding sentences. For example, suppose we have the discourse

All boxers are slow. Butch is a boxer. Butch is not slow.

Obviously the last sentence is *not* consistent with what has gone before. But is there a computation we can perform that will automatically show this inconsistency? Yes, there is. We use our semantic construction machinery to build first-order representations for these sentences, and then check whether the conjunction of the first two sentences implies the negation of the last sentence. This means that we need to check whether the following formula is valid or not:

$$\forall x(\text{BOXER}(x) \rightarrow \text{SLOW}(x)) \wedge \text{BOXER}(\text{BUTCH}) \rightarrow \neg \text{SLOW}(\text{BUTCH}).$$

The final sentence of the discourse is inconsistent with the preceding discourse if and only if this formula is valid. Now, it is pretty clear

that this formula is valid. Moreover we have computational tools for showing first-order validity (namely theorem provers) so we simply put a prover to work and—hey presto!—we've automatically demonstrated the inconsistency of this discourse.

So what's the problem? If we change the example, this will quickly become clear. Suppose our discourse was

All boxers are slow. Butch is a boxer. Mia likes Butch.

In this example, the last sentence clearly *is* consistent with what went before. But is there a computation we can perform that will automatically show this consistency? Let's think this through. As before, we can use our semantic construction machinery to build first-order representations for these sentences, and then check whether the conjunction of the first two sentences implies the negation of the last sentence. This means we need to check the following formula for validity:

$$\forall x(\text{BOXER}(x) \rightarrow \text{SLOW}(x)) \wedge \text{BOXER}(\text{BUTCH}) \rightarrow \neg \text{LIKE}(\text{MIA}, \text{BUTCH}).$$

As before, the final sentence of the discourse is inconsistent with the preceding discourse if and only if this formula is valid. Now, here it is clear that this formula is *not* valid. And now we have a real problem. First-order theorem provers are tools for demonstrating validity; they don't show non-validity. Worse, no general computational tool for determining non-validity exists.

To put it another way:

Theorem provers provide us with a negative check for consistency.

That is, as our first example shows, if a discourse is inconsistent, then (assuming the problem is not too hard for the prover) a decent first-order theorem prover can demonstrate the inconsistency simply by proving a theorem. However, as our second example shows, theorem proving does *not* provide us with a positive test for consistency, and indeed, because of the undecidability of first-order logic no full positive check exists. As we remarked at the end of the previous chapter, this is very different from what happens in propositional logic. Propositional logic is decidable and thus (at least, in principle) theorem proving provides a complete solution to the propositional consistency checking task. That is, in the propositional case, theorem proving provides both negative and positive checks for consistency.

Undecidability also haunts the informativity checking task. Suppose we are trying to check whether the last sentence of a discourse is informative with respect to the preceding discourse. For example, suppose we have the discourse

All boxers are crazy. Butch is a boxer. Butch is crazy.

Obviously the last sentence is *not* informative with what has gone before. Equally obviously, there is a computation we can perform that will show this lack of informativity: use our semantic construction machinery to build first-order representations for these sentences, and then check whether the conjunction of the first two sentences implies the last. That is, we simply need to check the validity of

$$\forall x(\text{BOXER}(x) \rightarrow \text{CRAZY}(x)) \wedge \text{BOXER}(\text{BUTCH}) \rightarrow \text{CRAZY}(\text{BUTCH}).$$

Clearly this formula is valid, and even a baby-prover could show this, so there is no problem in computing the failure of informativity here.

But what happens if the discourse is informative? For example, consider the discourse

All boxers are crazy. Butch is a boxer. Butch loves Fabian.

Here the last sentence clearly *is* informative with respect to what came before. So the inference task that faces us is to determine whether or not the following formula is valid:

$$\forall x(\text{BOXER}(x) \rightarrow \text{CRAZY}(x)) \wedge \text{BOXER}(\text{BUTCH}) \rightarrow \text{LOVES}(\text{BUTCH}, \text{FABIAN}).$$

Now, clearly this formula is *not* valid, and once again the same problem rears its ugly head: first-order theorem provers are tools for demonstrating validity; they don't show non-validity. Worse, they can't, as no general computational tool for determining non-validity exists.

To put it another way:

Theorem provers provide us with a negative check for informativity.

That is, as the third example shows, if a discourse is uninformative then a decent first-order theorem prover can demonstrate the uninformativity simply by proving a theorem. However, as the fourth example shows, theorem proving does *not* provide us with a positive test for informativity, and indeed, because of the undecidability of first-order logic no full positive check exists.

Once again, we have a problem. Moreover, this is a problem we cannot fully solve. First-order undecidability is a fact of life: full positive checks for consistency and informativity don't exist.

But this does not mean we are helpless. Granted, a full solution is not possible—but perhaps there are useful *partial* positive checks for consistency and informativity? And indeed, there are. The relevant tools are called *model builders*, a relatively new kind of automated reasoning tool.

What is a model builder? First off, *don't* confuse model building with model checking! Recall from Chapter 1 that model checkers are

tools for solving the querying task. That is, a model checker takes a formula and a model, and sees whether the formula is satisfied in that model or not.

A model builder does something far more difficult: it takes a formula, and tries to build a model that satisfies it. To put it another way, it takes a description, and tries to build a little picture of the world in which that description is true. It should be clear that this is likely to be a very difficult task indeed. For example, some satisfiable formulas are only satisfied in infinite models (recall Exercise 1.2.1). Now, no piece of software can be expected to build arbitrary infinite models, so it is clear that model builders won't always be able to build models even when models exist. Roughly speaking, model building is only capable of building (relatively small) finite models. Indeed, when using a model builder, you usually have to tell it to try and build a model with a certain domain size (say, 3 elements) or to try and build a model up to a certain domains size (for example, a model with at most 20 elements).

This may not sound too exciting—is the ability to do this really going to help us with the consistency and informativity checking tasks?

It is. What we would very much like to have are (partial) positive checks for consistency and informativity. And this can often be done with the help of small models. For example, consider again the following discourse

All boxers are slow. Butch is a boxer. Mia likes Butch.

As we pointed out above, the last sentence clearly is consistent with the first two. That is, the logical representation of the discourse

$$\forall x(\text{BOXER}(x) \rightarrow \text{SLOW}(x)) \wedge \text{BOXER}(\text{BUTCH}) \wedge \text{LIKE}(\text{MIA}, \text{BUTCH}),$$

really is consistent. And a model builder can show the consistency of this discourse in a very simple way. Suppose we ask a model builder to build a 2 element model for this discourse. Any half-way decent model builder will swiftly see that if it names one of the elements Mia, and the other Butch, and insists that Butch is a boxer, that Butch is slow, and that Butch is liked by Mia, then it has a model in which the above formula is true. Some model builders might decide to make Mia a boxer too (in which case they will also have to say that she is slow) and other model builders might decide that she is not a boxer (in which case it doesn't matter whether they decide to make her slow or not). But it really doesn't matter which model they build as far as establishing the consistency of the above discourse is concerned: all that matters is that they build at least one model. After all (recall the discussion of Chapter 1) a consistent formula is simply a formula that has at least one model.

We can sum up our discussion as follows:

Model builders provide us with a (partial) positive check for consistency.

Let's now return to our informativity example. We considered the following discourse

All boxers are crazy. Butch is a boxer. Butch loves Fabian.

Here the last sentence clearly *is* informative with respect to the first two. To put it another way, the following formula is *not* valid:

$$\forall x(\text{BOXER}(x) \rightarrow \text{CRAZY}(x)) \wedge \text{BOXER}(\text{BUTCH}) \rightarrow \text{LOVE}(\text{BUTCH}, \text{FABIAN}).$$

Now, if a formula is not valid, this means that its negation has at least one model. So we should ask a model builder to build a model for

$$\neg(\forall x(\text{BOXER}(x) \rightarrow \text{CRAZY}(x)) \wedge \text{BOXER}(\text{BUTCH}) \rightarrow \text{LOVE}(\text{BUTCH}, \text{FABIAN})),$$

or equivalently

$$\forall x(\text{BOXER}(x) \rightarrow \text{CRAZY}(x)) \wedge \text{BOXER}(\text{BUTCH}) \wedge \neg\text{LOVE}(\text{BUTCH}, \text{FABIAN}).$$

But this is easy: simply build a model of domain size two in which the element named Butch is a boxer and is crazy, and does not love the other element (named Fabian). Whether or not Fabian is also a boxer is unimportant, but if she is, she should be crazy too. Again, we don't really care which model is made: the crucial point is that the existence of a model for the (negated) formula shows that the original (un-negated) formula is not valid, and hence that the original discourse is informative.

Summing up:

Model builders provide us with a (partial) positive check for informativity.

So far our discussion has been completely theoretical. Let's turn it into computational reality. As we mentioned above, model building is a newer branch of automated reasoning than theorem proving, and there is not nearly such a wide range of model builders as theorem provers. But there are some interesting systems out there, and to demonstrate how model building can be used for inference in computational semantics, we have decided to work with the model builders Mace and Paradox.

As with our use of off-the-shelf theorem provers, we'll integrate Mace and Paradox into our inference architecture as black boxes. In fact, all we need to do is adapt the predicate `callInference.pl` to call model builders instead of theorem provers.

Let's begin with a few remarks about each of the model builders. Mace uses Otter syntax for first-order logic. However, we cannot simply re-use `fol2otter/2` to carry out the translation process as the initialisation options for the model builder are different. Accordingly, we have defined a separate predicate `fol2mace/2`. If we pose the query

```
?- fol2mace(some(X, and(man(X), all(Y, imp(woman(Y), love(X, Y))))), user).
```

we get

```
set(auto).
clear(print_proofs).
set(prolog_style_variables).
formula_list(usable).
(exists A (man(A) & (all B (woman(B) -> love(A, B))))..
end_of_list.
```

This is similar to the file built for Otter, but note that there is no time-out limit.

Paradox uses TPTP notation (this is the notation used in the *Conference on Automated Deduction* (CADE) automated reasoning competitions). If we pose the query:

```
?- fol2tptp(some(X, and(man(X), all(Y, imp(woman(Y), love(X, Y))))), user).
```

we get

```
input_formula(comsem, conjecture,
(? [A]: (man(A) & (! [B]: (woman(B) => love(A, B))))).
```

But, as with our theorem proving architecture, the reader doesn't really need to know about these translation predicates to make use of the system; grasping the overall architecture is more important. Let's take a closer look.

The architecture is essentially the same as the one we used for theorem provers. Once again, everything is controlled from the file `callInference.pl`. There are two differences worth noting. First, model builders construct finite models, and they typically do this by iteration (they try to build a model of domain size 1, and if that fails they try to build one of domain size 2, and so on). Most model builders allow us to specify a domain-size upper limit to avoid unnecessary labour, so when we call the model builders we need to provide domain size information. Second, we don't just want a binary value as answer; we really would like to get our hands on the constructed model (we'll put such models to good use in the following chapter when we meet Helpful Curt). Now, different model builders represent models in dif-

ferent ways, and none of them uses the notation of this book. Here's a model in the notation we introduced in Chapter 1:

```
model([d2,d1],[f(0,vincent,d1),
              f(1,snort,[d2,d1]),
              f(0,mia,d2),
              f(1,dance,[d1])])
```

Here's the same model in the Mace format (Mace has the option of returning models in Prolog format, and we use this):

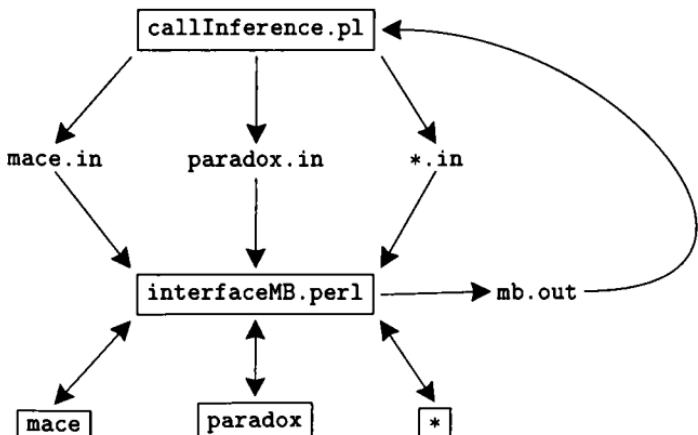
```
interpretation(2,[predicate(snort(_),[1,1]),
                  function(mia,[0]),
                  predicate(dance(_),[0,1]),
                  function(vincent,[1])]).
```

This representation tells us that the domain consists of two entities: 0 (Mia) and 1 (Vincent). The snort relation holds for both members of the domain, whereas the dance relations holds for the second member (1) only. And here's the same model in Paradox format (which is fairly self-explanatory):

```
-- Model -----
dance('1') : TRUE
dance('2') : FALSE
mia = '2
snort('1') : TRUE
snort('2') : TRUE
vincent = '1
```

So we'll need to be able to transform between the various model representations.

Here's the architecture we shall use:



The top-level file `callInference.pl` supplies us with a Prolog predicate `callMB(Problem,DomainSize,Model,ModelBuilder)` which succeeds if there is a model builder `ModelBuilder` that is able to construct a model `Model` with a maximum domain-size of `DomainSize` for the first-order formula `Problem`. If the model builder is not able to find a model for the input, `callMB/4` will unify `Model` with `unknown`.

As you can see, `callInference.pl` creates input files for the selected model builders. As with the theorem provers, the selection is made in the file `inferenceEngines.pl` (here we have selected Mace and Paradox, and an unnamed model builder *). Then a Perl program is called that activates the model builders with the created input files and the specified maximum domain-size, reads the output model, converts the model into our own Prolog notation, and then writes the result in the file `mb.out`. This file is then read by `callInference.pl`, and the circle is closed.

We are now ready to take the most interesting step of all. We want to provide the best computational handle possible on the consistency and informativity checking task. And now we have everything needed to do something useful. First, we know that theorem provers are capable of carrying out the *negative* checks we need. Secondly, we know that model builders are capable of carrying out at least some of the *positive* checks we need. Third, we have seen that our `callInference.pl` based architecture is capable of using different inference engines in parallel. Very well then—why not adapt our architecture so that it can simultaneously carry out both theorem proving and model checking on the input formula? To put it another way, why not adapt our architecture so that it is capable of carrying out both negative and (partial) positive checks in parallel?

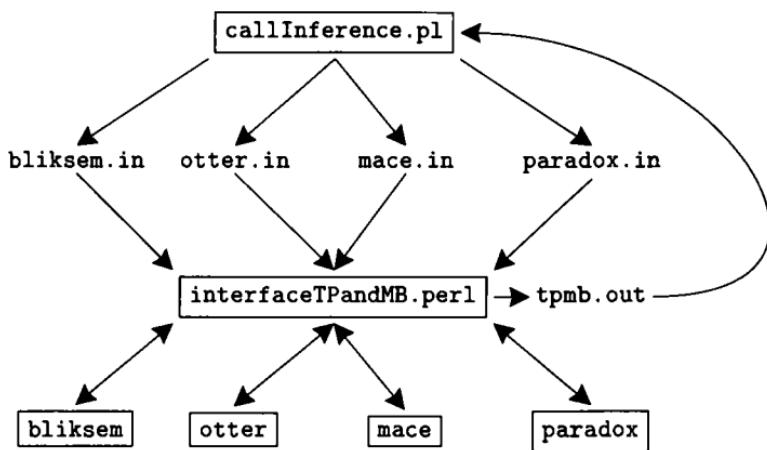
And this is exactly what we shall do. In `callInference.pl` you will find the following predicate:

`callTPandMB(TPProblem, MBProblem, Size, Proof, Model, Engine).`

This succeeds if *either* there is a model builder that is able to construct a model with a maximum domain-size of `DomainSize` for the first-order formula `MBProblem`, or there is a theorem prover that proves the formula `TPProblem`. (Incidentally, when we use this predicate in the following chapter, `TPProblem` will always be the negation of `MBProblem`.)

Depending on the result, `callTPandMB/6` will unify `Model` with a model if the model builder succeeds, or `Proof` with the atom `proof` if a theorem prover succeeded in finding a proof. Finally, `Engine` will be unified with the name of the successful model builder in case a model was found, or with the name of the successful theorem prover in case a

proof was found. Here is the diagram explaining the architecture behind this:



As you can see, we are simply making more interesting use of ideas we have discussed before. Now we really do have a useful computational handle on the consistency and informativity checking tasks—and, this time, any reader who feels that now we really *must* be at the end of the chapter, is right!

Exercise 5.8.1 This exercise shows you what the `interfaceTP.perl` Perl script does behind the scenes. Use `callInference.pl` and the `callTP/3` predicate to call a theorem prover (choose Bliksem). Then inspect the file `bliksem.in`. Run the shell command

```
bliksem < bliksem.in
```

and inspect the standard output. Finally, view the contents of the file `tp.out`. Do this for both a theorem and a non-theorem. Repeat the exercise choosing another theorem prover (for instance Otter).

Exercise 5.8.2 This exercise shows you what the `interfaceMB.perl` Perl script does behind the scenes. Use `callInference.pl` and the `callMB/4` predicate to call a model builder (choose Mace). Then inspect the file `mace.in`. Run the shell command

```
mace -t 30 -n1 -N10 -P < mace.in
```

and inspect the standard output. Finally, view the contents of the file `mb.out`. Do this for both a satisfiable and a non-satisfiable problem. Repeat the exercise choosing another model builder (for instance Paradox).

Exercise 5.8.3 Use `callInference.pl` to test the off-the-shelf theorem provers and model builders on the test suite `folTestSuite.pl`. Select in-

ference engines in the file `inferenceEngines.pl`. Use the `tpTestSuite/0`, `mbTestSuite/0`, `tpmbTestSuite/0`, predicates for your testing.

Programs for hooking up off-the-shelf inference tools

`callInference.pl`

The Prolog-interface to off-the-shelf theorem provers or model builders.

`fol2otter.pl`

Translates a formula in Otter syntax to standard output.

`fol2bliksem.pl`

Translates a formula in Bliksem syntax to standard output.

`fol2tptp.pl`

Translates a formula in TPTP syntax to standard output.

`callTP.perl`

A Perl script interfacing off-the-shelf theorem provers.

`callMB.perl`

A Perl script interfacing off-the-shelf model builders.

`callTPandMB.perl`

A Perl script interfacing off-the-shelf theorem provers and model builders.

`inferenceEngines.pl`

File that specifies the selected inference engines.

Notes

The human-oriented tableau system presented at the start of the chapter is a typical first-order signed tableau system; similar systems can be found in Smullyan (1995) (the ‘bible’ of the tableau method) and Sundholm (1983). For unsigned first-order tableau systems, see Bell and Machover (1977), Fitting (1996), Smullyan (1995), and Sundholm (1983). If you want to understand the tableau method in even more depth, consult the *Handbook of Tableau Methods* (D’Agostino et al..

1999). This contains a number of superb articles on many different aspects of tableau methods, both for first-order logic, and for a number of other logics too.

There are two articles in the Handbook we would particularly like to draw your attention to. The first is Letz (1999), which discusses in detail the various options available in first-order tableau systems; if you want to understand better the technicalities underlying free-variable tableau systems, this is the place to start. Secondly, for a fascinating practical perspective on first-order tableau theorem proving, see Posegga and Schmitt (1999). This describes a Prolog-based theorem prover for first-order logic called LeanTAP; the prover is coded in only 15 lines of Prolog and is more efficient than the free-variable tableau system in the text. If you are interested in experimenting further with tableau methods, we suggest you look at LeanTAP and try integrating it into the inference architecture of this book.

Unification and the unification algorithm were introduced in Robinson (1965), who introduced it as part of his pioneering work on first-order resolution. The algorithm discussed in the text is essentially Robinson's original. The literature on unification is huge, and a wide variety of unification algorithms have been investigated. We suggest that the reader who wants to know more try Fitting (1996) or Apt (1995). Fitting's discussion is very clear, and should be accessible to most readers of this book. Apt's discussion is more advanced, but still very approachable. He gives a nice account of idempotent substitutions, and analyses three unification algorithms, including the non-deterministic Robinson algorithm presented here. This is also a good source for further pointers to the literature.

Incidentally, recall that in the text we did not explain how the occurs check could be added to Prolog. Rather, we simply made use of the standard predicate

`unify_with_occurs_check/2`

and invited the reader in Exercise 5.2.2 to try and figure out how to implement it. If you had problems with this, you can find a complete discussion in Sterling and Shapiro (1986) and Fitting (1996).

In essence, the free-variable proof system presented in the text is a signed version of the unsigned free-variable tableau system presented in Fitting (1996). By and large, to extend our propositional implementation to first-order logic, we have simply followed the strategies used by Fitting. In particular, our use of `Qdepth`, and the subsequent decision to use `appendLists/3` to 'rotate' branches of tableaus, and universal signed formulas on branches, follows Fitting. (Incidentally, this tech-

nique has a name. We are essentially treating both tableau and branches as *priority queues*. The idea of doing this in tableau proofs dates back to Smullyan 1995.) The implementation of `skolemFunction/2` is taken from Fitting, and the `notatedFormula/3` predicate is Fitting's `notation/2` and `fmla/2` predicates rolled into one. Fitting's book is also a good source for learning the basics of equality reasoning with free variable tableau, and he gives a Prolog implementation. For an advanced discussion of equality reasoning in tableau (and other closely related proof systems) see Degtyarev and Voronkov (2001).

Our discussion of first-order resolution (as with our discussion of propositional resolution) is a simplified version of the approach presented in Leitsch (1997). Our discussion was informal and example-driven: the full details are painstakingly spelled out in Leitsch's book, which we strongly recommend. That said, some readers are likely to find Leitsch's discussion too technical. For a more accessible introduction, try Chang and Lee (1973); although dated, it still provides a clear exposition of the basic ideas of first-order resolution proving and related material. Another useful source is, once again, Fitting (1996). Fitting does not present resolution as a two-step process, but interleaves the reduction to normal form with applications of the resolution rule (this technique can be important when dealing with other logics, for example, various forms of modal logic). Fitting's approach also makes it clear that there is a closer relationship between the tableau and resolution methods than is at first sight apparent. Both Fitting and Chang and Lee discuss the basics of equality reasoning in resolution systems. For advanced material on this topic, see Nieuwenhuis and Rubio (2001).

As most readers of this book are probably aware, Prolog interpreters are in essence resolution theorem provers. But some readers may be puzzled by this: after all, in this chapter we have emphasized just how hard first-order theorem proving can be, and yet Prolog (which can give rise to highly efficient programs) is based on this idea. How can this be? The key point to observe is that Prolog only allows us to write a rather restricted form of first-order formulas, namely Horn clauses. When a set of Horn clauses (in effect, a Prolog program) is put into CNF, every clause will contain at most one negated literal. This vastly cuts down the number of possible resolvents and makes efficient implementations possible. For a simple discussion of the relation between Prolog and first-order logic, see Chapter 10 of Clocksin and Mellish (1984).

Jeff Pelletier (personal communication) tells the story behind Schubert's Steamroller this way. The Steamroller is named after Len Schubert, who designed it as a classroom exercise in transcribing English arguments into first-order logic. In 1982, Jeff Pelletier, then one of Schu-

bert's students, noticed that not only could his own theorem prover not handle it, but that neither could a number of (at that time) state-of-the-art first-order theorem provers. Pelletier refined the problem, and discussed it in his Masters thesis (see Pelletier (1982)) and in a later journal article (see Pelletier (1986)). A number of variant forms of the Steamroller can be found in the literature.

However, as we said in the text, nowadays the steamroller is no longer regarded as a particularly hard problem; this is largely because of the extraordinary improvements in efficiency that have been made over the last decade or so. The key source for further information on these developments is the two volume *Handbook of Automated Reasoning* (Robinson and Voronkov, 2001a), (Robinson and Voronkov, 2001b). This covers in depth all aspects of automated reasoning the reader is likely to need, and a great deal more besides.

To demonstrate how to integrate sophisticated off-the-shelf provers into an architecture for computational semantics, we chose the provers Otter and Bliksem. Why these two? Actually, for fairly down to earth reasons. Otter, which was written by William McCune, is rather old as theorem provers go (it was first written in the late 1980s) and probably can't compete with the newer generation of provers. Nonetheless, it is an excellent prover to work with, and is probably the system with the best claim to being that mythical beast, the 'standard' theorem prover. Otter is portable, compact, and well-documented, and there are books devoted to using Otter to tackle interesting problems (see Wos and Pieper (2000) and Kalman (2001)). Moreover, the Mace model builder used in the text comes along with the standard Otter distribution. And, last but not least, it should be emphasized that Otter is a very serious prover indeed—it has been used to solve open mathematical problems in a number of fields. What about Bliksem? This prover, written by Hans de Nivelle, is newer and (at least for the problems we encountered while writing this book) appears to be faster than Otter. We chose it because it is a relatively small, fast, easy to install prover.

We said nothing in the text about how model builders work. This was not merely because of lack of space, it was also because (in comparison with theorem proving) model building is a relatively new branch of automated reasoning and it is probably premature to talk about 'standard' approaches here. Nonetheless, a few words on the topic may be helpful. Alert readers will have noticed that a tableau system can be regarded as a kind of model builder: after all, at least in principle, to build a model for ϕ we should simply be able to give $T(\phi)$ to the tableau system and then 'read-off' possible models from any open branches the tableau system produces. Theoretically at least, this ob-

servation is correct. However the use of free-variable tableaus obscures the model building intuition that drives the tableau method, and at the time of writing we were not aware of any sophisticated model building systems for first-order logic that worked this way. It is also possible to use a form of resolution (called hyperresolution) to perform model building; for a discussion of this approach, see Fermüller et al. (2001).

The two model builders we used in the text, Mace (written by William McCune) and Paradox (written by Koen Claessen and Niklas Sörensson) use what is sometimes called a Mace-style strategy. In this approach the model builder takes a set of first-order clauses, and a number indicating the desired domain size, and transforms it into a set of *propositional* clauses. (Don't be surprised that this can be done. After all, if we are trying to build a model with only two elements of the domain, then a universal formula such as $\forall x P(x)$ is simply the conjunction $P(a) \wedge P(b)$, where a and b name the two elements of the domain, and this is a propositional formula.) Once the clause set has been converted to propositional form, a SAT solver (these were mentioned in the Notes to Chapter 4) is used to build the required model. Needless to say, the sketch just given vastly oversimplifies a highly demanding computational problem; for example, clever techniques are needed to avoid building isomorphic models many times over. For a discussion of the ideas underlying Mace, see McCune (1998). For the way Paradox builds on this basic approach and extends it, see Claessen and Sörensson (2003).

Another approach, often called the Sem-style approach, should be mentioned. This approach, named after the Sem model builder (see Zhang and Zhang (1996)), avoids the conversion to propositional logic and uses constraint-propagation techniques to build the model. It's also worth mentioning that the version of Mace we used in the text is now officially called Mace-2. William McCune's new model builder is called Mace-4, and this makes use of a Sem-style strategy rather than a Mace-style one. At the time of writing, we had not experimented with Sem-style model builders for consistency and informativity checking; it would be interesting to do so.

As we mentioned in the text, the lower levels of our inference architecture are implemented in Perl. Nowadays Perl is such a ubiquitous programming language that a quick search will swiftly lead the reader to sufficient online information and introductions to understand our Perl scripts. As for books on Perl, the classic overview is Wall et al. (2000), and Schwartz and Phoenix (2001) is an excellent introduction.

Finally, we mentioned in the text that the undecidability of first-order logic rests on the assumption that Church's Thesis, the standard

model of computation, is correct. Church's Thesis is interesting because it is not a mathematical theorem (that is, something that can be proved), rather it is a generalization that has (so far) successfully withstood the test of time. For a superb discussion of Church's Thesis and what it has to do with the undecidability of first-order logic, see Boolos and Jeffrey (1989). In particular, they discuss the important concept of *semi-decidability*, a concept we hinted at (but did not explicitly introduce) in the text. Establishing whether an arbitrary first-order formula is valid or not is the classic example of a semi-decidable problem. That is, although the problem is undecidable, one half of the problem (namely establishing the validity of an input formula, *given that the input is in fact valid*) turns out to be relatively easy (technically speaking, first-order validity is recursively enumerable). The real difficulties all stem from the other half of the problem: input that are non-valid can give rise to non-terminating computations, or to put it more technically, first-order non-validity is *not* recursively enumerable (that is, determining non-validity is *really* hard).

Putting It All Together

We now have some answers to the questions with which this book began:

1. *We can build first-order representations in a compositional way for simple natural language expressions. Moreover, we can do so in a way that takes scope ambiguities into account.*
2. *We know how to automate the process of performing inference with first-order representations.*

Along the way, we have developed a number of useful tools, and learned something about the tools developed by the automated reasoning community. Now it is time to put the pieces together. As we shall see, by plugging together lambda calculus, quantifier storage, theorem proving, model building, and model checking programs, we can build a system that can handle some simple but interesting interactions with a user. We call this system Curt—short for “Clever Use of Reasoning Tools”. The idea is that the user can extend Curt’s knowledge by entering English sentences, and can query Curt about its acquired knowledge.

We will present seven different versions of Curt, starting with a basic system, and gradually extending it. So let’s get down to business and examine the system from which all the others grow: Baby Curt.

6.1 Baby Curt

Baby Curt (which you will find in the file `babyCurt.pl`) is the backbone of the Curt system, and uses the absolute minimum of inference tools (namely none at all). It is built around the `kellerStorage.pl` program from Chapter 3. Here is an example of a dialogue with Baby Curt. The text preceded by the `>` symbol is the user’s input:

`> Vincent loves Mia.`

Curt: OK.

> Every woman knows a boxer.

Curt: OK.

Baby Curt's only response is to acknowledge the input by prompting "OK." But we are also able to inspect Curt's internal memory. We do this by typing one of the *reserved commands*. (Curt does not attempt to build semantic representations for reserved commands; it simply uses them as instructions.) One of these is the command **readings** which has the following effect:

```
> readings
1 (love(vincent,mia) &
   all A (woman(A) > some B (boxer(B) & know(A,B))))
2 (love(vincent,mia) &
   some A (boxer(A) & all B (woman(B) > know(B,A))))
```

There are two interpretations in Curt's memory, due to the scope ambiguity of the second sentence we entered. Note that both formulas represent the entire set of sentences entered so far, and that conjunction is used to combine the formulas. Furthermore, note that Curt displays representations in infix notation; this notation is summarised in Appendix D.

Normally Curt takes the first of the readings in its memory and combines it with all the readings of the next sentence entered; it simply forgets about any other interpretations of the previous discourse. So if we typed in a third sentence at this stage, Curt would combine its semantic representation with the first reading listed above, and the second reading would be lost. But there is another reserved command that forces Curt to work with the interpretation of the previous discourse chosen by the user. This is the **select** command:

```
> select 2
> readings
1 (love(vincent,mia) &
   some A (boxer(A) & all B (woman(B) > know(B,A))))
```

Here we see that Curt has discarded the first reading above and replaced it with the second reading. So if we now typed in a third sentence, it would be combined with this representation.

Another reserved command is **history**. Curt reacts to this by supplying all the sentences entered so far:

```
> history
1 [vincent,loves,mia]
2 [every,woman,knows,a,boxer]
```

There are a couple more reserved commands that Baby Curt understands. The command `bye` quits the dialogue with Curt, and the command `new` starts a new discourse (it clears the interpretations in Curt's memory, and throws away the history of the discourse). As we develop Curt, we will add new reserved commands, and these will be explained as they are introduced.

So, that is what Baby Curt does—but how is it implemented?

First a fundamental implementation decision. Curt needs to keep track of the readings it generates and the sentences the user enters. The number of readings and sentences grows as the dialogue proceeds, but the user is free to throw away this information at any time by using the `new` command. How should this be handled?

We use two *dynamic* Prolog predicates. The first, `readings/1`, will be used to store the interpretations of the sentences that we enter. The second, `history/1`, will be used to store the history of the discourse. Because we declare these predicates as dynamic, we can change their definition while running the program, using the built-in Prolog `assert/1` and `retract/1` predicates. The auxiliary predicates that handle this can be found in the file `curtPredicates.pl`.

With this noted, we can turn to the large-scale architecture of Curt. The core of Curt is a dialogue control structure implemented with the help of the recursive predicate `curtTalk/1`, whose argument designates the program's executing state. This is one of the values `quit` or `run`: the `quit` value stops the recursion (and thereby ends the dialogue with Curt), whereas the `run` value takes Curt through the recursion again:

```
curtTalk(quit).

curtTalk(run) :-
    readLine(Input),
    curtUpdate(Input, ReplyMoves, State),
    curtOutput(ReplyMoves),
    curtTalk(State).
```

The control strategy of Curt is straightforwardly coded in this recursive predicate. With `readLine/1` it asks us to type in a sentence (or a reserved command), which will be unified with the Prolog variable `Input`. Next `curtUpdate/3` takes the new input and (if it was a sentence) attempts to build a semantic representation and combine the result with the representations of any previously entered sentences. It returns a list of *reply-moves*, and a new value for the program's executing state. The reply-moves tell Curt how to react to our input, and we will have a closer look at them shortly when we discuss `curtOutput/1`.

Let's first see how `curtUpdate/3` is implemented. First of all we deal with the reserved commands. These are all coded as follows:

```

curtUpdate([new],[],run) :- !,
    updateReadings([]),
    clearHistory.

curtUpdate([readings],[],run) :- !,
    readings(R),
    printRepresentations(R).

curtUpdate([history],[],run) :- !,
    history(H),
    printRepresentations(H).

curtUpdate([select,X],[],run) :- 
    number(X),
    readings(R1),
    selectReadings(X,R1,R2), !,
    updateReadings(R2).

curtUpdate([bye],[bye],quit) :- !.

```

The first four of these reserved commands return (in the third argument position) the value `run` for the execution state of Curt, and return (in the second argument position) an empty set of reply-moves, because there is no special way Curt should reply after obeying one of these commands. The reserved command `bye` behaves differently: this sets the program's execution state to `quit` (which will stop the recursion in `curtTalk/1` and terminate the program) and unifies the reply-moves to `[bye]`.

If what we entered doesn't match any of the reserved commands, Curt attempts to parse the input using `kellerStorage/2`. If it succeeds in doing that, it updates the history, and combines the readings with those of the previous discourse:

```

curtUpdate(Input,[accept],run) :-
    kellerStorage(Input,Readings), !,
    updateHistory(Input),
    combine(Readings,NewReadings),
    updateReadings(NewReadings).

```

There are two more possibilities. Maybe we absent-mindedly jiggled the return key. This is not a particularly intellectually demanding task, but Baby Curt needs to know how to deal with it:

```
curtUpdate([], [clarify], run) :- !.
```

That is, in this case Curt will demand clarification from the user.

Finally, the input might not be any of the cases described above. That is, whatever the user had entered is neither a reserved command, nor parseable by our grammar, nor a nervous tic. We deal with any such cases as follows:

```
curtUpdate(_, [noparse], run).
```

So, what the predicate `curtUpdate/3` returns (and note that because of this last fallback clause it always succeeds) is a list of reply-moves. This list can be empty (in which case Curt doesn't reply) or it can contain one of the following values: `bye`, `clarify`, `accept`, and `noparse`. (As Curt develops, the number of reply-moves will grow.) Turning the abstract reply-moves into concrete output is done by recursively going through the list of moves:

```
curtOutput([]).
```

```
curtOutput([Move|Moves]) :-  
    realiseMove(Move, Output),  
    format('`n~nCurt: ~p`n', [Output]),  
    curtOutput(Moves).
```

The reply-moves themselves are realised by using a simple look-up table:

```
realiseMove(clarify, 'Want to tell me something?').  
realiseMove(bye, 'Goodbye!').  
realiseMove(accept, 'OK.').  
realiseMove(noparse, 'What?').
```

And that is the core of the Curt architecture. However, there are several predicates that we haven't explained yet. Some of them are not so important and we ask the reader to do Exercise 6.1.2 to find out how they work. But `combine/2`, which is used by `curtUpdate/3` to combine the readings of the new sentence with the representation of the previous discourse, is worth looking at right away:

```
combine(New, New) :-  
    readings([]).  
  
combine(Readings, Updated) :-  
    readings([Old|_]),  
    findall(and(Old, New), memberList(New, Readings), Updated).
```

The first clause succeeds when Curt's memory is empty (at the beginning of the dialogue, or when the user has just used the reserved command `new`). The second clause uses the built-in Prolog `findall/3` to yield a list of all new readings combined with the first reading of the

history.

Exercise 6.1.1 Load Baby Curt (which you will find in the file `babyCurt.pl`) and enter some sentences. Try all the reserved commands.

Exercise 6.1.2 Inspect the file `curtPredicates.pl` and check how the Prolog predicates `updateReadings/1`, `updateHistory/1`, `clearHistory/0`, and `selectReadings/3` are defined.

Exercise 6.1.3 Extend Baby Curt's vocabulary by extending the grammar loaded by `kellerStorage.pl`. Add ditransitive verbs (if you haven't done so already in earlier exercises), the determiners `no` and `the`, and quantified noun phrases such as `everything` and `nobody`. You will need to modify the files `englishLexicon.pl`, `englishGrammar.pl`, `semLexStorage.pl`, and `semRulesKeller.pl`.

Exercise 6.1.4 Implement a slightly-less-babyish Curt which uses hole semantics rather than Keller storage to cope with scope ambiguities.

Programs for Baby Curt

`babyCurt.pl`

This file loads everything needed to run Baby Curt.

`curtPredicates.pl`

Auxiliary predicates used by the Curt family.

6.2 Rugrat Curt

Baby Curt may be cute, but it's not that bright. In fact, it's downright dumb:

```
> Mia smokes.
```

```
Curt: OK.
```

```
> Mia does not smoke.
```

```
Curt: OK.
```

The discourse is blatantly contradictory, but Baby Curt simply parses it, stores the representation, and carries on as usual. It's clearly time baby started to grow up. What shall we do?

Time for Rugrat Curt. By hooking up Baby Curt to the free variable tableau prover of Chapter 5, we get a system (`rugratCurt.pl`) that handles some inconsistent discourses:

```
> Mia smokes.  
Curt: OK.  
  
> Mia does not smoke.  
Message (consistency checking): proof found.  
Curt: No! I do not believe that!
```

As we can see, Rugrat Curt reacts with furious disbelief to this discourse; it does so via a new reply-move called **contradiction** that triggers this response when the input is inconsistent. But how did Rugrat Curt know that the input was inconsistent?

Recall from Chapter 5 that theorem provers provide us with a *negative check for consistency*. That is, if Discourse-So-Far is the first-order representation built by Curt from the preceding dialogue, and ϕ is the first-order representation of the latest sentence, then we can use a theorem prover to test whether

$$\text{Discourse-So-Far} \models \neg\phi.$$

By the semantic deduction theorem, this boils down to asking the theorem prover to test the validity of the formula

$$\text{Discourse-So-Far} \rightarrow \neg\phi,$$

or (equivalently) the formula

$$\neg(\text{Discourse-So-Far} \wedge \phi).$$

If the prover can do this, then the latest sentence is inconsistent with the previous discourse. Now, note the message in the second line of our example: our free-variable tableau theorem prover found a proof of inconsistency. This is what lead to Curt's complaint.

That's the basic idea, but how do we make it work in Prolog? Actually, because Rugrat Curt is an extension of Baby Curt, there is very little that changes in the backbone of the system. One change is the clause of `curtUpdate/3` that deals with parsing a sentence. We impose an additional condition (implemented by `consistentReadings/2`) that filters out inconsistent readings:

```
curtUpdate(Input,Moves,run):-  
    kellerStorage(Input,Readings), !,  
    updateHistory(Input),  
    consistentReadings(Readings,[]-ConsReadings),  
    (
```

```

ConsReadings=[],
Moves=[contradiction]
;
\+ ConsReadings=[],
Moves=[accept],
combine(ConsReadings,CombinedReadings),
updateReadings(CombinedReadings)
).

```

The consistent readings are selected as follows. For each reading computed by Keller storage, Curt tries to prove whether it is inconsistent using the predicate `consistent/2`. All readings that are consistent are stored in the variable `ConsReadings`. If there are no consistent readings, then the reply-move is unified with `[contradiction]`, otherwise the reply-move is set to `[accept]`.

```

consistentReadings([],C-C).

consistentReadings([New|Readings],C1-C2) :-
    readings(Old),
    (
        consistent(Old,New), !,
        consistentReadings(Readings,[New|C1]-C2)
    ;
        consistentReadings(Readings,C1-C2)
    ).

```

Consistency checking is implemented by calling the free-variable tableau prover from Chapter 5 with the negation of the formula and a Q-depth of 75. If no proof is found then Rugrat Curt assumes this reading is consistent.

```

consistent([Old|_],New) :-
    tprove(not(and(Old,New))), !,
    nl, write('Message (consistency checking): proof found.'), 
    fail.

consistent([],New) :-
    tprove(not(New)), !,
    nl, write('Message (consistency checking): proof found.'), 
    fail.

consistent(_).

```

Because we have now added a new reply-move, we also need to add a clause for the realisation of that move:

```
realiseMove(contradiction,'No! I do not believe that!').
```

And that's that. Before reading the text further, take time to do the following exercises.

Exercise 6.2.1 Play around with Rugrat Curt and give it more complex discourses to handle. Try to determine the limits of our little rugrat. Does it handle all inconsistent input? For example, does it handle input in which a later input sentence contradicts an earlier one? And does it handle all input efficiently? And are there natural language constructions that it does not handle?

Exercise 6.2.2 Note that in our implementation of Rugrat Curt we gave the prover $\neg(\text{Discourse-So-Far} \wedge \phi)$ rather than $\text{Discourse-So-Far} \rightarrow \neg\phi$ as input. Nothing particularly deep lies behind this choice—it's merely that the negated form is the negation of the formula we will give to the model builder (when we implement Clever Curt) to carry out the corresponding positive test. We want the reader to do two things. First, check that the two formulas really are equivalent. Second, change the implementation so that it uses the implicational form instead.

Exercise 6.2.3 Rugrat Curt is implemented using the free variable tableau prover from Chapter 5, with the Q-depth preset to 75. This means that if a proof of inconsistency requires a greater Q-depth, then Rugrat Curt will give up and wrongly conclude that the input is consistent. Try and find a natural language example which you know to be inconsistent, and which Rugrat Curt can build a representation for, but which Rugrat Curt cannot prove inconsistent.

Exercise 6.2.4 Change Rugrat Curt so that it uses the first-order resolution prover from the previous chapter instead of the tableau prover.

Programs for Rugrat Curt

`rugratCurt.pl`

This file loads everything needed to run Rugrat Curt.

6.3 Clever Curt

If you did the previous exercises carefully, you will have discovered that Rugrat Curt has a number of problems. For a start, there is a basic natural language construction (namely sentences whose verb phrase

consists of *is* followed by a noun phrase; that is, what a linguist would call a copula construction) that it does not handle correctly. Consider the following dialogue:

> Vincent is a man

Curt: OK.

> Mia likes every man.

Curt: OK.

> Mia does not like Vincent.

Curt: OK.

This is blatantly contradictory, but Rugrat Curt just doesn't see it. Why not? Because Rugrat Curt can't handle equality reasoning. The semantic representation we build for **Vincent is a man** is

`some A (man(A) & vincent = A).`

Note the way the equality symbol is used to predicate Vincent's property of being a man. But the free-variable tableau system of Chapter 5 doesn't handle equality (and neither does the resolution prover) so Rugrat Curt fails in a very obvious (and very fundamental) way.

Another problem you will have noticed is that Rugrat Curt can be extremely slow. As we emphasized in the previous chapter, first-order theorem proving is a computationally difficult task. So it is easy to come up with problems that will overwhelm a naive theorem prover (remember the Steamroller). Rugrat Curt can only crawl—we want something better.

This is where Clever Curt comes in. Clever Curt does consistency checking, but does so by using a sophisticated theorem prover and a model builder in parallel. It does this with the help of the program `callInference.pl` which we developed in Chapter 5. Recall that `callInference.pl` offers several combinations of theorem prover and model builder; you choose the combination you want by commenting out the other options. In what follows we assume that Clever Curt uses this predicate with the theorem prover Bliksem and the model builder Mace selected.

Both Bliksem and Mace are highly sophisticated reasoning tools, so it is hardly surprising that Clever Curt handles consistency checking far better than Rugrat Curt does. For a start, Bliksem handles equality reasoning, so we have no further trouble with sentences like **Vincent is a man** or **Butch is not a boxer**. Moreover, as we saw in the last chapter, Bliksem's performance is in a completely different league from that of our little free-variable tableau prover. So if a discourse contains incon-

sistencies Clever Curt will generally be far faster at finding them.

But (as we discussed in the previous chapter) the use of a model builder also gives us something useful. If a discourse is consistent, then a theorem prover will never be able to detect an inconsistency. But this means it will keep attacking the problem until its pre-allocated computational resources are used up. By running a model builder in parallel with a theorem prover, we may be able to show that a discourse is consistent before the theorem prover does all this wasteful work, for a model builder provides us with a (partial) *positive check for consistency*. If the model builder can show that

$$\text{Discourse-So-Far} \wedge \phi$$

has a model, where Discourse-So-Far is the first-order representation built by Curt from the preceding dialogue, and ϕ is the first-order representation of the latest sentence, then the latest sentence is consistent with the previous discourse.

To sum up, by running a positive check for consistency (using the model builder) in parallel with a negative test for consistency (using the theorem prover) we can hope for a better all-round performance. Bearing this in mind, let's take a look at Clever Curt (which you will find in the file `cleverCurt.pl`) in action:

```
> Mia dances.  
Message (consistency checking): mace found a result.  
Curt: OK.
```

We have something new here: a message from the model builder Mace stating that it has found a model for the input. This means that the discourse is consistent. What does the model that Mace built look like? Clever Curt has a new reserved command `models` that allows us to inspect the models created in the course of the dialogue:

```
> models  
1 model([d1],[f(1,dance,[d1]),f(0,mia,d1)])
```

This displays the models using the notation introduced in Chapter 1. Here Mace has found the simplest possible model for the discourse: a model with a domain consisting of one entity, named Mia, that has the property of dancing.

Very nice—but be careful! The models we are given are not always those we might expect. To give an example, let's extend the previous discourse as follows:

```
> Jody dances  
Message (consistency checking): mace found a result.  
Curt: OK.
```

So far so good. Mace has (correctly) told us that the discourse consisting of the sentence **Mia** dances followed by the sentence **Jody** dances is consistent (that is, Mace succeeded in building a model of it). But now let's look at the model it constructed:

```
> models
1 model([d1],[f(1,dance,[d1]),f(0,jody,d1),f(0,mia,d1)])
```

The model in its memory still contains only one entity **d1**—this entity has *two* names (namely **Jody** and **Mia**) and it dances. This is a logically sensible model of the discourse, but it is probably not what you had in mind. Why does Mace give us this model? Because model builders normally try to find a *minimal* model. And as there is no information at Curt's disposal telling it that **Mia** and **Jody** are two different people, it maps both names to the same entity. When we explicitly tell Curt that **Mia** and **Jody** are different, Curt gives us the expected model:

```
> Mia is not Jody.
Message (consistency checking): mace found a result.
Curt: OK.
```

```
> models
1 model([d1,d2],[f(0,mia,d1),f(0,jody,d2),f(1,dance,[d1,d2])])
```

As a final example, let's give Clever Curt the dialogue that Rugrat Curt failed on:

```
> Vincent is a man
Message (consistency checking): mace found a result.
Curt: OK.
```

```
> Mia likes every man.
Message (consistency checking): mace found a result.
Curt: OK.
```

```
> Mia does not like Vincent.
Message (consistency checking): bliksem found a result.
Curt: No! I do not believe that!
```

Here we see a natural interplay between the theorem prover and the model builder. The model builder shows consistency (for the first two lines of dialogue) and then the theorem prover steps in and detects the inconsistency introduced by the third line.

Now for the implementation. Because Clever Curt is an extension of Rugrat Curt, once again there is relatively little that changes. One change is the definition of **consistentReadings/3** which is now a

three-place predicate (because we want to keep track of the models returned by the model builder):

```
curtUpdate(Input,Moves,run):-
    kellerStorage(Input,Readings), !,
    updateHistory(Input),
    consistentReadings(Readings,[]-ConsReadings,[]-Models),
    (
        ConsReadings=[], 
        Moves=[contradiction]
    ;
        \+ ConsReadings=[],
        Moves=[accept],
        combine(ConsReadings,CombinedReadings),
        updateReadings(CombinedReadings),
        updateModels(Models)
    ).
```

The predicate `consistent/3` is defined as follows. It sets the maximum domain size, and then makes a call to `callTPandMB/6` (recall Chapter 5), prints the result and succeeds only if there is no proof and a model could be constructed. Note that the formula given to the theorem prover is the negation of the formula given to the model builder.

```
consistent([Old|_],New,Model):-
    Size=15,
    callTPandMB(not(and(Old,New)),and(Old,New),Size,
                Proof,Model,Engine),
    format('`nMessage (consistency checking): `c
          `p found a result.',[Engine]),
    \+ Proof=proof, Model=model([_|_],_).

consistent([],New,Model):-
    Size=15,
    callTPandMB(not(New),New,Size,Proof,Model,Engine),
    format('`nMessage (consistency checking): `c
          `p found a result.',[Engine]),
    \+ Proof=proof, Model=model([_|_],_).
```

And that's pretty much it. Once again, we suggest the reader thinks hard about the following exercises before moving on.

Exercise 6.3.1 Load Clever Curt and experiment with different discourses. Ask Curt to display the models it finds for consistent discourses. Are there any 'strange' models? Find some new examples of inconsistent discourses. Do the theorem prover and model builder always interact in the expected way?

Exercise 6.3.2 In Exercise 6.2.3 we asked you to find a natural language

example which you knew to be inconsistent, and which Rugrat Curt could build a representation for, but which Rugrat Curt could not prove inconsistent. How does Clever Curt handle your example?

Programs for Clever Curt

`cleverCurt.pl`

This file loads everything needed to run Clever Curt.

6.4 Sensitive Curt

Detecting inconsistency is one of the most fundamental inference tasks of all, and Clever Curt handles it rather well. But there is another task of interest in natural language semantics, namely informativity checking. It is often important to be able to distinguish old and new information.

Clever Curt can't do this. For example, consider the following dialogue:

```
> Vincent knows every boxer
Message (consistency checking): mace found a result.
Curt: OK.
```

```
> Butch is a boxer
Message (consistency checking): mace found a result.
Curt: OK.
```

```
> Vincent knows Butch
Message (consistency checking): mace found a result.
Curt: OK.
```

The third sentence, **Vincent knows Butch**, follows from the previous two sentences, **Vincent knows every boxer** and **Butch is a boxer**. That is, it doesn't introduce any new information—it is uninformative with respect to what has gone before. Clever Curt, however, doesn't realise this. Now, in some contexts this example wouldn't be a problematic example of language use, but Clever Curt can't detect even extremely blatant examples of repeated information:

```
> Mia smokes
Message (consistency checking): mace found a result.
Curt: OK.
```

> Mia smokes

Message (consistency checking): mace found a result.

Curt: OK.

> Mia smokes

Message (consistency checking): mace found a result.

Curt: OK.

In short, Clever Curt is blissfully unaware of the difference between old and new information. How can we sensitise Curt to this distinction?

We learned two important answers in Chapter 5 (a different answer is explored in Exercise 6.4.2). First, we learned that theorem provers provide us with a *negative check for informativity*. That is, if Discourse-So-Far is the first-order representation built by Curt from the preceding dialogue, and ϕ is the first-order representation of the latest sentence, then we can use a theorem prover to test whether

$$\text{Discourse-So-Far} \models \phi.$$

By the semantic deduction theorem, we can do so by asking the theorem prover to test the validity of the formula

$$\text{Discourse-So-Far} \rightarrow \phi,$$

or (equivalently) of

$$\neg(\text{Discourse-So-Far} \wedge \neg\phi).$$

If the prover can do this, then the latest sentence is not informative with respect to the previous discourse. We also learned that model builders provide us with a (partial) *positive check for informativity*. So if a model builder can show that

$$\text{Discourse-So-Far} \wedge \neg\phi$$

has a model, then the latest sentence is informative with respect to the previous discourse. (Note that once again we have arranged matters so that the formula given to the theorem prover is the negation of the formula given to the model builder.)

Both checks are built into Sensitive Curt (see file `sensitiveCurt.pl`). Sensitive Curt is an extension of Clever Curt (that is, Sensitive Curt performs positive and negative consistency checks in parallel). But Sensitive Curt is also alert for uninformative contributions by the user. It does so by using a theorem prover and model builder in parallel to check whether new contributions are informative with respect to the readings Curt has in its memory.

Here's an example of Sensitive Curt in action:

> Mia smokes

Message (consistency checking): mace found a result.

Curt: OK.

> Mia smokes

Message (consistency checking): mace found a result

Message (informativity checking): bliksem found a result.

Curt: Well, that is obvious!

This is how Sensitive Curt behaves, but how is it implemented? The key step is to add yet another filter to the clause `curtUpdate/3`. As well as watching out for consistent readings (as we did in Clever Curt) we check whether or not readings are informative by using the predicate `informativeReadings/2`:

```
curtUpdate(Input,Moves,run):-
    kellerStorage(Input,Readings), !,
    updateHistory(Input),
    consistentReadings(Readings,[]-ConsReadings,[]-Models),
    (
        ConsReadings=[], 
        Moves=[contradiction]
    ;
        \+ ConsReadings=[],
        informativeReadings(ConsReadings,[]-InfReadings),
        (
            InfReadings=[],
            Moves=[obvious]
        ;
            \+ InfReadings=[],
            Moves=[accept]
        ),
        combine(ConsReadings,CombinedReadings),
        updateReadings(CombinedReadings),
        updateModels(Models)
    ).
```

There are two disjunctive clauses in this predicate. The first clause checks whether the number of readings is empty. If this is the case, all readings are inconsistent, so there is no need to check for informativity. The second disjunctive clause checks if there are no informative readings among the set of consistent readings. It does so with the help of the predicate `informativeReadings/2`. This sets the reply-move `[obvious]` if it succeeds. Otherwise, the reply-moves gets the value `[accept]`.

This predicate is defined as follows:

```
informativeReadings([],I-I).
```

```
informativeReadings([New|L], I1-I2) :-  
    readings(Old),  
    (  
        informative(Old, New), !,  
        informativeReadings(L, [New|I1]-I2)  
    ;  
        informativeReadings(L, I1-I2)  
    ).
```

So, how is `informative/2` implemented? As follows:

```
informative([0|_], N) :-  
    Size=15,  
    callTPandMB(not(and(0,not(N))), and(0,not(N)), Size,  
                Proof, Model, Engine),  
    format('`nMessage (informativity checking): \c  
          `p found a result.', [Engine]),  
    \+ Proof=proof, Model=model([_|_], _).  
  
informative([], New) :-  
    Size=15,  
    callTPandMB(New, not(New), Size, Proof, Model, Engine),  
    format('`nMessage (informativity checking): \c  
          `p found a result.', [Engine]),  
    \+ Proof=proof, Model=model([_|_], _).
```

Finally, because we have added a new reply-move, we need to add a clause for the realisation of that move:

```
realiseMove(obvious, 'Well, that is obvious!').
```

And that's that. But before moving on, a few more remarks are in order.

Clever Curt was completely insensitive to the distinction between new and old information. Sensitive Curt goes to the other extreme—it is completely tuned to this distinction (or at least, tuned to them up to the limits imposed by the theorem prover and the model builder that it uses). That is, within these limits, Curt can judge whether arguments are valid or invalid.

Recall from Chapter 1 that a natural language argument is a sequence of sentences, the last of which is called the *conclusion*, the rest *premises*. Here, for example, is a simple two premise natural language argument:

- (6) Vincent knows every boxer.
 Butch is a boxer.

 Vincent knows Butch.

This argument is *valid*. If the premises are true, then the conclusion is true too. And if we give the premises and conclusions of this argument to Sensitive Curt, it will smugly announce ‘Well, that is obvious!’, thereby establishing that the argument is valid.

Similarly, we can use Sensitive Curt to show that arguments are *not* valid. The following, for instance, is not a valid argument:

- If Mia snorts, then Vincent smokes.
 (7) Vincent smokes.

 Mia snorts.

If we enter these three lines one by one, Sensitive Curt will *not* make its triumphant ‘Well, that is obvious!’ cry. For it’s not a valid argument at all. If you doubt whether Sensitive Curt is right about this try, give it the same two premises, but change the third line to *Mia does not snort*. Sensitive Curt will happily generate a model, thereby showing that *Mia snorts* is informative with respect to the premises, and hence that the original argument is not valid.

One final remark about argumentation is worth making. Consider the following argument:

- A woman loves every man.
 (8) Every boxer is a man.

 A woman loves every boxer.

Is this a valid or not? The answer is: *sometime yes, sometimes no*. The premises have scope ambiguities, and so does the conclusion, so it really depends on which readings are intended. We leave the reader to sort out the details in the following exercise.

Exercise 6.4.1 Try Sensitive Curt on a number of examples. A good starting point is the argument just given. When is it valid, and when is it invalid? (Recall that there is a reserved command **select** to allow us to choose which reading is carried through.)

Exercise 6.4.2 In the text we used the approach discussed in Chapter 5 (namely running a theorem prover and model builder in parallel) to determine informativity. But it is interesting to think about other methods of establishing informativity.

Clever Curt carries out consistency checks before it checks for informativity, so if ϕ is a first-order representation of (one of the readings of) the new sentence that survives this check, then we know that ϕ is consistent with what has gone before. Now, suppose that M is the model Clever Curt has made for the discourse so far, and suppose that ϕ is *false* in this model. What does this tell us? Use this observation (and the first-order model checker implemented in Chapter 1) to implement a positive test for informativity. What are

the advantages and disadvantages of this approach to informativity checking compared to the approach in the text?

Programs for Sensitive Curt

`sensitiveCurt.pl`

This file loads everything needed to run Sensitive Curt.

6.5 Scrupulous Curt

Although Sensitive Curt can perform both consistency and informativity checking, in one respect it is rather naive: it accepts as distinct readings all the output provided by the Keller Storage program. Now, the Keller Storage program does weed out α -equivalent readings, but it can't detect more logically sophisticated examples of equivalent readings. Consider, for example, the following:

```
> A boxer loves a woman.  
Message (consistency checking): mace found a result.  
Message (consistency checking): mace found a result.  
Curt: OK.
```

```
> readings  
1 some A (boxer(A) & some B (woman(B) & love(A,B)))  
2 some A (woman(A) & some B (boxer(B) & love(B,A)))
```

Now, a superficial glance at this output suggests that this sentence has two readings. But it should be intuitively obvious that there is really only one reading, hence the two formulas above must boil down to the same thing. This is easy to show. First note that we can move the innermost quantifiers to the outside without changing the meaning:

```
1 some A some B (boxer(A) & (woman(B) & love(A,B)))  
2 some A some B (woman(A) & (boxer(B) & love(B,A)))
```

Then, by appealing to the commutativity and associativity of conjunction, we can rewrite the first formula as follows:

```
1 some A some B (woman(B) & (boxer(A) & love(A,B)))
```

Finally, if we permute the existential quantifiers at the start of the first formula (another operation which does not change the meaning) we obtain:

```
1 some B some A (woman(B) & (boxer(A) & love(A,B)))
```

But this is α -equivalent to

```
2 some A some B (woman(A) & (boxer(B) & love(B,A))).
```

Hence both representations say the same thing, hence one of the representations can be eliminated.

Now, it would be nice if our system could eliminate superfluous readings, and this is what Scrupulous Curt (see `scrupulousCurt.pl`) does for us. In essence, given two readings ϕ and ψ from a set of readings, Scrupulous Curt calls the theorem prover to try and prove both $\phi \rightarrow \psi$ and $\psi \rightarrow \phi$ (that is, it tries to prove $\phi \leftrightarrow \psi$). Scrupulous Curt has a new reserved command called `summary` which eliminates logically equivalent readings from its memory. Here is how Scrupulous Curt deals with the previous example:

```
> A boxer loves a woman
```

```
Message (consistency checking): mace found a result.
```

```
Message (consistency checking): mace found a result.
```

```
Message (informativity checking): mace found a result.
```

```
Message (informativity checking): paradox found a result.
```

```
Curt: OK.
```

```
> summary
```

```
Message (eliminating equivalent readings): there are 2 readings:
```

```
1 some(A, and(woman(A), some(B, and(boxer(B), love(B,A)))))
```

```
2 some(A, and(boxer(A), some(B, and(woman(B), love(A,B)))))
```

```
Readings 1 and 2 are equivalent (otter).
```

```
> readings
```

```
1 some(A, and(boxer(A), some(B, and(woman(B), love(A,B)))))
```

Well, that is what Scrupulous Curt does, but how does it work? We need to make the following changes to Sensitive Curt. First we extend the `curtUpdate/3` predicate:

```
curtUpdate([summary],[],run):-
    readings(Readings),
    elimEquivReadings(Readings,Unique),
    updateReadings(Unique),
    updateModels([]).
```

The predicate that pulls this all together is `elimEquivReadings/2`. This consists of three clauses. The first and second clauses are for the cases where there are no readings or only one reading. The third clause

numbers the readings and gives them to `elimEquivReadings/3`, which is where the logical crunching is carried out.

```
elimEquivReadings([], []).  
  
elimEquivReadings([Reading], [Reading]).  
  
elimEquivReadings(Readings, Unique) :-  
    numberReadings(Readings, 0, N, Numbered),  
    format(`~nMessage (eliminating equivalent readings): \c  
          there are ~p readings:', [N]),  
    printRepresentations(Readings),  
    elimEquivReadings(Numbered, [], Unique).
```

Numbering the readings is easy—it's done by converting the list of readings into a list of terms $n/2$ where the first argument is the number, and the second argument holds the reading:

```
numberReadings([], N, N, []):-  
    N > 1.  
  
numberReadings([X|L1], N1, N3, [n(N2, X)|L2]) :-  
    N2 is N1 + 1,  
    numberReadings(L1, N2, N3, L2).
```

And now for `elimEquivReadings/3`, which is where we actually try to prove the equivalences. This predicate has two clauses. The first clause selects two numbered readings, checks whether we haven't tested equivalence for these readings already (it does so using the book-keeping term `diff/2`), and calls the theorem prover. This is repeated (using recursion) until all pairs of readings are checked, and this is where the second clause comes into action, which converts the list of numbered readings back into an ordinary list of readings:

```
elimEquivReadings(Numbered, Diff, Unique) :-  
    selectFromList(n(N1, R1), Numbered, Readings),  
    memberList(n(N2, R2), Readings),  
    \+ memberList(diff(N1, N2), Diff), !,  
    Formula=and(imp(R1, R2), imp(R2, R1)),  
    callTP(Formula, Result, Engine),  
    (  
        Result=proof, !,  
        format('Readings ~p and ~p are equivalent \c  
              (~p).~n',[N1,N2,Engine]),  
        elimEquivReadings(Readings, Diff, Unique)  
    ;  
        format('Readings ~p and ~p are probably \c  
              not equivalent.~n',[N1,N2]),
```

```

    elimEquivReadings([n(N1,R1)|Readings],
                      [diff(N1,N2),diff(N2,N1)|Diff],Unique)
    ).

elimEquivReadings(Numbered,_,Unique):-
    findall(Reading,memberList(n(_,Reading),Numbered),Unique).

```

And that's that. But is this all there is to it? Well, yes and no. Note that we've only made use of a theorem prover. Can't a model builder help us out here too? Yes, of course it can, as the following dialogue makes clear:

```

> Every boxer loves a woman.
Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Curt: OK.

> summary
Message (eliminating equivalent readings):
Readings 1 and 2 are probably not equivalent.

```

Now (as every reader of this book should know by now!) the two readings of Every boxer loves a woman are certainly *not* equivalent. But that's not what Scrupulous Curt says. Scrupulous Curt is indeed scrupulous: it merely says that the two readings are probably not equivalent. Why? Because what has happened in this case is that the theorem prover has tried, and failed, to prove equivalence. But failure to prove something is no guarantee that a proof does not exist. It would be nice to use the model builder to build a concrete model in which one of the formulas is true and the other false, thereby explicitly showing the non-equivalence of these formulas. The reader is asked to implement this extension in Exercise 6.5.4

Exercise 6.5.1 Try Scrupulous Curt on other examples involving spurious scoping ambiguities. Test it to destruction. That is, try to come up with examples that grind Scrupulous Curt to a halt.

Exercise 6.5.2 As the previous exercise shows, it is easy to overload Scrupulous Curt. This is not surprising: even quite simple sentences may give rise to lots of readings, and theorem proving is a computationally intensive task, so systematically testing for equivalences on all readings is not exactly a holiday!

Nonetheless, we certainly didn't do ourselves any favours in the above implementation. It is possible to rewrite the Prolog code so that fewer calls to the theorem prover are made. (Hint: if you have already proved that

$\phi \rightarrow \theta$ and $\theta \rightarrow \psi$ it follows that $\phi \rightarrow \psi$; calling a theorem prover on this last problem is a waste of resources.)

Exercise 6.5.3 Sometimes (recall Chapter 3) there is a strongest reading, namely one that implies all the others. And sometimes (recall the Every owner of a hashbar... example in Chapter 3) there are two (or more) strongest readings (that is, sometimes there is a collection of readings ϕ_1, \dots, ϕ_n , such that any other reading follows from one of these formulas).

Rewrite Scrupulous Curt so that it calculates all the strongest readings, retains them, and throws all other readings away.

Exercise 6.5.4 Extend `eliminateEquivalentReadings/3` by calling a model builder in parallel to explicitly demonstrate the non-equivalence of readings.

Programs for Scrupulous Curt

`scrupulousCurt.pl`

This file loads everything needed to run Scrupulous Curt.

`elimEquivReadings.pl`

Contains the predicates for eliminating equivalent readings.

6.6 Knowledgeable Curt

Curt can now do so many clever things (perform consistency and informativity checks, and weed out redundant readings) that it is rather sad that we now have to make an insulting observation about it: Curt is ignorant. Pig ignorant. It knows absolutely nothing about anything, and this can make its responses look absurd. Consider the following example:

```
> Jody is a woman.
```

```
Message (informativity checking): mace found a result.
```

```
Message (consistency checking): mace found a result.
```

```
Curt: OK.
```

```
> Jody is a plant.
```

```
Message (consistency checking): mace found a result.
```

```
Message (informativity checking): mace found a result.
```

```
Curt: OK.
```

A human being, of course, immediately sees the conflict between the claims that Jody is a woman and that she is a plant. But for Scrupulous Curt, ignorance is bliss. It blindly accepts the input and carries on.

What should we do? Well, there is only one cure for ignorance: knowledge, knowledge, and yet more knowledge! Adding background knowledge leads us to Knowledgeable Curt, an educated version of Scrupulous Curt. Now, recall the discussion of Section 1.4. As we pointed out there, one of the pleasant aspects of using first-order logic as a representation formalism is the simple way it enables background knowledge to be incorporated into inference. If we can formulate the required background knowledge in first-order logic, then using this knowledge is merely a matter of using these formulas as additional premises. Let's be precise about what "using these formulas as additional premises" means.

In the versions of Curt we have seen so far, if Discourse-So-Far is the representation built by Curt from the preceding dialogue, and ϕ is the representation of the latest sentence, then consistency checking boils down to using a theorem prover to test whether

$$\text{Discourse-So-Far} \models \neg\phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\text{Discourse-So-Far} \wedge \phi$$

has a model (the partial positive test). As for informativity checking, this boils down to using a theorem prover to test whether

$$\text{Discourse-So-Far} \models \phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\text{Discourse-So-Far} \wedge \neg\phi$$

has a model (the partial positive test).

So what happens when we add (first-order) background knowledge? Well, let's suppose that we have written down first-order formulas expressing the required background knowledge, and let's suppose that we have classified this knowledge into three kinds: *lexical knowledge (LK)*, *world knowledge (WK)*, and *situational knowledge (SK)*. Then consistency checking boils down to using a theorem prover to test whether

$$\text{LK} \wedge \text{WK} \wedge \text{SK} \wedge \text{Discourse-So-Far} \models \neg\phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\text{LK} \wedge \text{WK} \wedge \text{SK} \wedge \text{Discourse-So-Far} \wedge \phi$$

has a model (the partial positive test). As for informativity checking, this boils down to using a theorem prover to test whether

$$\text{LK} \wedge \text{WK} \wedge \text{SK} \wedge \text{Discourse-So-Far} \models \phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\text{LK} \wedge \text{WK} \wedge \text{SK} \wedge \text{Discourse-So-Far} \wedge \neg\phi$$

has a model (the partial positive test).

Let's look at an example. Knowledgeable Curt has a small fund of lexical knowledge, world knowledge, and situational knowledge at its disposal, and among the things it knows is that women are people and that people cannot be plants. Knowledgeable Curt brings this information to bear in the way just described, and thus handles the previous dialogue differently:

```
> Jody is a woman.  
Message (consistency checking): mace found a result.  
Message (informativity checking): mace found a result.  
Curt: OK.  
  
> Jody is a plant.  
Message (consistency checking): bliksem found a result.  
Curt: No! I do not believe that!
```

This is a conceptually clean and simple way of thinking about inferences that involve knowledge. Nonetheless, one point should be made right away: a *lot* of knowledge is needed to ensure that Curt behaves in even a semi-sensible way. It is revealing to see just how much information Knowledgeable Curt brings to bear on even the simplest problem. Consider the following dialogue:

```
> Mia smokes.  
Message (consistency checking): mace found a result.  
Message (informativity checking): mace found a result.  
Curt: OK.  
  
> readings  
1 smoke(mia)
```

This dialogue is about as simple as it gets: one sentence and one reading (and a very simple reading at that). But when we look at what took place behind the scenes, we are in for a bit of a shock. Knowledgeable Curt has a new reserved command **knowledge** which computes and shows the background knowledge used for the current reading. What background information did the model builder (and indeed, the

theorem prover) take into account in its consistency check? All this:

> knowledge

```
1 (all A (event(A) > thing(A)) &
  (all B (entity(B) > thing(B)) &
  (all C (object(C) > entity(C)) &
  (all D (organism(D) > entity(D)) &
  (all E (animal(E) > organism(E)) &
  (all F (person(F) > organism(F)) &
  (all G (man(G) > person(G)) &
  (all H (woman(H) > person(H)) &
  (all I (entity(I) > ~ event(I)) &
  (all J (organism(J) > ~ object(J)) &
  (all K (person(K) > ~ animal(K)) &
  (all L (woman(L) > ~ man(L)) &
  all M (M = mia > woman(M)))))))))))))))
```

Now, that's a lot of formulas. And this raises a host of questions. Where exactly did Knowledgeable Curt get hold of this knowledge? Where are these formulas stored? Is this all the knowledge at its disposal, or is it merely a selection? And if it's a selection, why were these formulas selected for this inference and not others? And how exactly was this background knowledge chosen? Did we haphazardly write down a host of useful looking formulas, or was there an attempt to structure the knowledge representation process? We discuss these issues in the coming pages.

Lexical Knowledge

Well, did we just haphazardly write down a host of plausible looking formulas, or did we try to take a more structured approach? Unsurprisingly, we tried to be as systematic as possible. Formulating background knowledge is an extremely difficult business. Even for a simple system like Curt, it is not easy to see what is required, and it is easy to make mistakes or overlook things. Guiding principles are vital.

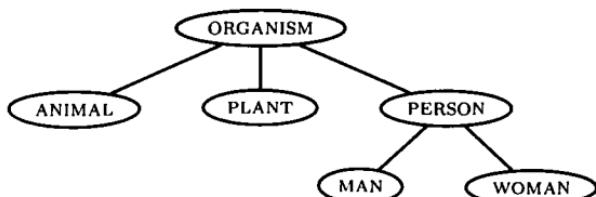
We've already mentioned one useful principle: splitting background knowledge into lexical knowledge (meaning of words), world knowledge (general facts about the world) and situational knowledge (facts that hold in the current situation). Why is this helpful? For a start, simply realising that these distinctions can be drawn (even if they are a bit fuzzy) is a useful first step. More importantly, isolating lexical knowledge as an interesting subcategory of knowledge forces us to think hard about the meaning of words—and this will lead us to systematic ways of formulating and storing knowledge.

But haven't we already thought hard about the meaning of words? In one sense, yes. For example we have said that the meaning of **man** is $\lambda y.\text{MAN}(y)$, and that the meaning of **every** is $\lambda u.\lambda v.\forall x(u@x \rightarrow v@x)$. These are important things to say about the meanings of these words. In particular, these representations pin down, precisely and elegantly, what we might call the *combinatorial*, or *compositional*, meanings of these words.

But there is an important difference between the compositional meanings of **man** (an open class word, namely a noun) and **every** (a closed class word, namely a determiner). Intuitively, we feel that associating **every** with the representation $\lambda u.\lambda v.\forall x(u@x \rightarrow v@x)$ gives us the essence of its meaning; clearly there are other things that could (and should) be said, but simply giving this representation accomplishes a lot. But we certainly don't feel this way when we give **man** the representation $\lambda y.\text{MAN}(y)$. Far from thinking we've said it all, we feel we've hardly started. Instead of getting to grips with all aspects of the meaning of **man**, this definition essentially gives us the symbol **MAN**, and that's that. In particular, how this symbol relates to other symbols such as **PERSON**, **MALE**, **FATHER**, **HUSBAND**, **CHROMOSOME**, and so on, is completely unspecified.

In short, we need to say something about *lexical semantics*. Lexical semantics studies the relationships that hold *between* the concepts expressed by words. It is a fascinating (and difficult) subject which we won't be able to explore in this book. But it is important to say something about it, for thinking systematically about the relationships that hold between (the concepts expressed by) words is an important way of structuring background knowledge. We shall illustrate this with a simple (but systematic) treatment of *nouns*. In essence we will describe the fine structure of entities (that is, the things these words denote) by designing an *ontology of concepts*. The work involved is essentially classificatory: we arrange the concepts expressed by words into a hierarchy. Once we've done this, we'll go on and discuss how to make such knowledge available to Curt.

Here is a simple tree structure representing an ontology made up of six concepts (**organism**, **animal**, **plant**, **person**, **man**, and **woman**):



These are the kind of tree-structures that we will use to classify nouns. Thinking about noun meanings in terms of trees is a natural way of formulating knowledge, for such diagrams enable us to use the idea of *inheritance*: daughter nodes inherit information from mother nodes. For example, according to the above tree, every plant is an organism, every animal is an organism, every person is an organism, every man is a person, and every woman is a person.

Some of these classes will be *disjoint*. For example, the concept ANIMAL is disjoint from PLANT, because animals are classified as animate, whereas plants are inanimate. That is, according to this partial picture of the world, nothing can be a plant and an animal, nothing can be an animal and a person, and nothing can be a plant and a person. And other classes may inherit such disjointness properties. For example, MAN inherits the property of being disjoint from plants and animals, because MAN is a daughter node of PERSON. To sum up, the tree implicitly encodes the idea of disjointness between concepts: daughter nodes are disjoint.

Read this way, the classification presented by the above tree makes fairly good sense. It is possible to quibble with the terminology—a biologist would insist that persons *are* animals—but for many purposes it is a sensible starting point. But whether or not you like this particular classification, it is important that you appreciate the tree-based thinking that gave rise to it: such thinking is a useful way of structuring our attempts at knowledge representation.

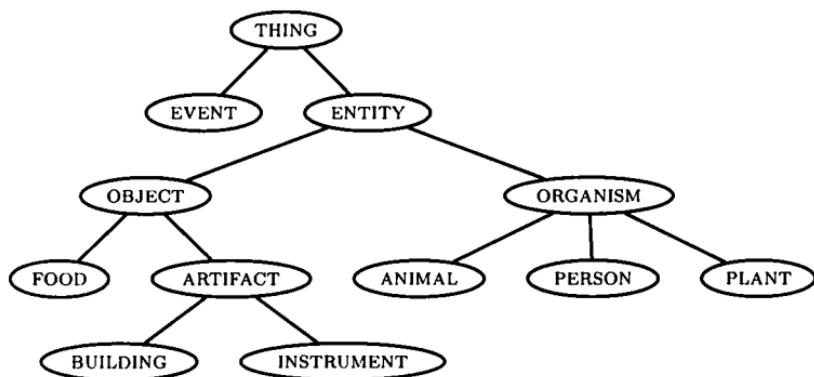
Once we have designed an ontology, we want to use this information to perform inference. Now, in this book the inference tools used are those of first-order logic—so if we want to make use of the lexical knowledge we have so carefully defined, we need to make it available in a form that first-order theorem provers, model builders and model checkers can use.

This is easy to do—we simply implement the ontology as a collection of first-order axioms:

$$\begin{aligned}
 & \forall x (\text{ANIMAL}(x) \rightarrow \text{ORGANISM}(x)) \\
 & \forall x (\text{PLANT}(x) \rightarrow \text{ORGANISM}(x)) \\
 & \forall x (\text{PERSON}(x) \rightarrow \text{ORGANISM}(x)) \\
 & \forall x (\text{MAN}(x) \rightarrow \text{PERSON}(x)) \\
 & \forall x (\text{WOMAN}(x) \rightarrow \text{PERSON}(x)) \\
 \\
 & \forall x (\text{ANIMAL}(x) \rightarrow \neg \text{PLANT}(x)) \\
 & \forall x (\text{ANIMAL}(x) \rightarrow \neg \text{PERSON}(x)) \\
 & \forall x (\text{PERSON}(x) \rightarrow \neg \text{PLANT}(x)) \\
 & \forall x (\text{MAN}(x) \rightarrow \neg \text{WOMAN}(x)).
 \end{aligned}$$

That's the basic idea. Let's now apply it on a bigger scale, and attempt to classify all the nouns in our lexicon. That is, we shall give a tree that covers all of the nouns that our Prolog programs work with. Obviously there are many ways of carrying out this task, and it is highly likely that if two different people classified the same set of nouns, they'd each come up with a different ontology, for everyone has a slightly different conception of the world. Is this a serious obstacle? For our purposes, no. We simply want to illustrate how important it is to provide *some* (consistent) picture of the world, and how to put this picture to work. But of course, if you want to adapt the tools provided in this book to some particular domain, then it is virtually inevitable that you will have to extend or otherwise modify the ontology provided for Knowledgeable Curt. This will require a serious (and probably lengthy) analysis of your problem domain. There's no getting away from it: knowledge representation is hard work.

Here is the top of our ontology for nouns:



Just below the root of this structure we have EVENT (an abstract thing), and ENTITY (a concrete thing). Then ENTITY is divided into OBJECT (any non-living entity) and ORGANISM (a living entity). The remaining classifications should be fairly self-explanatory.

And that is our ontology (or at least, the uppermost part of it). And now, taking this tree as our point of departure, the next step is to use it as a guide to the world of nouns (or at least, the nouns in our lexicon). That is, we must go painstakingly through our lexicon and relate each of the nouns it contains to one of these nodes. For example, foot massage is an event, piercing is an artifact, five dollar shake is a beverage, and so on. Incidentally, this approach also lets us deal with homonyms (words that have multiple unconnected meanings), as Exercise 6.6.1 asks the reader to show.

All the examples discussed so far involve noun meaning—and indeed, nearly all of Knowledgeable Curt's lexical knowledge is about nouns. Nonetheless, it also knows a little about proper names (for example, that Mia is a woman's name), about adjectives (for example, that no object can be both blue and red) and about verbs (for example, that die holds of organisms).

The file `lexicalKnowledge.pl` contains the lexical knowledge coded as axioms. The axioms are stored using the `lexicalKnowledge/3` predicate. Here are some examples:

```
lexicalKnowledge(organism,1,Axiom):-  
    Axiom = all(A,imp(organism(A),entity(A))).  
  
lexicalKnowledge(animal,1,Axiom):-  
    Axiom = all(A,imp(animal(A),organism(A))).  
  
lexicalKnowledge(plant,1,Axiom):-  
    Axiom = all(A,imp(plant(A),not(person(A)))).  
  
lexicalKnowledge(instrument,1,Axiom):-  
    Axiom = all(A,imp(instrument(A),not(building(A)))).
```

Note that we do not give an unstructured list of axioms. Instead, `lexicalKnowledge/3` also captures information about what we call the trigger symbol and its arity. For instance, in the first axiom above, ORGANISM (a one-place predicate symbol) is the trigger symbol for the axiom. We won't discuss the role of trigger symbols now, but will return to the topic at the end of this section.

World Knowledge

The lexical knowledge we have added is useful, but Curt needs more than the knowledge generated by words alone—it needs knowledge of a more general kind: world knowledge.

It is hard to be precise about what should fall under the heading 'world knowledge'. For example, it is arguable that some of the knowledge we have called 'lexical knowledge' could have (and perhaps should have) been thought of as 'world knowledge', and that some of our 'world knowledge' is really lexical. Moreover, it is difficult to formulate world knowledge in a systematic manner; a truly systematic approach is probably only possible in clearly defined application domains. So we haven't made a serious attempt to add world knowledge; in essence we've simply created a place in our inference architecture where world knowledge can be added as needed.

This place is the file `worldKnowledge.pl`. It contains three example

axioms:

```

worldKnowledge(have,2,Axiom) :-
    Axiom = and(not(some(X,have(X,X))), all(X,all(Y,
        imp(some(Z, and(object(X), and(object(Y),
            and(object(Z), and(have(X,Z),
                have(Y,Z))))))), eq(X,Y)))). 

worldKnowledge(husband,1,Axiom) :-
    Axiom = all(A,imp(husband(A),married(A))). 

worldKnowledge(wife,1,Axiom) :-
    Axiom = all(A,imp(wife(A),married(A))). 
```

That is, the first axiom tries to pin down some very general facts about relations between objects: it says that no object has itself as a subpart, and that no object is a subpart of two distinct objects. The next two axioms specify an obvious consequence of being a wife (or being a husband), namely that if you are either you are married.

It should be clear that such knowledge has an important role to play in inference and thus is relevant to computational semantics. For a start, with this knowledge at its disposal, Curt can be more choosy about which sentences and discourses it accepts as informative and consistent. For example, suppose we start a discourse with the claim that

Mia is Vincent's wife.

This is perfectly reasonable—but if we then continued with

Mia is married.

we would like Curt to spot that this second sentence is not informative with respect to the first. Exercise 6.6.4 asks the reader to extend the grammar to cover such examples. We'll soon see a more interesting example of world knowledge in action, but first we'll introduce the concept of situational knowledge.

Situational Knowledge

World knowledge is intended to embody very *general* claims about the world. Situational knowledge, on the other hand, is highly specific: roughly speaking, it is the place in our inference architecture where facts about the here-and-now live.

If you look at `situationalKnowledge.pl` you will find it contains the following example axiom

```

situationalKnowledge(Axiom) :-
    Axiom = some(X,some(Y, and(car(X), and(car(Y), not(eq(X,Y)))))). 
```

That is, it states that the current situation contains at least two cars.

Here's a nice example of this knowledge in action. If we use this situational axiom together with our world knowledge axioms we can filter out some impossible quantifier scopings. Consider the sentence *every car has a radio*. This has the same structure as *every boxer loves a woman*, but clearly world knowledge typically rules out the reading where *a radio* has wide scope. After all—unless we are working in a domain containing no cars at all, or only one car—it is highly implausible that all cars will share the same radio.

Here's how Knowledgeable Curt handles this example:

```
> Every car has a radio.  
Message (consistency checking): mace found a result.  
Message (consistency checking): bliksem found a result.  
Curt: OK.
```

That is, the Keller Storage program generates two candidate readings, and consistency checking is performed. Now, the first message says that the model builder Mace (which carries out the positive test) found a result, which means that the first reading is consistent. However, the second message says that the theorem prover Bliksem (which carries out the negative test) found a result, which means that an inconsistency was detected. And indeed, when we give the command `readings` we see that only one of the candidate readings (namely, where every car takes wide scope) has survived:

```
> ?- readings  
1 all A (car(A) > some B (radio(B) & have(A,B)))
```

What lead Curt to reject the other reading? The interplay between world and situational knowledge. From the world knowledge component Curt drew on the principle that no object can be a subpart of two distinct objects. Now, this alone is not enough to rule out the unwanted scoping (after all, our application domain might not have contained any cars). But the situational knowledge component supplied the additional information that there were at least two cars, which enabled Curt to spot the inconsistency.

Selecting Background Knowledge

We use the predicates `lexicalKnowledge/3` and `worldKnowledge/3` to store background knowledge. For instance, for the concept `car` we have the following axiom:

```
lexicalKnowledge(car,1,Axiom):-  
    Axiom = all(X,imp(car(X),vehicle(X))).
```

Recall that we called the first argument of `lexicalKnowledge/3` the *trigger symbol* for the axiom (in this case, we have `CAR` as trigger symbol for the axiom $\forall x(CAR(x) \rightarrow VEHICLE(x))$). Why do we need triggers?

The answer can be summed up in one word: efficiency. It should be clear that storing all our axioms in one big unstructured database is not a good idea. We *don't* want to call on all the background knowledge we have at disposal for every inference. The more background knowledge we use, the greater the risk that we overwhelm our theorem provers and model builders. Given a particular instance of a problem, we should weed out axioms that are irrelevant. For example, if we are talking about cars, we may well need to know about vehicles too—but there is no need for knowledge about vehicles if we are only talking about different kinds of beverages.

The predicate `backgroundKnowledge/2` takes this into account by selecting the background knowledge axioms related to a formula. Its first argument should be instantiated to the formula you want your background knowledge for (we call this the *trigger formula*), and the second argument will be unified with a formula representing the background knowledge.

In essence, `backgroundKnowledge/2` works as follows: it finds all the non-logical symbols that appear in the trigger formula, and then checks whether there are any axioms in the database containing these symbols. It does so via the `lexicalKnowledge/3` and `worldKnowledge/3` predicates discussed earlier in this section. It forms the conjunctions of all such formulas plus the situational knowledge, and only this selected information is actually used in the inference task. (This relevant conjunction is what you see when you use the reserved command `knowledge in Curt`.)

The inclusion of `backgroundKnowledge/2` certainly does *not* solve the efficiency problems raised by working with background knowledge. For a brutally direct demonstration of this, look at Exercise 6.6.10. There we add the three classical axioms for equality as background knowledge, and the reader will not find it hard to come up with examples where the use of these three simple axioms (and no others) overwhelms the prover. In view of this fundamental example, it would be naive to expect that any simple “select the relevant axioms predicate” (such as `backgroundKnowledge/2`) is going to guarantee efficiency.

Rather, the point of including `backgroundKnowledge/2` into our inference architecture is this. First-order theorem proving—especially with background knowledge—is computationally demanding. Nonetheless, it seems plausible that in many situations of relevance to computational semantics, it may be possible to find useful *heuristics* that

make life simpler. The predicate `backgroundKnowledge/2` is essentially a simple demonstration of how mechanisms for using heuristics might be incorporated into our inference architecture.

Exercise 6.6.1 Think of a way to put homonyms (words that have multiple unconnected meanings) in the ontology of nouns. For instance, according to WordNet, a boxer could be someone who fights with his fists for sport, a worker who packs things into containers, or a breed of stocky medium-sized short-haired dog with a brindled coat and square-jawed muzzle developed in Germany.

Exercise 6.6.2 Examine the predicates in the file `backgroundKnowledge.pl`, in particular the predicates `backgroundKnowledge/2`, `formula2symbols/2`, and `computeBackgroundKnowledge/2`.

Exercise 6.6.3 Some of the axioms at Knowledgeable Curt's disposal are too restrictive. For example, defining cleaning as a relation between persons and artifacts means that Curt rejects the sentence `Mia cleans Vincent`. Change the lexical knowledge so that this sentence (and sentences like `Mia cleans a plant`) are accepted, but sentences like `Mia cleans a footmassage` are rejected.

Exercise 6.6.4 We would like Knowledgeable Curt to classify the discourse `Mia is Vincent's wife`. `Mia is married`. as uninformative. But although we have the relevant axioms, the first sentence makes use of the genitive construction (`Vincent's wife`) and this is not covered by the grammar. Modify the grammar so that such examples are covered, integrate your work into the inference architecture, and check that the above example is correctly handled.

Exercise 6.6.5 The sentence `Every boxer has a broken nose` has two logically distinct readings, but world knowledge rules one of these out as biologically implausible. Make the changes necessary to ensure that Curt rejects the unwanted reading in a situation with two boxers or more.

Exercise 6.6.6 Compare the performance of Knowledgeable Curt on the `Every boxer has a broken nose` example when Bliksem and Otter, respectively, are used as Curt's theorem prover.

Exercise 6.6.7 Design a practical Prolog representation format for the ontology for nouns. Then write a Prolog predicate that automatically compiles the ontology into first-order axioms.

Exercise 6.6.8 Use WordNet (see the Notes at the end of the chapter) as a source to automatically generate an ontology for the nouns in the lexicon, using the hypernym relation defined in WordNet.

Exercise 6.6.9 Use WordNet as a source to automatically generate disjointness relations for the adjectives in the lexicon, using the antonym relation defined in WordNet.

Exercise 6.6.10 Recall that Rugrat Curt didn't know how to cope with equality. A simple way to add equality reasoning to our home-brewed provers is to add background knowledge that explains to Curt what equality means. This can be done by adding the first-order equality axioms:

$$\begin{aligned} \forall x(x = x) & \text{ (reflexivity)} \\ \forall x\forall y(x = y \rightarrow y = x) & \text{ (symmetry)} \\ \forall x\forall y\forall z(x = y \wedge y = z \rightarrow x = z) & \text{ (transitivity)} \end{aligned}$$

These three axioms are strong enough to infer everything that holds of the equality relation in a first-order language, so by adding them to Rugrat Curt's background knowledge, in effect we convert our first-order theorem provers into theorem provers for first-order logic plus equality.

Try this out. How well does it work in practice?

Programs for Knowledgeable Curt

`knowledgeableCurt.pl`

This file loads everything needed to run Knowledgeable Curt.

`backgroundKnowledge.pl`

Contains all the predicates for computing background knowledge.

6.7 Helpful Curt

The CURTs we have met so far only allow us to assert information. There is no way to query information, or at least no natural way. Helpful CURT is a little more gracious:

> Vincent knows every woman.

CURT: OK.

> Mia is a woman.

CURT: OK.

> Who knows Mia?

CURT: This question makes sense!

CURT: vincent

Roughly speaking, Helpful CURT works as follows. It takes the user's wh-question, translates it into a quasi-logical form, uses the model

checker developed in Chapter 1 to query the discourse model it has built, and then generates a noun phrase that expresses the model checker's response. But this description is only a first approximation: for reasons we shall discuss below, we're also going to call the theorem prover to see whether our answers are "fully warranted". But this is jumping ahead—for the time being, let's focus on the central idea. We want to hook Curt up to our model checker so that it can answer questions. How do we go about doing this?

First the representational issue. Questions, unlike assertions, don't have truth-values, so it would be misleading to represent them as ordinary formulas. We will instead use a simple quasi-logical format to represent their content. Here's an example. We will represent the wh-question Who shot Marvin? by:

```
que(X, person(X), shot(X, marvin)).
```

We represent all wh-questions using this format: a principal variable (here **X**), the restriction (here **person(X)**), and the nuclear scope (here **shot(X,marvin)**). This is similar to the way we represented the quantifiers and the lambda operator in Prolog, and the resemblance is intentional. Like the quantifiers and lambda, our question marker **que** is a variable binder: **que** binds the free occurrences of the principal variable **X** in both the restriction and the nuclear scope.

The semantic role of the restriction and nuclear scope should be intuitively clear: the restriction represents what is being asked for, and the nuclear scope supplies additional information that must hold of this entity. Here are some more examples to think about:

Who did not shoot Marvin?

```
que(X, person(X), not(shot(X, marvin)))
```

Which customer ordered a five dollar shake?

```
que(Y, customer(Y), some(X, and(fdshake(X), order(Y, X))))
```

Which man or woman knows Butch?

```
que(X, or(man(X), woman(X)), know(X, butch)).
```

We have called these representations quasi-logical forms: they are not first-order formulas, hence we can't interpret them directly or (more to the point) use them directly with inference tools. But we *almost* can. The meaning of questions can be thought of in first-order logical terms if we adopt the following two-step perspective: first, translate **que(X,R,S)** into the first-order sentence **some(X, and(R,S))**, and use this to check whether the question makes sense. Second, query the model with the matrix of this first-order formula (that is, **and(R,S)**) to find suitable instantiations for the free variable **X**. Let's discuss this two-step process

more carefully, for it lies at the heart of Helpful Curt.

The first step is to check that the sentence makes sense. We do so by giving (the revised version of) the model checker from Chapter 1 the model so far built of the discourse, and then making the query `some(X, and(R,S))` about this model. If our model checker returns the result undefined (`undef`), then we know that we can't say anything useful about this question. Here's an example of a dialogue where Helpful Curt runs into this situation:

```
> Vincent loves every woman.  
Message (consistency checking): mace found a result.  
Curt: OK.  
  
> Who is a plant?  
Curt: I have no idea.
```

What's going on? With the first sentence we tell Curt something about Vincent and his relation to women, and the model builder (here Mace) successfully builds a model of the (consistent) situation described by the input. Now, because Helpful Curt is an extension of Knowledgeable Curt, this model will build in a lot of background information supplied by the lexical and world knowledge components. Nonetheless, there is simply no reason for Curt to build in any information about plants on the basis of the input sentence, so it doesn't do so. But this means that the discourse model Mace constructs won't be of the correct signature to handle queries about plants, and when the model checker is asked to do so it returns `undef`. This explains Curt's (quite reasonable) reaction: the question, with its abrupt change of topic, comes as a shock, and is (quite rightly) dismissed.

So, that's what happens if the model builder returns `undef`. But what if the result is positive (`pos`) or negative (`neg`)? This tells that we *can* answer the question, either positively or negatively. Now, if the result is negative, then Curt will simply say "none", and this is fine. But if the response is positive then (because we are dealing with wh-questions) the answer "yes" would be insufficient: in the positive case we need to provide answers like "Mia" or "Mia and Vincent". How do we do this? We use our model checker once more—but this time, rather than querying with the sentence `some(X, and(R,S))`, we throw away the existential quantifier (thereby freeing `X`) and query as follows:

```
?- satisfy(and(R,S),Model,[g(X,A)],pos).
```

This query will unify `A` with an entity in the model that satisfies both `R` and `S`—precisely what is required to generate an appropriate positive answer to a wh-question.

How is this implemented? The key predicate is `answerQuestion/3`, which takes the question representation and the discourse model as arguments and returns a set of reply-moves:

```
answerQuestion(que(X,R,S),Models,Moves) :-  
  (  
    Models=[Model|_],  
    satisfy(some(X, and(R,S)), Model, [], Result),  
    \+ Result=undef,  
    !,  
    findall(A, satisfy(and(R,S), Model, [g(X,A)], pos), Answers),  
    realiseAnswer(Answers, que(X,R,S), Model, String),  
    Moves=[sensible_question, answer(String)]  
  ;  
    Moves=[unknown_answer]  
  ).
```

This predicate is constructed as a disjunction: the first part of the disjunct deals with sensible questions (reply-moves: sensible-question plus the answer), the second part deals with questions that cannot be answered by Curt (reply-move: unknown-answer). It clearly shows that the model checker is used twice: first to check whether the question makes sense, and then (as an argument to `findall/3`) to collect the entities that satisfy the query.

So far so good—but now we need to go a little deeper. We have explained the core ideas involved in using a model checker to answer questions, but question answering is a subtle business. Is querying a discourse model really all that is involved in question answering? The following observation should give us a pause for thought: discourse models show a *possible* picture of the world. The pictures they give show us the way the agent imagines them to be: this need not correspond to the way things actually are. Moreover, in the case of Curt, the ‘agent’ generating the discourse model is not a person blessed with a human being’s sophisticated array of cognitive abilities: it’s just a Prolog program calling a model builder.

Here’s a simple example of how things can go wrong. Suppose we tell Helpful Curt that Mia or Jody dances. Depending on the model builder used, it could be that the discourse model shows that either Mia dances (and Jody does not) or that Jody dances (and Mia does not) or that both dance. In such cases, if Curt simply uses the model checker to inspect the discourse, then replying either “Mia” or “Jody” goes beyond what is fully warranted on the basis of the information supplied by the input sentence (together with the background knowledge).

Now, we can't resolve all the issues that such examples raise, but we can do something. We used the phrase "fully warranted on the basis of the information supplied by the input sentence (together with the background knowledge)". And this is something we can test for: we can use a theorem prover to see whether the entity the model checker selects is guaranteed to be an answer (given the discourse so far and the background knowledge) or whether it is merely a possible answer (because of the particular choices the discourse model embodies). Building in such a check gives rise to dialogues like this:

> **Mia or Jody dances.**

Message (consistency checking): mace found a result.

Curt: OK.

> **Who dances?**

Message (answer checking): unknown found result "unknown".

Curt: This question makes sense!

Curt: maybe jody

Let's spell out what happened here. On the basis of the input sentence **Mia or Jody dances** (a disjunction) the model builder built a discourse model in which **Jody dances** is true. Hence when asked **Who dances?** by the user, Curt uses the model checker and finds that **Jody** is a candidate answer. But now Helpful Curt lives up to his name: perhaps **Jody** is only a candidate answer because of the peculiarities of the particular discourse model built? Helpful Curt checks this possibility by using the theorem prover to see whether **Jody dances** follows logically from the discourse so far and the background knowledge. In fact it doesn't—that is, the answer is not fully warranted—hence Helpful Curt hedges its bets and responds **maybe Jody**. Here's how the required check is implemented:

```
checkAnswer(Answer,Proof):-  
    readings([F|_]),  
    backgroundKnowledge(F,BK),  
    callTP(imp(and(F,BK),Answer),Proof,Engine),  
    format('`nMessage (answer checking): \c  
          ~p found result ``~p''.,[Engine,Proof]).
```

Now we are ready to examine the final step: how do we generate answers on the basis of the model checker's response. Rather than outputting the sort of response the model checker gives us (like d3) we want Curt to generate a noun phrase that names, or at least describes, this entity. How do we do this?

The wh-questions we deal with only have noun phrases as possible answers. Helpful Curt only generates two kinds of noun phrases, proper names and indefinite noun phrases (in Exercise 6.7.5 we ask the reader to extend Helpful Curt so that it can generate definite descriptions as well). How do we choose between proper names and indefinite noun phrases? We shall use the following rule of thumb: generating a proper name is more informative than generating an indefinite noun phrase (that is, we assume that **Mia** will generally be better as an answer than **a woman**, given that we know that the entity chosen by the model checker is indeed called **Mia** and is indeed a woman).

So our strategy will be to first attempt to generate a proper name for entities of the model's domain. The following code does this:

```
realiseString(que(X,R,S),Value,Model,String):-  
    lexEntry(pn,[symbol:Symbol,syntax:Answer|_]),  
    satisfy(eq(Y,Symbol),Model,[g(Y,Value)],pos), !,  
    checkAnswer(some(X, and(eq(X,Symbol),and(R,S))),Proof),  
    (  
        Proof=proof, !,  
        list2string(Answer,String)  
    ;  
        list2string([maybe|Answer],String)  
    ).
```

This clause is pretty straightforward. First it does a lexical look-up for a proper name **Answer** with the constant **Symbol**. Then it uses the model checker (note: this is the third round of calls to the model checker!) to see if this symbol is one that matches the entity we want to generate. It does so by posing a simple equality query to the model checker. Using backtracking, it keeps on attempting to find a suitable proper name until it either succeeds (in which case we propose the proper name it found as answer) or fails (in which case we start trying to generate an indefinite noun phrase). Suppose it succeeds. Then it uses the theorem prover to check whether the answer is fully warranted on the basis of the discourse so far and the background knowledge; this is done using a call to **checkAnswer/2** as we discussed above. If the answer is not fully warranted, Curt adds a “maybe” before the proper name, otherwise it responds with the proper name alone. As **Answer** is declared in the lexicon to be a list of atoms, Helpful Curt builds the final response via a call to **list2string/2**.

Then, if we fail to find a proper name for the entity given to us by the model checker, we try to generate an indefinite noun phrase that describes it instead. Here's how:

```
realiseString(que(X,R,S),Value,Model,String):-
```

```

lexEntry(noun,[symbol:Symbol,syntax:Answer|_]),
compose(Formula,Symbol,[X]),
satisfy(Formula,Model,[g(X,Value)],pos), !,
checkAnswer(some(X, and(Formula, and(R,S))),Proof),
(
  Proof=proof, !,
  list2string([a|Answer],String)
;
  list2string([maybe,a|Answer],String)
).

```

This clause is very similar to the previous one. We start with a lexical look-up for nouns (again we use backtracking to find a match), then using the model checker we check whether the entity we're trying to describe has the property the noun expresses. If it does, we use the theorem prover to see whether it is fully warranted (and add a "maybe" if it is not) and use `list2string/2` to output the final response as a string.

We're nearly there. All that remains is to define a high level predicate that wraps the calls to `realiseString/4` up:

```

realiseAnswer([],_,_,'_none').

realiseAnswer([Value],Q,Model,String):-  

  realiseString(Q,Value,Model,String).

realiseAnswer([Value1,Value2|Values],Q,Model,String):-  

  realiseString(Q,Value1,Model,String1),  

  realiseAnswer([Value2|Values],Q,Model,String2),  

  list2string([String1, and, String2],String).

```

The first clause deals with the case where no satisfying entity is found, the second where one is found, and the third where several are found. And that's Helpful Curt (which you will find in the file `helpfulCurt.pl`).

Exercise 6.7.1 Some of Curt's answers (formed by nouns) to questions are not as specific as they could be. Change the implementation of the realisation of answers so that it gives the most specific possible answer.

Exercise 6.7.2 The generation of nouns as coded by `realiseString/4` doesn't necessarily generate the most informative answer. Play around with Helpful Curt and identify cases where funny answers are generated and explain the problem. Propose a solution to overcome this. Implement the solution proposed in the previous exercise.

Exercise 6.7.3 Use the restriction of questions to generate more appropriate answers. For instance, Curt's default negative answer to wh-questions is none, but a more appropriate negative answer to a Who-question would be nobody.

Exercise 6.7.4 Use the model checker to generate answers expressed by universally quantified noun phrases such as everybody and everything.

Exercise 6.7.5 The clause of `realiseString/4` implementing nouns always generates nouns following an indefinite article. But answers can sometimes be made more precise by generating a definite article instead. Discuss the situations in which this is the case and provide an additional clause for `realiseString/4` that implements this.

Exercise 6.7.6 Extend the grammar fragment and main predicate in such a way that yes-no questions are covered as well. Would you use the theorem prover, the model builder, or the model checker (or all of them) to generate an appropriate answer?

Exercise 6.7.7 Our grammar for English implements wh-questions by using the gap-threading technique to pass on the variable of the wh-phrase down the syntax tree, as required in Who did Vincent kill?. Inspect the file `englishGrammar.pl` to find out how exactly this is done.

Programs for Helpful Curt

`helpfulCurt.pl`

This file loads everything needed to run Helpful Curt.

Notes

Much of the content of this chapter requires little comment. In particular, as we remarked in the Introduction, the idea of using logic as a tool for representation and inference in natural language is not novel—it's a mainstay of classical AI. But it is worth reflecting on the way we developed this idea: by combining the best. The Curt systems combine techniques for semantic construction developed by the formal semantics community with the sophisticated theorem provers developed by the automated reasoning community. Although the Curt systems are merely simple programs designed for teaching purposes, we hope that

the methodology of “combining the best available” underlying their design comes through clearly. We believe that this approach will be crucial for the development of more sophisticated architectures for computational semantics.

While the use of theorem proving in natural language processing is old hat, the use of model builders is a recent development. In spite of this, model building has been used for a surprisingly wide range of linguistic applications. For the use of model builders to construct models of ongoing discourses, see Ramsay and Seville (2000) and Blackburn and Bos (2003). First-order model building has also been used to extract information from spoken dialogues: Bos and Oka (2002) describe a spoken dialogue system for activating domestic appliances in an intelligent house, and essentially the same architecture is used to control a mobile robot with voice commands in Bos et al. (2003). At present these systems only perform in reasonable time on relatively small domains, but current research is starting to address ways of overcoming these limitations, such as tuning model builders (and theorem provers) to linguistic problems, making the model building process incremental, and using machine learning techniques to estimate domain sizes (Bos, 2003). Model builders have also been proposed for the resolution of natural language ambiguity (see Gardent and Konrad (2000a), Gardent and Webber (2001), and Cimiano (2003)), by exploiting the concept of *minimal models* (that is, by looking for the smallest model of some described situation). In addition to these practically-oriented papers, there is also recent theoretical work on model building for natural language understanding; see Baumgartner and Kühn (1999), Kohlhase (2000) and Kohlhase and Koller (2003). Finally, it's worth remarking that there are similarities between model building and an inference technique called *abduction*. Abduction can be thought of as using deductive rules backwards to provide explanations; for example, an abductive inference might use the information $p \rightarrow q$ and q to hypothesise p as a plausible explanation for q (after all, if p holds, then q follows deductively from the given information). Search for explanations is clearly an important form of inference, and in fact Hobbs et al. (1990) suggests that abduction should be the inferential backbone of computational semantics. The use of model builders to “guess” pictures of the world can yield results similar to abduction; see (Blackburn and Bos, 2003) for further discussion.

One of the biggest omissions in this book is that we have said almost nothing about lexical semantics (the meaning of words). The nearest we came to it was the brief discussion of lexical knowledge that accompanied our description of Knowledgeable Curt. There we noted that words

are often semantically related in systematic ways. Some approaches to lexical semantics use this observation to try to *decompose* the lexical semantics of words into more elementary units of meaning. The tradition of lexical decomposition traces back to at least Jakobson (1936); it was introduced into transformational grammar in Katz and Fodor (1963), and into formal semantics in Dowty (1979). Another aspect of lexical semantics is the analysis of *thematic roles* of individuals in events or states described in utterances. Good starting points to explore this terrain are Fillmore (1968), Jackendoff (1990) and recent work on Berkely's FrameNet project (Baker et al., 2003). This hardly exhausts the field however: other important directions in lexical semantics are explored in Pustejovsky (1995) and Levin (1995). For a computationally-oriented introduction to the subject, see Chapters 16 and 17 of Jurafsky and Martin (2000).

Recall that we used a simple ontology to provide background knowledge for Curt. A linguistic ontology widely used in natural language processing is WordNet (see Miller (1995) and Fellbaum (1998)). WordNet is a lexical database whose design was inspired by psycholinguistic theories of human lexical memory. WordNet contains English nouns, verbs, adjectives and adverbs, organised into synonym sets ('synsets', as they are called in WordNet). Each synset represents an underlying lexical concept, and different relations (such as hypernym, hyponym, holonym, meronym, and antonym) link the synsets. Several related initiatives for languages other than English have been undertaken, a fine example being EuroWordNet (Vossen, 1998), designed for several European languages.

Another example of an ontology is Cyc (www.cyc.org), a manually-coded database of common-sense knowledge initiated in the early 1980s by Doug Lenat. Cyc covers areas ranging from biology and physics to history and politics, with the ultimate goal of enabling AI applications to perform human-like reasoning. The full version of the database (a registered trademark owned by Cycorp) contains over 120,000 concepts but is proprietary. OpenCyc (www.opencyc.org) is a limited version (containing some 6,000 concepts and 60,000 facts) released under an open-source licence. It is interesting to note that there have been recent attempts to link up WordNet with Cyc, and that the Cyc framework supplies tools for natural language processing as well.

Recently there has been upsurge of interest in ontology engineering, partly because of efforts to develop the Semantic Web. However, the logical formalism most closely associated with this new ontological wave is not first-order logic but (various kinds of) description logic (Baader et al., 2003). Description logics are, in essence, decidable fragments of

first-order logic, written in a simpler notation. Moreover, description logic notation encourages users to think about knowledge in a structured way: users have to provide both what is called a TBox (which, roughly speaking, contains what we called lexical and world knowledge) and an ABox (which contains what we called situational knowledge). A number of excellent theorem provers dedicated to description logics are available. Two of the best (and best supported) are Fact (Horrocks, 1988) and Racer (Haarslev and Möller, 2003). Finally, ideas from description logic (and first-order logic) have been used to provide logical underpinnings for Semantic Web formalisms such as Web Ontology Language (OWL); see <http://www.w3.org/2004/OWL/>.

But there is more to logics for knowledge representation than description logic (or first-order logic for that matter). There are a number of interrelated issues that we have not discussed in this book, such as how to adjust one's belief when incoming knowledge is inconsistent with what one already knows (the Curt system merely rejects the latest utterance, but is this always the best strategy?) or how to make inferential 'leaps' on the basis of partial knowledge. There are many theories and formalisms which attempt to capture such ideas. For example, the theory of belief revision analyses in detail how to update knowledge in the face of incoming information that may be inconsistent with what is already known; see Gärdenfors (1988). Various kinds of default and non-monotonic logics, on the other hand, deal with inferential leaps. An example is the leap to "Tweety flies" from the information that "Tweety is a bird". Usually this leap is fine—but what if Tweety is a kiwi? These brief comments only scratch the surface of a vast area, with many intersecting research themes. For a detailed application of non-monotonic logic to discourse semantics, see Asher and Lascarides (2003). For the relevance of default logic to the semantics of generics, see Pelletier and Asher (1997). For a general overview of non-monotonicity in linguistics, see Thomason (1997).

Knowledgeable Curt used model checking and theorem proving to answer simple questions—and as probably became clear while working through this material, there are a lot of subtleties in question answering. There is an interesting literature on both the theoretical and practical issues involved. Good starting points for studying the semantics of questions are Groenendijk and Stokhof (1997), from the *Handbook of Logic and Language* (Van Benthem and Ter Meulen, 1997), and Higginbotham (1997), from the *Handbook of Contemporary Semantic Theory* (Lappin, 1997). Other key references are Hamblin (1973), Karttunen (1977), and Ginzburg and Sag (2001). Approaches that interpret questions using automated inference or other logical tools can be found

in Bos and Gabsdil (2000), Gaylard and Ramsay (2004), and Ten Cate and Shan (2002). For the use of ideas of computational semantics in wide-coverage question answering systems, see Moldovan et al. (2003). And we recommend the overview article on automated question answering by Hirschman and Gaizauskas (2001), who present the history, the current trends, and possible future developments in question answering. For more information on the gap-threading technique used to handle the syntax of wh-questions, see Pereira and Shieber (1987).

That concludes our discussion of material that is related to this chapter, but to conclude these Notes—and the book—let's move on from what is known, to what is not. We shall present a short wish list, noting a number of research directions and themes that we believe are important to pursue.

New applications. More applications of the ideas of computational semantics need to be developed—there is nothing like applied work for sorting out the wheat from the chaff. Some applications (the intelligent house, the mobile robots) were noted above, but there is one area which seems particularly ripe for more systematic application of ideas from computational semantics, namely natural language generation. For a broad introduction to natural language generation, consult Reiter and Dale (2000). Interesting work on the generation of nominal expressions (that is, the task of mapping domain entities to natural language descriptions) already exists; see Stone (2000), Gardent (2002), Van Deemter (2004), and Striegnitz (2004). It will be interesting to see whether computational semantics techniques can be applied to other generation tasks.

Statistics and computational semantics. As we said in the Introduction, we don't feel that there is any tension between logically-inspired approaches to computational semantics, and the statistical orientation of much contemporary computational linguistics. The approach to semantic construction taught in this book has been used in tandem with a statistical parser to produce wide coverage semantics (Bos et al., 2004). Actually, the semantic representations generated in this work are rather more sophisticated than those of this book, and are based on the Discourse Representation Structures used in DRT (Kamp and Reyle, 1993) rather than first-order logic. But can architectures for representation and inference be devised to help develop “gold standard” annotated *semantic corpora*? (An example of a semantic corpus is being created by the PropBank project (Kingsbury et al., 2002): it is a corpus of text annotated with basic propositional information.) This is an important question to investigate.

Higher-order inference engines. Inference in this book has revolved around first-order logic, but there may well be interesting mileage in looking at alternatives. We have already mentioned some weaker logics (such as description logic) but there is also a stronger logic worth exploring, namely higher-order logic. As we mentioned in the Notes at the end of Chapter 2, an expression such as $\lambda x.\text{MAN}(x)$ is not only a useful piece of glue, it can also be regarded as a well-formed expression of higher-order logic with a model-theoretic interpretation. So it makes complete sense to say (for example) that $\lambda x.\text{PERSON}(x)$ is a logical consequence of $\lambda x.\text{MAN}(x)$. As this example shows, higher-order inference is potentially useful in computational semantics as it opens up the possibility of inference at the subsentential level: here we have an entailment relation between the nouns MAN and PERSON, and more generally the way is opened up for entailments between NPs, VPs, PPs and so on.

What are the prospects for higher-order inference in computational semantics? A difficult question. For a start, higher-order logic is not merely undecidable, it is not even possible to give complete proof systems for it. But (as we noted in Chapter 1) there is a loop-hole: by giving a non-standard semantics to higher-order logical notation (namely the famous Henkin semantics (Henkin, 1950)) one can obtain a first-order perspective on higher-order logic, and under this perspective complete proof systems *do* exist.

Does this open the door to automating (a form of) higher-order reasoning? Yes, but formidable difficulties remain. To give an idea of what is involved, recall from Chapter 5 that extending first-order theorem provers to cope efficiently with the equality symbol is a non-trivial task. Matters are far worse when we have to add not merely the humble equality symbol, but all the notational apparatus of higher-order logic. In particular, the process of unification becomes much more complex—indeed undecidable (see Lucchesi (1972) and Huet (1973)). As we learned in Chapter 5, unification is the key to efficient implementations of first-order theorem proving. If unification itself has become undecidable, we know we're in trouble.

In spite of this, a number of logicians and computer scientists have risen to the challenge. Important theoretical contributions include Huet (1972), which encodes the required unifications as constraints and delays application of higher-order unification, and Huet (1975). For detailed theoretical overviews, see Andrews (2001) and Dowek (2001). Moreover, implementations exist, for example the widely available TPS system (Andrews et al., 1996), and the LEO prover (Benzmüller, 1999). Another interesting system is Kimba, a higher-order model generator

that has been applied to linguistic applications; see Konrad (2000) and Gardent and Konrad (2000b). It remains to be seen how much impact higher-order inference methods will have on computational semantics, but it is definitely an area to keep an eye on.

Test suites for inference in computational semantics. In this book we provided baby test suites for testing our more important programs. We did this for two reasons. For a start, we found them helpful when updating programs (retyping examples is a pain; better to keep them together in a file). More importantly however, we wanted to encourage the reader to “think test suite!”, for test suites have proved important in a number of fields. A classic example is automated reasoning. One of the key resources of this community are large collections of problems, categorised and subdivided along a number of dimensions. This makes it easy to compare new provers with old, and to experiment with the practical impact of theoretical innovations.

Could test suites play an important role in computational semantics? In fact, the concept of test suites for computational semantics was introduced in the FraCaS project (Cooper et al., 1996) and some test suites for inference were developed; see

<http://www.ling.gu.se/projekt/nordsem/description/node8.html>.

More recently, Claire Gardent and Bonnie Webber have argued that the large scale development of test suites is a key task for computational semantics (see, for example, the remarks in Gardent and Webber (2001)). Their point is this: it is all very well talking about the need for inference in computational semantics, but in reality we have only imprecise ideas about which types of inference are truly important. A fair comment. In this book we gave simple (and sometimes rather artificial) examples involving consistency and informativity checking—but if we are ever to understand the role of inference in computational semantics we need to do far better. A detailed, theory neutral, collection of inference problems could be used to evaluate both theories and computational implementations for their coverage and accuracy, and might open up further interesting possibilities (for example, tuning theorem provers and model builders to cope with that style of problem). The difficulty of constructing such a test suite should not be underestimated, but the potential rewards are great.

Theoretical computational semantics. This book has presented computational semantics from a resolutely practical perspective—but just because we presented it this way doesn’t mean that we view theory as unimportant. On the contrary, we believe that theory and practice should go hand-in-hand in computational semantics, as in other fields.

What sort of topics might be viewed as “theoretical computational semantics”? A classic example is Richard Montague’s result (from “Universal Grammar”) that under certain general conditions, intermediate logical representations can be eliminated: fragments of natural language can be given a direct model theoretic interpretation. A more recent example can be found in Pratt-Hartmann (2004). In this book we translated into first-order logic, and in Chapter 5 we were careful to emphasise the difficulties that arise because of the undecidability of first-order logic. But this is a rather crude way of looking at things—exactly where in natural language does inferential complexity arise? And how much arises? Pratt-Hartmann’s paper provides some fine-grained answers to such questions. He takes simple Montague-style fragments, successively adds further syntactic constructions, and at each stage determines the computational complexity of the associated inference task (he does so by determining the computational complexity of the subset of first-order logic that each fragment translates into). He starts with a simple decidable (in fact, polynomial time) fragment capable of generating classical syllogisms (for example, All boxers are fighters. All fighters are stupid. So all boxers are stupid.) and works his way up to undecidable fragments. As far as we are aware, this is the first paper to determine the *intrinsic* computational complexity of inference for various fragments of natural language, and it is a good example of the sort of investigation that could help establish a deeper theoretical perspective on computational semantics.

A

Running the Software – FAQ

Where can I download the software?

All the Prolog programs discussed in this book are available for download at our website www.blackburnbos.org. To run them you will need a Prolog interpreter. We have tested our programs using two Prolog dialects: Sicstus Prolog (see www.sics.se/sicstus) and SWI Prolog (see www.swi-prolog.org/). Running Sicstus Prolog requires a licence, but SWI-Prolog is a free software Prolog compiler. Both can be downloaded by following the links just given.

Which platforms does it run on?

Our software was primarily designed and tested in Linux and Unix environments (including MacOS X). Running our programs directly under Microsoft Windows (or older Mac platforms) might require a little tweaking, but there is an indirect approach which works well: install the free software Linux emulator Cygwin (see www.cygwin.com/). All our software (and all the software the CURT programs make use of) runs under Cygwin.

Is it possible to run the CURT programs?

Yes, you can. All you need to do is install a little extra software.

First of all, you need a theorem prover and a model builder. The theorem provers Otter and Bliksem mentioned in the text are good ones to start with, as are the model builders Mace and Paradox. Our software has interfaces to all four of these inference engines. See Appendix C for instructions on downloading them.

Second, you need to be able to run Perl. This is because we use Perl scripts to hook up the theorem provers and model builders with our Prolog programs.

I've downloaded the software. How do I get started?

Simple. Create a new folder containing all the Prolog programs. Then start the Prolog interpreter (if you don't have a Prolog interpreter, install one first—see above). The Prolog prompt (`?-`) will appear, and you type

```
?- [menu].
```

and hit the return key. This will evoke a little menu (which follows the chapter-by-chapter structure of the book). Using this menu you can start any of the programs discussed in the text.

Alternatively, if you know the name of the program you want to work with, you don't need to go via the menu but can start it directly. For example, suppose you want to run `kellerStorage.pl`. What you do is start the Prolog interpreter, and then type

```
?- [kellerStorage].
```

to consult the program. All the options the program offers (for example, to run the test suite) are automatically displayed on start-up.

Formulas in Prolog notation are sometimes hard to read. Is there a simple way of displaying them in a prettier format?

Yes, there is. The software comes with a toggle between the internal Prolog representations (prefix format) and a more readable infix format. To switch to infix format, type

```
?- infix.
```

after you've loaded the program you want to work with. To switch back to prefix display mode, type:

```
?- prefix.
```

What if I want to know more about Prolog?

There are several introductions to Prolog available; take your pick. However, we'd like to draw your attention to *Learn Prolog Now!*, by Patrick Blackburn, Johan Bos, and Kristina Striegnitz. This was written in parallel with the present book, and it contains everything needed to understand the code presented here. Moreover *Learn Prolog Now!* was designed for self-study, so if you can't take a course in Prolog, it may be a good choice. *Learn Prolog Now!* is available free on the internet at www.learnprolognow.org.

B

Propositional Logic

The quantifier-free fragment of any first-order language (as the terminology suggests) simply consists of all formulas of the language that contain no occurrences of the symbols \exists or \forall . For example, ROBBER(PUMPKIN), CUSTOMER(MIA), CUSTOMER(y), and

$$\text{CUSTOMER}(y) \rightarrow \text{LOVE}(\text{VINCENT}, y)$$

are all quantifier-free formulas. On the other hand,

$$\forall y(\text{CUSTOMER}(y) \rightarrow \text{LOVE}(\text{VINCENT}, y))$$

clearly isn't, since it contains an occurrence of the quantifier \forall .

The key thing to note about quantifier-free formulas is the following. Suppose we are given a model M (of appropriate vocabulary) and an assignment g in M . Now, in full first-order logic we need to work with two semantic notions: satisfaction for arbitrary formulas and truth for sentences. However, when working with quantifier-free formulas, there are no bound variables to complicate matters, so this distinction is unnecessary. In fact, when working with a quantifier-free fragment, we may as well view each variable x as a constant interpreted by $g(x)$. If we do this, then every atomic quantifier-free formula is either true or false in M with respect to g . Moreover, it is obvious how to calculate the semantic value of complex sentences: conjunctions will be true if and only if both conjuncts are true, disjunctions will be true if and only if at least one disjunct is true, a negated formula will be true if and only if the formula itself is not true, and so on. (In short, we need to make truth table calculations, which many readers are doubtless familiar with, and which we shall review below.)

Another pleasant aspect of the quantifier-free fragment is that there is an obvious way to simplify our notation when working with it. Because we don't have quantifiers, the internal structure of atomic formulas is irrelevant, for we're never going to bind any free variables they

may contain. All that is important is whether the atomic symbols are true or false, and how they are joined together using the boolean connectives. For example, while it may be mnemonically helpful to choose propositional symbols such as CUSTOMER(x) or LOVE(VINCENT,MIA), we lose nothing if we replace them by simpler symbols such as p and q . This line of thought leads us to define *propositional logic*. To specify a language of propositional logic, we first say which symbols we are going to start with. (A fairly standard choice is p , q , r , s , t , and so on, often decorated with superscripts and subscripts: for example, p'' , r''' , or q_2 .) The chosen symbols are called proposition symbols, or sentence symbols. Complex sentences are built up using the boolean connectives \neg , \wedge , \vee and \rightarrow in the obvious way.

As we have already mentioned, we can calculate whether a complex sentence of propositional logic is true or not if we know the truth values of its component formulas; *truth tables* tell us how this is to be done. Here are the truth tables for \wedge , \vee and \rightarrow :

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$
TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

For example, the second row of this table tells us that if p is true and q is false then $p \wedge q$ is false, $p \vee q$ is true, and $p \rightarrow q$ is false. Compare this table with the satisfaction definition for full first-order logic given in Chapter 1. As you can see, this table contains all the information in the first-order satisfaction definition that is relevant to the simpler propositional notation.

The truth table for \neg is even simpler:

p	$\neg p$
TRUE	FALSE
FALSE	TRUE

Again, it is clear that this table contains all the information from the first-order satisfaction definition that is relevant to the simpler notation.

Truth tables can be used to test propositional formulas for validity. In Chapter 1 we said that a formula of first-order logic was valid if it was true in all models. Now, in propositional logic a model is essentially something that specifies the truth value of each proposition symbol. Hence a valid propositional formula is simply one that is true no matter what truth values the proposition symbols have. Truth tables are a

convenient way of systematically calculating all possible truth values.

Let's consider an example. The propositional formula

$$(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$$

is valid. This may not be obvious, so let's fill out a truth table for it and check (we'll shorten TRUE to T and FALSE to F). The first step is simply to systematically assign all possible combinations of truth values to the proposition symbols. The following table does this:

$(\neg p \rightarrow \neg q)$				\rightarrow	$(q \rightarrow p)$	
T		T			T	T
T		F			F	T
F		T			T	F
F		F			F	F

That is, each row of the truth table specifies a possible combination of truth values. Using this information, we can now calculate the possible truth values of $\neg p$, $\neg q$ and $q \rightarrow p$:

$(\neg p \rightarrow \neg q)$				\rightarrow	$(q \rightarrow p)$	
F	T	F	T		T	T
F	T	T	F		F	T
T	F	F	T		T	F
T	F	T	F		F	T

And now we can calculate the truth value of $\neg p \rightarrow \neg q$:

$(\neg p \rightarrow \neg q)$					\rightarrow	$(q \rightarrow p)$	
F	T	T	F	T		T	T
F	T	T	T	F		F	T
T	F	F	F	T		T	F
T	F	T	T	F		F	T

Finally, we can calculate the truth value of the entire formula:

$(\neg p \rightarrow \neg q)$					\rightarrow	$(q \rightarrow p)$	
F	T	T	F	T		T	T
F	T	T	T	F		F	T
T	F	F	F	T		T	F
T	F	T	T	F		F	T

As you can see, no matter what truth values are assigned to its propositional symbols, the formula itself is always true. That is (as we claimed above) it is indeed valid.

Summing up, propositional logic is essentially a simple notation for the quantifier-free formulas of first-order logic. When we explore inference mechanisms for first-order logic in the text, it turns out to be sensible to first investigate inference methods for the quantifier free

fragment (we do this in Chapter 4), and only then turn to the problem for the full first-order language (the task of Chapter 5). In Chapter 4, we make use of the simpler propositional notation just discussed.

C

Automated Reasoning for First-Order Logic

Automated reasoning tools have improved enormously over the last decade. This appendix is a guide to a number of first-order inference engines you may wish to experiment with for applications in computational semantics.

Theorem Provers for First-Order Logic

- **Bliksem**

<http://www.mpi-sb.mpg.de/~nivelle/software/bliksem>

Bliksem (the Dutch word for “lightning”) is an efficient resolution based theorem prover for first-order logic with equality, written by Hans de Nivelle. This prover, implemented in C, accepts formula syntax as input (our program `fol2bliksem.pl` translates between Bliksem syntax and the syntax used in this book).

- **FDPLL**

<http://www.uni-koblenz.de/~peter/FDPLL/>

A theorem prover for first-order clausal logic designed by Peter Baumgartner. The underlying calculus is a generalization of the Davis-Putnam-Loveland-Logemann Procedure (best known as a tool for complete SAT solving). FDPLL is written in Eclipse Prolog and runs on Unix systems.

- **Gandalf**

<http://www.ttu.ee/it/gandalf/>

Gandalf is really a family of automated theorem provers developed by Tanel Tammet. It contains classical, type theory, intuitionist and linear logic provers, plus a finite model builder (see below). Version c-2.5 contains the classical logic prover for clause form input and a finite

model builder.

- **Otter**

<http://www-unix.mcs.anl.gov/AR/otter/>

The classic theorem prover for first-order logic, developed by William McCune. Otter's inference rules are based on resolution and paramodulation, and it has a wide range of advanced facilities. Recent books on Otter are Wos and Pieper (2000) and Kalman (2001).

- **Scott**

<http://discus.anu.edu.au/software/scott/>

Scott is a combination of the theorem prover Otter and the model builder Finder. The models are used to guide selection of given clauses, clauses which are false in the guiding models being given preference. Scott is developed by John Slaney and Kahlil Hodgson.

- **Spass**

<http://spass.mpi-sb.mpg.de/>

A powerful theorem prover for first-order logic with equality developed at the Max-Planck-Institut für Informatik in Saarbrücken, Germany. Spass accepts input formulas in first-order syntax in DFG notation.

- **Vampire**

<http://www.cs.man.ac.uk/%7Eriazanoa/Vampire/>

Probably the world's fastest theorem prover at the time of writing. Developed by Andrei Voronkov and Alexandre Riazanov, Vampire is a resolution based system for first-order logic with equality. It solves problems in the TPTP syntax in both CNF and full first-order logic syntax.

Model Builders for First-Order Logic

- **Gandalf**

<http://www.ttu.ee/it/gandalf/>

<http://www.ee.princeton.edu/~chaff/zchaff.php>

The finite model building component of Gandalf uses the zChaff propositional logic solver (developed by Lintao Zhang and Zhaojun Fu) as an external program. zChaff is a propositional logic solver implemented in C++.

- **Mace**

<http://www-unix.mcs.anl.gov/AR/mace/>

<http://www-unix.mcs.anl.gov/AR/mace4/>

Mace is short for "Model and Counter-Examples" and is a model

builder for first-order logic with equality, developed by William McCune. We use Mace 2.0 in this book. The related model builder MACE4 (also known as ICGNS) performs better on some classes of problems, but (at the time of writing) did not accept formula syntax as input.

- **Paradox**

<http://www.math.chalmers.se/~koen/paradox/>

Paradox, the other model builder used in this book, is a tool that processes first-order logic problems and tries to find finite-domain models for them. Paradox is written by Koen Claessen and Niklas Sörensson.

- **Satchmo**

<http://www.pms.informatik.uni-muenchen.de/software/>

Satchmo is a model generator for first-order theories implemented in Prolog. Satchmo compiles a clausal theory into a Prolog program which then generates models of the theory efficiently.

D

Notation

This appendix summarises the notation used for the logical symbols in the text (second column of the tables), in the Prolog programs (third column) and, when applicable, in the Prolog infix format (fourth column).

First-Order Logic

conjunction	$(\phi \wedge \psi)$	and(Phi,Psi)	(Phi & Psi)
disjunction	$(\phi \vee \psi)$	or(Phi,Psi)	(Phi v Psi)
implication	$(\phi \rightarrow \psi)$	imp(Phi,Psi)	(Phi > Psi)
negation	$\neg\phi$	not(Phi)	\sim Phi
universal quantif.	$\forall x\phi$	all(X,Phi)	all X Phi
existential quantif.	$\exists x\phi$	some(X,Phi)	some X Phi
equality	$\tau_1 = \tau_2$	eq(Tau1,Tau2)	Tau1 = Tau2

Lambda Calculus

application	$(\epsilon_1 @ \epsilon_2)$	app(E1,E2)	(E1 @ E2)
abstraction	$\lambda x.\epsilon$	lam(X,E)	lam X E

Storage

store	$\langle \phi, (\epsilon_1, i_1), \dots, (\epsilon_n, i_n) \rangle$	[Phi, bo(E ₁ , I ₁), ..., bo(E _n , I _n)]
-------	---	--

Questions

wh-question	?x(Φ, Ψ)	que(X,Phi,Psi)
yes/no-question	?Φ	que(Phi)

References

- Abelson, H. and G. Sussman. 1985. *Structure and Interpretation of Computer Programs*. MIT Press.
- Abiteboul, S., R. Hull, and V. Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
- Alshawi, H., ed. 1992. *The Core Language Engine*. MIT Press.
- Althaus, E., D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. 2003. An efficient graph algorithm for dominance constraints. *Journal of Algorithms* 48(1):194–219.
- Andrews, P. 2001. Classical Type Theory. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. II, chap. 15, pages 965–1007. Elsevier Science.
- Andrews, P., M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and X. Hongwi. 1996. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning* 16(3):321–353.
- Apt, K. 1995. *From Logic Programming to Prolog*. Prentice-Hall.
- Asher, N. and A. Lascarides. 2003. *Logics of Conversation*. Cambridge University Press.
- Baader, F., D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, eds. 2003. *The Description Logic Handbook*. Cambridge University Press.
- Baker, C.F., C.J. Fillmore, and B. Cronin. 2003. The Structure of the Framenet Database. *International Journal of Lexicography* 16(3):281–296.
- Barendregt, H. 1984. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 2nd edn.
- Barendregt, H. 1992. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, eds., *Handbook of Logic in Computer Science*, vol. 2, pages 117–310. Oxford University Press.
- Barwise, J. and R. Cooper. 1981. Generalized Quantifiers and Natural Language. *Linguistics and Philosophy* 4:159–219.
- Baumgartner, P. and M. Kühn. 1999. Abducting Coreference by Model Construction. In C. Monz and M. de Rijke, eds., *ICoS-1, Inference in Computational Semantics, Workshop Proceedings*, pages 21–38. Institute for Logic, Language and Computation (ILLC), Amsterdam.

- Bell, J. and M. Machover. 1977. *A Course in Mathematical Logic*. North Holland.
- Benzmüller, C. 1999. *On Equality and Extensionality in Higher-Order Automated Theorem Proving*. Ph.D. thesis, Department of Computer Science, University of the Saarland.
- Blackburn, P. 2000. Representation, Reasoning, and Relational Structures: a Hybrid Logic Manifesto. *Logic Journal of the IGPL* 8(3):339–365.
- Blackburn, P. and J. Bos. 2003. Computational Semantics. *Theoria. Revista De Teoria, Historia Y Fundamentos De La Ciencia* 18(46):27–45.
- Blackburn, P., J. Bos, M. Kohlhase, and H. De Nivelle. 2001a. Inference and Computational Semantics. In H. Bunt, R. Muskens, and E. Thijssse, eds., *Computing Meaning. Volume 2*, pages 11–28. Kluwer.
- Blackburn, P., M. De Rijke, and Y. Venema. 2001b. *Modal Logic*. Cambridge University Press.
- Bodirsky, M., D. Duchier, J. Niehren, and S. Miele. 2004. A new algorithm for normal dominance constraints. In J. Munro, ed., *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 59–67. New Orleans.
- Boolos, G. and R. Jeffrey. 1989. *Computability and Logic*. Cambridge University Press.
- Bos, J. 1996. Predicate Logic Unplugged. In P. Dekker and M. Stokhof, eds., *Proceedings of the Tenth Amsterdam Colloquium*, pages 133–143. ILLC/Department of Philosophy, University of Amsterdam.
- Bos, J. 2001. *Underspecification and Resolution in Discourse Semantics*. Ph.D. thesis, Department of Computational Linguistics, University of the Saarland.
- Bos, J. 2003. Exploring Model Building for Natural Language Understanding. In *ICoS-4, Inference in Computational Semantics, Workshop Proceedings*, pages 41–57. LORIA, Nancy, France.
- Bos, J. 2004. Computational Semantics in Discourse: Underspecification, Resolution and Inference. *Journal of Logic, Language and Information* 13(2):139–157.
- Bos, J., B. Buschbeck-Wolf, M. Dorna, and C.J. Rupp. 1998. Managing information at linguistic interfaces. In *The 17th International Conference on Computational Linguistics*, pages 160–166. Montreal.
- Bos, J., S. Clark, M. Steedman, J.R. Curran, and J. Hockenmaier. 2004. Wide-Coverage Semantic Representations from a CCG Parser. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING '04)*. Geneva, Switzerland.
- Bos, J. and M. Gabsdil. 2000. First-Order Inference and the Interpretation of Questions and Answers. In M. Poesio and D. Traum, eds., *Proceedings of Götaglog 2000, Fourth Workshop on the Semantics and Pragmatics of Dialogue*, pages 43–50.

- Bos, J., B. Gambäck, C. Lieske, Y. Mori, M. Pinkal, and K. Worm. 1996. Compositional Semantics in Verbmobil. In *The 16th International Conference on Computational Linguistics*, pages 131–136. Copenhagen, Denmark.
- Bos, J., E. Klein, and T. Oka. 2003. Meaningful Conversation with a Mobile Robot. In *Proceedings of the Research Note Sessions of the 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL'03)*, pages 71–74.
- Bos, J. and M. Kohlhase, eds. 2003. *Logic Journal of the IGPL: Inference in Computational Semantics*, vol. 11(4). Oxford University Press.
- Bos, J. and T. Oka. 2002. An Inference-based Approach to Dialogue System Design. In S.-C. Tseng, ed., *COLING 2002. Proceedings of the 19th International Conference on Computational Linguistics*, pages 113–119. Taipei, Taiwan.
- Bunt, H. and R. Muskens, eds. 1999. *Computing Meaning, Volume 1*. Studies in Linguistics and Philosophy. Kluwer: Dordrecht/Boston/London.
- Bunt, H., R. Muskens, and E. Thijssse, eds. 2001. *Computing Meaning, Volume 2*. Studies in Linguistics and Philosophy. Kluwer: Dordrecht/Boston/London.
- Burton-Roberts, N. 1986. *Analysing Sentences*. Longman.
- Carnap, R. 1947. *Meaning and Necessity*. The University of Chicago Press.
- Carpenter, B. 1997. *Type-Logical Semantics*. MIT Press.
- Chang, C.-L. and R.C.-T. Lee. 1973. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press.
- Charniak, E. and D. McDermott. 1985. *Introduction to Artificial Intelligence*. Addison-Wesley.
- Charniak, E. and Y. Wilks, eds. 1976. *Computational Semantics. An Introduction to Artificial Intelligence and Natural Language Comprehension*, vol. 4 of *Fundamental Studies in Computer Science*. North-Holland Publishing Company: Amsterdam, New York, Oxford.
- Chaves, R. 2003. Non-Redundant Scope Disambiguation in Underspecified Semantics. In B. Ten Cate, ed., *Proceedings of the 8th ESSLLI Student Session*, pages 47–58. Vienna, Austria.
- Cherniak, C. 1986. *Minimal Rationality*. MIT Press.
- Chierchia, G., B.H. Partee, and R. Turner, eds. 1989. *Properties, Types and Meaning. Volume 1. Foundational Issues*. Dordrecht: Kluwer.
- Church, A. 1940. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic* 5(2):56–68.
- Church, A. 1941. *The Calculi of Lambda Conversion*. Princeton University Press.
- Cimiano, P. 2003. Building Models for Bridges. In *ICoS-4, Inference in Computational Semantics, Workshop Proceedings*, pages 57–71. LORIA, Nancy, France.

- Claessen, K. and N. Sörensson. 2003. New Techniques that Improve MACE-style Model Finding. In *Model Computation – Principles, Algorithms, Applications (Cade-19 Workshop)*. Miami, Florida, USA.
- Clocksin, W. and C. Mellish. 1984. *Programming in Prolog*. Springer-Verlag, 2nd edn.
- Cook, S. and D. Mitchell. 1997. Finding hard instances of the satisfiability problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 35:1–17.
- Cooper, R. 1975. *Montague's Semantic Theory and Transformational Syntax*. Ph.D. thesis, University of Massachusetts.
- Cooper, R. 1983. *Quantification and Syntactic Theory*. Dordrecht: Reidel.
- Cooper, R., R. Crouch, J. Van Eijck, C. Fox, J. Van Genabith, J. Jaspars, H. Kamp, M. Pinkal, D. Milward, M. Poesio, and S. Pulman. 1996. Using the Framework. Tech. rep., FraCaS: A Framework for Computational Semantics. FraCaS deliverable D16.
- Cooper, R., I. Lewin, and A.W. Black. 1993. Prolog and Natural Language Semantics. Notes for AI3/4 Computational Semantics, University of Edinburgh.
- Copestake, A., D. Flickinger, R. Malouf, S. Riehemann, and I. Sag. 1995. Translation using Minimal Recursion Semantics. In *Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation*, pages 15–32. University of Leuven, Belgium.
- D'Agostino, M. 1992. Are tableaux an improvement on truth tables? *Journal of Logic, Language and Information* 1:235–252.
- D'Agostino, M., D. Gabbay, R. Hähnle, and J. Posegga, eds. 1999. *Handbook of Tableau Methods*. Kluwer Academic Publishers.
- Dalrymple, M., J. Lamping, F.C.N. Pereira, and V.A. Saraswat. 1997. Quantifiers, Anaphora, and Intentionality. *Journal of Logic, Language, and Information* 6:219–273.
- Davis, M., G. Logemann, and D. Loveland. 1962. A machine program for theorem-proving. *Communications of the ACM* 5(7):394–397.
- Davis, M. and H. Putnam. 1960. A computing procedure for quantification theory. *Journal of the ACM* 7(3):201–215.
- Degtyarev, A. and A. Voronkov. 2001. Equality Reasoning in Sequent-Based Calculi. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. I, chap. 10, pages 611–706. Elsevier Science.
- Doets, K. and J. Van Benthem. 1983. Higher-Order Logic. In D. Gabbay and F. Guenther, eds., *Handbook of Philosophical Logic*, vol. 1, pages 275–329. Reidel.
- Dowek, G. 2001. Higher-Order Unification and Matching. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. II, chap. 16, pages 1009–1062. Elsevier Science.
- Dowty, D.R. 1979. *Word Meaning and Montague Grammar*. Synthese Language Library. D. Reidel Publishing Company.

- Dowty, D.R., R.E. Wall, and S. Peters. 1981. *Introduction to Montague Semantics*. Dordrecht: Reidel.
- Duchier, D. and J. Niehren. 2000. Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, vol. 1861 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag.
- Egg, M., A. Koller, and J. Niehren. 2001a. The constraint language for lambda structures. *Journal of Logic, Language, and Information* 10(4):457–485.
- Egg, M., J. Niehren, P. Ruhrberg, and F. Xu. 1998. Constraints over Lambda-Structures in Semantic Underspecification. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics. Proceedings of the Conference*, pages 353–359. Université de Montréal, Montreal, Quebec, Canada.
- Egg, M., M. Pinkal, and J. Pustejovsky, eds. 2001b. *Journal of Logic, Language, and Information: Special Issue on Underspecification*, vol. 10(4). Kluwer.
- Enderton, H. 1977. *Elements of Set Theory*. Academic Press.
- Enderton, H. 2001. *A Mathematical Introduction to Logic*. New York and London: Academic Press, 2nd edn.
- Erk, K., A. Koller, and J. Niehren. 2003. Processing Underspecified Semantic Descriptions in the Constraint Language for Lambda Structures. *Journal of Research on Language and Computation* 1(1):127–169.
- Fellbaum, C., ed. 1998. *WordNet. An Electronic Lexical Database*. MIT Press.
- Fermüller, C., A. Leitsch, U. Hustadt, and T. Tammet. 2001. Resolution Decision Procedures. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. II, chap. 25, pages 1791–1849. Elsevier Science.
- Fillmore, C.J. 1968. The case for case. In E. Bach and R. Harms, eds., *Universals in Linguistic Theory*, pages 1–88. New York: Holt, Rinehart, and Winston.
- Fitting, M. 1996. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 2nd edn.
- Fitting, M. 2002. *Types, Tableaus and Gödel's God*. Kluwer.
- Franconi, E. 2002. Description Logics for Natural Language Processing. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, eds., *Description Logic Handbook*, pages 450–461. Cambridge University Press.
- Frege, G. 1892. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik* 100:25–50.
- Gabsdil, M. and K. Striegnitz. 2000. Classifying Scope Ambiguities. *Language and Computation* 1(2):307–313.
- Gallier, J. 1986. *Logic for Computer Science*. New York: Harper Row.
- Gallin, D. 1975. *Intentional and Higher-Order Modal Logic*. North Holland.

- Gamut, L.T.F. 1991a. *Logic, Language, and Meaning. Volume I. Introduction to Logic*. Chicago and London: The University of Chicago Press.
- Gamut, L.T.F. 1991b. *Logic, Language, and Meaning. Volume II. Intensional Logic and Logical Grammar*. Chicago and London: The University of Chicago Press.
- Gärdenfors, P. 1988. *Knowledge in Flux. Modelling the Dynamics of Epistemic States*. MIT Press.
- Gardent, C. 2002. Generating Minimal Definite Descriptions. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 96–103. Philadelphia, USA.
- Gardent, C. and E. Jacquay. 2003. Lexical reasoning. In *Proceedings of ICON'03 (International Conference on Natural Language Processing)*, Mysore, India.
- Gardent, C. and K. Konrad. 2000a. Interpreting Definites using Model Generation. *Journal of Language and Computation* 1(2):193–209.
- Gardent, C. and K. Konrad. 2000b. Understanding "Each Other". In *Proceedings of the First Annual Meeting of the North American Chapter of the Association for Computational Linguistics*, Seattle, pages 319–326.
- Gardent, C. and K. Striegnitz. 2003. Generating Bridging Definite Descriptions. Submitted manuscript.
- Gardent, C. and B.L. Webber. 2001. Towards the Use of Automated Reasoning in Discourse Disambiguation. *Journal of Logic, Language and Information* 10(4):487–509.
- Garey, M. and D. Johnson. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman.
- Gaylard, H. and A. Ramsay. 2004. Relevant Answers to WH-Questions. *Journal of Logic, Language and Information* 13(2):173–186.
- Gazdar, G. and C. Mellish. 1989. *Natural Language Processing in Prolog*. Addison-Wesley Publishing Company.
- Ginzburg, J. and I.A. Sag. 2001. *Interrogative Investigations: the form, meaning, and use of English Interrogatives*. Stanford, California: CSLI Publications.
- Grice, H.P. 1975. Logic and Conversation. In P. Cole and J. Morgan, eds., *Syntax and Semantics, Volume 3: Speech Acts*, pages 41–58. New York: Academic Press.
- Groenendijk, J. and M. Stokhof. 1991. Dynamic Predicate Logic. *Linguistics and Philosophy* 14:39–100.
- Groenendijk, J. and M. Stokhof. 1997. Questions. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 19, pages 1055–1124. MIT Press and North-Holland.
- Grosz, B.J., K. Sparck Jones, and B.L. Webber, eds. 1986. *Readings in Natural Language Processing*. Morgan Kaufmann.
- Gu, J. 1992. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin* 3(1):8–12.

- Gunji, T. 1981. *Towards a Computational Theory of Pragmatics — Discourse, Presupposition, and Implicature*. Ph.D. thesis, Ohio State University.
- Haarslev, V. and R. Möller. 2003. RACER User's Guide and Reference Manual version 1.7.7.
- Haken, A. 1985. The intractability of resolution. *Theoretical Computer Science* 39(2–3):297–308.
- Halpern, J.Y. and M.Y. Vardi. 1991. Model Checking vs. Theorem Proving: A Manifesto. In J. Allen, R. E. Fikes, and E. Sandewall, eds., *Proceedings 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning, KR'91*, pages 325–334. San Mateo, CA: Morgan Kaufmann Publishers.
- Hamblin, C.L. 1973. Questions in Montague Grammar. *Foundations of Language* 10:41–53.
- Harman, G. 1986. *Change in View*. MIT Press.
- Heim, I. 1982. *The Semantics of Definite and Indefinite Noun Phrases*. Ph.D. thesis, University of Massachusetts.
- Henkin, L. 1950. Completeness in the Theory of Types. *Journal of Symbolic Logic* 15(2):81–91.
- Higginbotham, J. 1997. The Semantics of Questions. In S. Lappin, ed., *Handbook of Contemporary Semantic Theory*, chap. 14, pages 361–384. Oxford.
- Hindley, J. and J. Seldin. 1986. *Introduction to Combinators and λ -Calculus*. London Mathematical Society Student Texts vol. 1. Cambridge University Press.
- Hirschman, L. and R. Gaizauskas. 2001. Natural language question answering: The view from here. *Natural Language Engineering* 7(4):275–300.
- Hobbs, J.R. and S. Rosenschein. 1978. Making computational sense of Montague's intensional logic. *Artificial Intelligence* 9(3):287–306.
- Hobbs, J.R. and S.M. Shieber. 1987. An algorithm for generating quantifier scopings. *Computational Linguistics* 13(1–2):47–55.
- Hobbs, J.R., M.E. Stickel, D. Appelt, and P. Martin. 1990. Interpretation as Abduction. Tech. Rep. 499, AI Center, SRI International, Menlo Park, California.
- Hodges, W. 1983. Elementary predicate logic. In D. Gabbay and F. Guenther, eds., *Handbook of Philosophical Logic. Volume I*, pages 1–131. Reidel.
- Horrocks, I. 1988. The FaCT System. In H. de Swart, ed., *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, no. 1397 in Lecture Notes in Artificial Intelligence, pages 307–312. Springer-Verlag.
- Huet, G. 1972. *Constrained Resolution: A Complete Method for Higher Order Logic*. Ph.D. thesis, Case Western Reserve University.
- Huet, G. 1973. The undecidability of unification in third order logic. *Information and Control* 22(3):257–267.

- Huet, G. 1975. A unification algorithm for typed lambda calculus. *Theoretical Computer Science* 1:27–57.
- Huth, M. and M. Ryan. 2000. *Logic in Computer Science*. Cambridge University Press.
- Jackendoff, R. 1990. *Semantic Structures*. MIT Press.
- Jakobson, R. 1936. Beitrag zur allgemeinen Kasuslehre. *Travaux de Cercle Linguistique de Prague* 6:240–299.
- Janssen, T.M.V. 1986a. *Foundations and Applications of Montague Grammar, Part 1: Philosophy, Framework, Computer Science*. CWI Tracts no. 19. Amsterdam: Centre for Mathematics and Computer Science.
- Janssen, T.M.V. 1986b. *Foundations and Applications of Montague Grammar, Part 2: Applications to Natural Language*. CWI Tracts no. 28. Amsterdam: Centre for Mathematics and Computer Science.
- Janssen, T.M.V. 1997. Compositionality. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 7, pages 417–473. Elsevier.
- Johnson, M. and M. Kay. 1990. Semantic Abstraction and Anaphora. In *COLING-90. Papers presented to the 13th International Conference on Computational Linguistics*, pages 17–27. University of Helsinki.
- Jurafsky, D. and J. Martin. 2000. *Speech and Language Processing*. Prentice-Hall.
- Kalman, J.A. 2001. *Automated Reasoning with OTTER*. Rinton Press.
- Kamp, H. 1984. A Theory of Truth and Semantic Representation. In J. Groenendijk, T. Janssen, and M. Stokhof, eds., *Truth, Interpretation and Information; Selected Papers from the Third Amsterdam Colloquium*, pages 1–41. Dordrecht - Holland/Cinnaminson - U.S.A.: Foris.
- Kamp, H. and U. Reyle. 1993. *From Discourse to Logic*. Dordrecht: Kluwer.
- Karttunen, L. 1977. Syntax and semantics of questions. *Linguistics and Philosophy* 1:3–44.
- Katz, J.J. and J.A. Fodor. 1963. The Structure of a Semantic Theory. *Language* 39:170–210.
- Keenan, E.L. and D. Westerståhl. 1997. Generalized quantifiers in linguistics and logic. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 15, pages 837–893. Elsevier Science.
- Keller, W.R. 1988. Nested Cooper Storage: The Proper Treatment of Quantification in Ordinary Noun Phrases. In U. Reyle and C. Rohrer, eds., *Natural Language Parsing and Linguistic Theories*, pages 432–447. Dordrecht: Reidel.
- Kingsbury, P., M. Palmer, and M. Marcus. 2002. Adding Semantic Annotation to the Penn TreeBank. In *Proceedings of the Human Language Technology Conference*. San Diego, California.
- Kohlhase, M. 2000. Model Generation for Discourse Representation Theory. In W. Horn, ed., *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 441–445. IOS Press.

- Kohlhase, M., ed. 2004. *Journal of Logic, Language and Information: Special Issue on Computational Semantics*, vol. 13(2). Kluwer.
- Kohlhase, M. and A. Koller. 2003. Resource-Adaptive Model Generation as a Performance Model. *Logic Journal of the IGPL* 11(4):435–456.
- Koller, A. 2004. *Constraint-based and graph-based resolution of ambiguities in natural language*. Ph.D. thesis, Department of Computer Science, University of the Saarland.
- Koller, A., R. Debusmann, M. Gabsdil, and K. Striegnitz. 2004. Put my galakmid coin into the dispenser and kick it: Computational linguistics and theorem proving in a computer game. *Journal of Logic, Language and Information* 13(2):187–206.
- Koller, A., K. Mehlhorn, and J. Niehren. 2000. A Polynomial-Time Fragment of Dominance Constraints. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL'2000)*, pages 368–375. Hong Kong.
- Koller, A., J. Niehren, and S. Thater. 2003. Bridging the gap between underspecification formalisms: Hole semantics as dominance constraints. In *Proceedings of the 11th Meeting of the European Chapter of the Association for Computational Linguistics (EACL'2003)*, pages 195–202. Budapest, Hungary.
- König, E. and U. Reyle. 1997. A General Reasoning Scheme for Underspecified Representations. In H. J. Ohlbach and U. Reyle, eds., *Logic and its Applications. Festschrift for Dov Gabbay. Part I*. Kluwer.
- Konrad, K. 2000. *Model Generation for Natural Language Interpretation and Analysis*. Ph.D. thesis, Department of Computer Science, University of the Saarland.
- Landsbergen, J. 1982. Machine translation based on logically isomorphic Montague Grammars. In J. Horecky, ed., *Proceedings of COLING 82*. North-Holland.
- Langholm, T. 1988. *Partiality, Truth and Persistence*. CSLI Publications.
- Lappin, S., ed. 1997. *The Handbook of Contemporary Semantic Theory*. Blackwell.
- Leitsch, A. 1997. *The Resolution Calculus*. Springer-Verlag.
- Letz, R. 1999. First-order Tableau Methods. In M. D'Agostino, D. Gabbay, R. Hahnle, and J. Posegga, eds., *Handbook of Tableau Methods*, pages 125–196. Kluwer Academic Publishers.
- Levin, B. 1995. Approaches to Lexical Semantic Representation. In D. Walker, A. Zampolli, and N. Calzolari, eds., *Automating the Lexicon I: Research and Practice in a Multilingual Environment*, pages 53–91. Oxford University Press, Oxford.
- Lewin, I. 1990. A Quantifier Scoping Algorithm without A Free Variable Constraint. In *Coling-90. Papers presented to the 13th International Conference on Computational Linguistics*, pages 190–193. University of Helsinki.

- Lønning, J.T. 1997. Plurals and collectivity. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 18, pages 1009–1053. Elsevier Science.
- Lucchesi, C. 1972. The undecidability of the unification problem for third-order languages. Tech. Rep. CSRR 2059, University of Waterloo, Waterloo, Canada.
- Main, M.G. and D.B. Benson. 1983. Denotational Semantics for "Natural" Language Question-Answering Programs. *American Journal of Computational Linguistics* 9(1):11–21.
- Martinich, A., ed. 1996. *The Philosophy of Language*. Oxford University Press.
- McCune, W. 1998. Automatic Proofs and Counterexamples for Some Ortholattice Identities. *Information Processing Letters* 65(6):285–291.
- Miller, G. 1995. WordNet: A lexical database for English. *Communications of the ACM* 38(11):39–41.
- Moldovan, D., M. Pasca, S. Harabagiu, and M. Surdeanu. 2003. Performance issues and error analysis in an open-domain question answering system. *ACM Trans. Inf. Syst.* 21(2):133–154.
- Montague, R. 1968. Pragmatics. In R. Klibansky, ed., *Contemporary Philosophy: a Survey*, pages 102–122. Florence, La Nuova Italia Editrice. Reprinted in Montague (1974), 95–118.
- Montague, R. 1969. On the nature of certain philosophical entities. *The Monist* 53:159–194. Reprinted in Montague (1974), 148–187.
- Montague, R. 1970a. English as a Formal Language. In B. Visentini et al., eds., *Linguaggi nella Società e nella Tecnica*, pages 189–224. Milan, Edizioni di Comunita. Reprinted in Montague (1974), 188–221.
- Montague, R. 1970b. Pragmatics and intensional logic. *Synthese* 22:68–94. Reprinted in Montague (1974), 119–147.
- Montague, R. 1970c. Universal Grammar. *Theoria* 36:373–398. Reprinted in Montague (1974), 222–246.
- Montague, R. 1973. The Proper Treatment of Quantification in Ordinary English. In J. Hintikka, J. Moravcsik, and P. Suppes, eds., *Approaches to Natural Language*, pages 221–242. Dordrecht: Reidel. Reprinted in Montague (1974), 247–270.
- Montague, R. 1974. *Formal Philosophy. Selected Papers of Richard Montague. Edited and with an Introduction by Richmond H. Thomason*. New Haven: Yale University Press.
- Monz, C. and M. De Rijke, eds. 2000. *Journal of Language and Computation: Special Issue on Inference in Computational Semantics*, vol. 1(2). Hermes.
- Moore, R.C. 1989. Unification-Based Semantic Interpretation. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 33–41. Vancouver, British Columbia, Canada.
- Moore, R.C. 1995. *Logic and Representation*. CSLI Publications.

- Muskens, R. 1996. *Meaning and Partiality*. Studies in Logic, Language and Information. CSLI Publications.
- Muskens, R., J. Van Benthem, and A. Visser. 1997. Dynamics. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 10, pages 587–648. Elsevier Science.
- Nerbonne, J. 1992a. A Feature-Based Syntax/Semantics Interface. Research Report RR-92-42, DFKI, Saarbrücken.
- Nerbonne, J. 1992b. Constraint-Based Semantics. In M. Stokhof and P. Dekker, eds., *Proceedings of the Eighth Amsterdam Colloquium*, pages 425–443. Institute for Logic, Language and Computation, University of Amsterdam.
- Nerbonne, J. 1997. Computational Semantics—Linguistics and Processing. In S. Lappin, ed., *The Handbook of Contemporary Semantic Theory*, chap. 17, pages 461–484. Blackwell.
- Nielson, H. and F. Nielson. 1992. *Semantics with Applications*. Johan Wiley and Sons Ltd.
- Nieuwenhuis, R. and A. Rubio. 2001. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. I, chap. 7, pages 371–443. Elsevier Science.
- Nilsson, N. 1980. *Principles of Artificial Intelligence*. Springer-Verlag.
- Pagin, P and D. Westerståhl, eds. 2001. *Journal of Logic, Language and Information: Special Issue on Compositionality*, vol. 10(1). Kluwer.
- Papadimitriou, C.H. 1994. *Computational Complexity*. Addison-Wesley.
- Partee, B.H. 1997a. Montague Grammar. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 1, pages 5–91. Elsevier Science.
- Partee, B.H. 1997b. The Development of Formal Semantics in Linguistic Theory. In S. Lappin, ed., *The Handbook of Contemporary Semantic Theory*, chap. 1, pages 11–38. Blackwell.
- Pelletier, F.J. 1982. Completely non-clausal, completely heuristically driven automatic theorem proving. Tech. rep., Department of Computing Science, University of Alberta.
- Pelletier, F.J. 1986. Seventy-five Problems for Testing Automatic Theorem provers. *Journal of Automated Reasoning* 2(2):191–216.
- Pelletier, F.J. and N. Asher. 1997. Generics and defaults. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 20, pages 1125–1177. Elsevier.
- Pereira, F.C.N. and S.M. Shieber. 1987. *Prolog and Natural Language Analysis*. CSLI Lecture Notes 10. Stanford: Chicago University Press.
- Player, N.J. 2004. *Logics of Ambiguity*. Ph.D. thesis, Department of Computer Science, University of Manchester.
- Posegga, J. and P. Schmitt. 1999. Implementing Semantic Tableau. In M. D'Agostino, D. Gabbay, R. Hahnle, and J. Posegga, eds., *Handbook of Tableau Methods*, pages 581–629. Kluwer Academic Publishers.

- Pratt-Hartmann, I. 2003. A two variable fragment of English. *Journal of Logic, Language and Information* 12(1):13–45.
- Pratt-Hartmann, I. 2004. Fragments of language. *Journal of Logic, Language and Information* 13(2):207–223.
- Pustejovsky, J. 1995. *The Generative Lexicon*. MIT Press.
- Radford, A. 1997. *Syntax: A Minimalist Introduction*. Cambridge University Press.
- Ramsay, A. and H. Seville. 2000. Models and Discourse Models. *Language and Computation* 1(2):167–181.
- Ranta, A. 1994. *Type-Theoretical Grammar*. Oxford: Clarendon Press.
- Reiter, E. and R. Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press.
- Reyle, U. 1992. Dealing with Ambiguities by Underspecification: A First Order Calculus for Unscoped Representations. In M. Stokhof and P. Dekker, eds., *Proceedings of the Eighth Amsterdam Colloquium*, pages 493–512. Institute for Logic, Language and Computation, University of Amsterdam.
- Reyle, U. 1993. Dealing with Ambiguities by Underspecification: Construction, Representation and Deduction. *Journal of Semantics* 10(2):123–179.
- Rich, E. and K. Knight. 1990. *Artificial Intelligence*. McGraw-Hill Higher Education.
- Robinson, A. and A. Voronkov, eds. 2001a. *Handbook of Automated Reasoning*, vol. I. Elsevier Science.
- Robinson, A. and A. Voronkov, eds. 2001b. *Handbook of Automated Reasoning*, vol. II. Elsevier Science.
- Robinson, J.A. 1965. A machine oriented logic based on the resolution principle. *Journal of the ACM* 12(1):23–41.
- Rosner, M and R. Johnson, eds. 1992. *Computational linguistics and formal semantics*. Studies in Natural Language Processing. Cambridge University Press.
- Russell, S. and P. Norvig. 1995. *Artificial Intelligence. A Modern Approach*. Prentice-Hall.
- Schubert, L.K. and F.J. Pelletier. 1982. From English to Logic: Context-Free Computation of ‘Conventional’ Logical Translation. *Computational Linguistics* 8(1):26–44.
- Schwartz, R.L. and T. Phoenix. 2001. *Learning Perl*. O'Reilly.
- Scott, D. 1970. Outline of a Mathematical Theory of Computation. Tech. rep., Programming Research Group, Oxford, England. Technical Monograph PRG-2.
- Scott, D. 1973. Models for various type-free calculi. In P. Suppes, L. Henkin, and A. Moisil, eds., *Logic and Philosophy of Science IV*, pages 157–187.
- Selman, B., H. Levesque, and D. Mitchell. 1992. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on AI*, pages 440–446. American Association for Artificial Intelligence.

- Shapiro, S. 1999. *Foundations Without Foundationalism: A Case for Second-order Logic*. Oxford University Press.
- Smullyan, R. 1995. *First-Order Logic*. Dover Publications. This is a reprinted and corrected version of the 1968 Springer-Verlag edition.
- Sterling, L. and E. Shapiro. 1986. *The Art of Prolog; Advanced Programming Techniques*. MIT Press.
- Stone, M. 2000. On Identifying Sets. In *Proceedings of the 1st International Conference on Natural Language Generation, INLG 2000*, pages 116–123.
- Stoy, J. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press.
- Striegnitz, K. 2004. *Generating Anaphoric Expressions*. Ph.D. thesis, Department of Computational Linguistics, University of the Saarland, and Department of Computer Science, University Henri Poincaré, Nancy.
- Sundholm, G. 1983. Systems of Deduction. In D. Gabbay and F. Guenther, eds., *Handbook of Philosophical Logic. Volume I*, pages 133–188. Reidel.
- Sundholm, G. 1986. Proof Theory and Meaning. In D. Gabbay and F. Guenther, eds., *Handbook of Philosophical Logic. Volume III*, pages 471–506. Reidel.
- Tarski, A. 1935. Der Wahrheitsbegriff in den formalisierten Sprachen (German translation of a book in Polish, 1933). *Studia philosophica* 1:261–405. English translation in A. Tarski, *Logic, Semantics, and Metamathematics*, Oxford University Press, 1956.
- Ten Cate, B. and C. Shan. 2002. Question answering: from partitions to prolog. In U. Egly and C. Fermüller, eds., *Proceedings of TABLEAUX 2002: Automated Reasoning with Analytic Tableaux and Related Methods*, Lecture Notes in Artificial Intelligence 2381, pages 251–265. Springer-Verlag.
- Thomason, R.T. 1997. Nonmonotonicity in linguistics. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, chap. 14, pages 777–831. Elsevier.
- Turner, R. 1997. Types. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, pages 535–586. Elsevier.
- Urquhart, A. 1995. The complexity of propositional proofs. *Bulletin of Symbolic Logic* 1(4):425–467.
- Van Benthem, J. 1991. *The Logic of Time*. Kluwer Academic Publishers, 2nd edn.
- Van Benthem, J. and A. Ter Meulen, eds. 1997. *Handbook of Logic and Language*. Elsevier Science.
- Van Deemter, K. 2004. Generating Referring Expressions: Boolean Extensions of the Incremental Algorithm. *Computational Linguistics* 28(1):37–52.
- Van Deemter, K. and S. Peters, eds. 1996. *Semantic Ambiguity and Under-specification*. CSLI Lecture Notes 55. Stanford: Chicago University Press.

- Van Eijck, D.J.N. and J. Jaspars. 1996. Ambiguity and Reasoning. Tech. Rep. Report ISSN 0169-118X, CSR9616, Centrum voor Wiskunde en Informatica (CWI).
- Van Eijck, J. and H. Kamp. 1997. Representing discourse in context. In J. Van Benthem and A. Ter Meulen, eds., *Handbook of Logic and Language*, pages 179–237. Elsevier.
- Vardi, M.Y. 1982. The complexity of relational query languages. In *Annual ACM Symposium on Theory of Computing Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146. New York: ACM Press.
- Vestre, E.J. 1991. An Algorithm for Generating Non-Redundant Quantifier Scopings. In *Fifth Conference of the European Chapter of the ACL*, pages 251–256. Berlin, Germany.
- Vossen, P. 1998. *EuroWordNet: a multilingual database with lexical semantic networks*. Kluwer Academic Publishers.
- Wall, L., T. Christiansen, and J. Orwant. 2000. *Programming Perl*. O'Reilly.
- Wassermann, R. 1999. *Resource-bounded Belief Revision*. Ph.D. thesis, University of Amsterdam.
- Winston, P. 1981. *Artificial Intelligence*. Addison-Wesley, 2nd edn.
- Wos, L. and G.W. Pieper. 2000. *A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning*. World Scientific.
- Zhang, H. and J. Zhang. 1996. Generating Models by SEM. In M. A. McRobbie and J. K. Slaney, eds., *Proceedings of the International Conference on Automated Deduction (CADE'96)*, pages 308–312. Springer-Verlag.

Author Index

A

- Abelson, H., 97
Abiteboul, S., 51
Alshawi, H., 151
Althaus, E., 152
Andrews, P., 305
Appelt, D., 301
Apt, K., 253
Asher, N., 303

B

- Baader, F., xxiii, 53, 302
Baker, C.F., 302
Barendregt, H., 102
Barwise, J., 53
Baumgartner, P., 301, 315
Bell, J., 199, 252
Benson, D.B., xxiv
Benzmüller, C., 305
Bishop, M., 305
Black, A.W., vii
Blackburn, P., xix, xx, 52, 53,
 103, 301, 310
Bodirsky, M., 152
Boole, G., 6
Boolos, G., 52, 257
Bos, J., xix, xx, xxv, 103, 128,
 151, 153, 301, 304, 310
Bunt, H., xxv
Burton-Roberts, N., 94
Buschbeck-Wolf, B., 151

C

- Calvanese, D., xxiii, 53, 302
Carnap, R., xxi
Carpenter, B., 96
Chang, C.-L., 254
Charniak, E., xxiii, xxiv
Chaves, R., 149
Cherniak, C., 52
Chierchia, G., 102
Christiansen, T., 256
Church, A., 97, 98
Cimiano, P., 301
Claessen, K., 256, 317
Clark, S., 304
Clocksin, W., 254
Cook, S., 201
Cooper, R., vii, 53, 113, 150, 306
Copestake, A., 151
Cronin, B., 302
Crouch, R., 306
Curran, J.R., 304

D

- D'Agostino, M., 200, 253
Dale, R., 304
Dalrymple, M., 103
Davis, M., 200
De Nivelle, H., 103, 255, 315
De Rijke, M., xxv, 52
Debusmann, R., 53
Doets, K., 53, 102
Dorna, M., 151

- Dowek, G., 305
 Dowty, D.R., 96, 149, 302
 Duchier, D., 152
- E**
 Egg, M., 151, 153
 Enderton, H., 50, 52–54
 Erk, K., 152
- F**
 Fellbaum, C., 302
 Fermüller, C., 256
 Fillmore, C.J., 302
 Fitting, M., vii, 102, 199, 219,
 225, 252–254
 Flickinger, D., 151
 Fodor, J.A., 302
 Fox, C., 306
 Franconi, E., 53
 Frege, G., xx, 94
 Fu, Z., 316
- G**
 Gabbay, D., 253
 Gabsdil, M., 53, 149, 304
 Gärdenfors, P., 303
 Gaizauskas, R., 304
 Gallier, J., 199
 Gallin, D., 102
 Gambäck, B., 151
 Gamut, L.T.F., 50, 96
 Gardent, C., 53, 301, 304, 306
 Garey, M., 200
 Gaylard, H., xxv, 304
 Gazdar, G., 96, 102
 Ginzburg, J., 303
 Grice, H.P., 52
 Groenendijk, J., 54, 303
 Grosz, B.J., xxiv
 Gu, J., 200
 Gunji, T., xxiv
- H**
 Haarslev, V., 303
 Hähnle, R., 253
 Haken, A., 200
- Halpern, J.Y., 51
 Hamblin, C.L., 303
 Harabagiu, S., 304
 Harman, G., 52
 Heim, I., xxii, 54
 Henkin, L., 53, 305
 Higginbotham, J., 303
 Hindley, J., 102
 Hirschman, L., 304
 Hobbs, J.R., xxiv, 150, 301
 Hockenmaier, J., 304
 Hodges, W., 50
 Hodgson, K., 316
 Hongwi, X., 305
 Horrocks, I., 303
 Huet, G., 305
 Hull, R., 51
 Hustadt, U., 256
 Huth, M., 51
- I**
 Issar, S., 305
- J**
 Jackendoff, R., 302
 Jacquey, E., 53
 Jakobson, R., 302
 Janssen, T.M.V., 94, 96
 Jaspars, J., 52, 151, 152, 306
 Jeffrey, R., 52, 257
 Johnson, D., 200
 Johnson, M., vii
 Johnson, R., xxiv
 Jurafsky, D., xxv, 96, 302
- K**
 Kalman, J.A., 255, 316
 Kamp, H., xxii, 53, 54, 151, 304,
 306
 Karttunen, L., 303
 Katz, J.J., 302
 Kay, M., vii
 Keenan, E.L., 53
 Keller, W.R., 124, 150
 Kingsbury, P., 304
 Klein, E., 301

- Knight, K., xxiii
 König, E., 153
 Kohlhase, M., xxv, 103, 301
 Koller, A., 53, 151–153, 301
 Konrad, K., 301, 306
 Kühn, M., 301

L

- Lamping, J., 103
 Landsbergen, J., xxiv
 Langholm, T., 53
 Lappin, S., xxv, 303
 Lascarides, A., 303
 Lee, R.C.-T., 254
 Leitsch, A., 199, 254, 256
 Letz, R., 253
 Levesque, H., 200
 Levin, B., 302
 Lewin, I., vii, 150
 Lieske, C., 151
 Logemann, G., 200
 Loveland, D., 200
 Lucchesi, C., 305
 Lønning, J.T., 53

M

- Machover, M., 199, 252
 Main, M.G., xxiv
 Malouf, R., 151
 Marcus, M., 304
 Martin, J., xxv, 96, 302
 Martin, P., 301
 Martinich, A., xx
 McCune, W., 255, 256, 316, 317
 McDermott, D., xxiii
 McGuinness, D., xxiii, 53, 302
 Mehlhorn, K., 152
 Mellish, C., 96, 102, 254
 Miele, S., 152
 Miller, G., 302
 Milward, D., 84, 306
 Mitchell, D., 200, 201
 Möller, R., 303
 Moldovan, D., 304
 Montague, R., xii–xiv, xxi, 94,
 96, 102, 109, 149

- Monz, C., xxv
 Moore, R.C., xxv, 103
 Mori, Y., 151
 Muskens, R., xxv, 53, 54, 102

N

- Nardi, D., xxiii, 53, 302
 Nerbonne, J., xxv, 151
 Nesmith, D., 305
 Niehren, J., 151, 152
 Nielson, F., 98
 Nielson, H., 98
 Nilsson, N., xxiii
 Norvig, P., xxiii

O

- Oka, T., 301
 Orwant, J., 256

P

- Pagin, P., 94
 Palmer, M., 304
 Papadimitriou, C.H., 51, 200
 Partee, B.H., xxii, 96, 102
 Pasca, M., 304
 Patel-Schneider, P., xxiii, 53, 302
 Pelletier, F.J., xxiv, 150, 254,
 255, 303
 Pereira, F.C.N., vii, 96, 103, 304
 Peters, S., 96, 149, 153
 Pfenning, F., 305
 Phoenix, T., 256
 Pieper, G.W., 255, 316
 Pinkal, M., 151, 153, 306
 Player, N.J., 151, 153
 Poesio, M., 306
 Posegga, J., 253
 Pratt-Hartmann, I., 53, 307
 Pulman, S., 306
 Pustejovsky, J., 153, 302
 Putnam, H., 200

R

- Radford, A., 94
 Ramsay, A., xxv, 301, 304
 Ranta, A., 201

- Reiter, E., 304
 Reyle, U., xxii, 54, 151, 153, 304
 Riazanov, A., 316
 Rich, E., xxiii
 Riehemann, S., 151
 Robinson, A., 255
 Robinson, J.A., xiii, 237, 253
 Rosenschein, S., xxiv
 Rosner, M., xxiv
 Ruhrberg, P., 151
 Rupp, C.J., 151
 Russell, B., xxi, 19
 Russell, S., xxiii
 Ryan, M., 51
- S**
 Sag, I., 151, 303
 Saraswat, V.A., 103
 Schmitt, P., 253
 Schubert, L.K., xxiv, 150, 254
 Schwartz, R.L., 256
 Scott, D., 98
 Seldin, J., 102
 Selman, B., 200
 Seville, H., 301
 Shan, C., 304
 Shapiro, E., 253
 Shapiro, S., 53
 Schieber, S.M., vii, 96, 150, 304
 Slaney, J., 316
 Smallyan, R., 199, 252, 254
 Sörensson, N., 256, 317
 Sparck Jones, K., xxiv
 Steedman, M., 304
 Sterling, L., 253
 Stickel, M.E., 301
 Stokhof, M., 54, 303
 Stone, M., 304
 Stoy, J., 98
 Striegnitz, K., xix, 53, 149, 304,
 310
 Sundholm, G., 199, 201, 252
 Surdeanu, M., 304
 Sussman, G., 97
- T**
 Tammet, T., 256, 315
 Tarski, A., xxi
 Ten Cate, B., 304
 Ter Meulen, A., 303
 Thater, S., 152
 Thiel, S., 152
 Thijssse, E., xxv
 Thomason, R.T., 303
 Turner, R., 102
- U**
 Urquhart, A., 200
- V**
 Van Benthem, J., 53, 54, 102, 303
 Van Deemter, K., 153, 304
 Van Eijck, J., 54, 151, 152, 306
 Van Genabith, J., 306
 Vardi, M.Y., 51
 Venema, Y., 52
 Vestre, E.J., 149
 Vianu, V., 51
 Visser, A., 54
 Voronkov, A., 255, 316
 Vossen, P., 302
- W**
 Wall, L., 256
 Wall, R.E., 96, 149
 Wassermann, R., 52
 Webber, B.L., xxiv, 301, 306
 Westerståhl, D., 53, 94
 Wilks, Y., xxiv
 Winston, P., xxiii
 Worm, K., 151
 Wos, L., 255, 316
- X**
 Xu, F., 151
- Z**
 Zhang, H., 256
 Zhang, J., 256
 Zhang, L., 316

Prolog Index

Symbols

`!`, 172
`=..`, 37
`==`, 37, 80, 83
`\+ predicate`, 38

A

`admissiblePlugging/1` predicate, 146
`all/2` term, 32
`alphabeticVariants/2` predicate, 83
`alphaConvert/2` predicate, 80, 82
`alphaConvert/4` predicate, 80, 81, 83
`alphaConvertList/4` predicate, 81
`and/2` term, 32
`answerQuestion/3` predicate, 296
`app/2` term, 73–75, 89
`appendLists/3` predicate, 221–223
`arg/3` predicate, 60, 61
`assert/1` predicate, 261
`assertUSR/2` predicate, 144

B

`babyCurt.pl` file, 259
`basicFormula/1` predicate, 221
`betaConversion.pl` file, 84
`betaConversionTestSuite.pl` file, 84

`betaConvert/2` predicate, 78
`betaConvert/3` predicate, 77, 78, 82
`betaConvertList/2` predicate, 78
`betaConvertTestSuite/0` predicate, 82
`bimp/2` term, 173, 188
`bo/2` term, 117, 126

C

`callInference.pl` file, 241, 242, 248
`callInference.pl` file, 268
`callMB/4` predicate, 250
`callTP/3` predicate, 240
`callTPandMB/6` predicate, 250
`checkAnswer/2` predicate, 297
`cleverCurt.pl` file, 269
`closedBranch/1` predicate, 172, 221
`closedTableau/1` predicate, 170
`closedTableau/2` predicate, 221
`cnf/2` predicate, 185, 231
`cnfTestSuite/0` predicate, 187
`combine/2` predicate, 88–90, 92, 118, 119, 126, 143
`compose/3` predicate, 37, 78, 81, 92
`comsemPredicates.pl` file, 36, 37
`conjunctive/3` predicate, 172, 223

conjunctiveExpansion/2

predicate, 171, 222

consistent/2 predicate, 266**consistent/3 predicate**, 271**consistentReadings/2**

predicate, 266

consistentReadings/3

predicate, 270

cooperStorage.pl file, 119**cooperStorage/0 predicate**, 120**curtOutput/1 predicate**, 263**curtPredicates.pl file**, 261, 264**curtTalk/1 predicate**, 261**curtUpdate/3 predicate**, 262,
263, 265, 271, 274, 278**D****disjunctive/3 predicate**, 172,
223**disjunctiveExpansion/2**
predicate, 222**disjunctiveExpansion/3**
predicate, 171**dom/2 predicate**, 145**E****elimEquivReadings/2 predicate**,
279**englishGrammar.pl file**, 87, 121,
127, 149**englishLexicon.pl file**, 87, 90,
121, 127, 149**eq/2 term**, 32**evaluate/3 predicate**, 39, 40**example/2 predicate**, 39**exampleModels.pl file**, 39**existential/1 predicate**, 223**existentialExpansion/2**
predicate, 223**expand/2 predicate**, 171**expand/4 predicate**, 221**experiment1.pl file**, 58, 63**experiment2.pl file**, 58**experiment3.pl file**, 58**expression/2 predicate**, 82, 83**F****f/1 term**, 169**fail/0 predicate**, 40**filterAlphabeticVariants/2**
predicate, 120**findall/3 predicate**, 296**fol2bliksem/2 predicate**, 239**fol2mace/2 predicate**, 248**fol2otter/2 predicate**, 239**fol2tptp/2 predicate**, 248**folTestSuite.pl file**, 224**foResolution.pl file**, 234**formula/2 predicate**, 172**freeVarTabl.pl file**, 224, 225**G****g/2 term**, 33**H****helpfulCurt.pl file**, 299**history/1 predicate**, 261**holeSemantics.pl file**, 143**holeSemantics/0 predicate**, 148**I****i/4 predicate**, 37**imp/2 term**, 32**inferenceEngines.pl file**, 239**inferenceEngines/1 predicate**,
240**infix/0 predicate**, 310**informativeReadings/2**
predicate, 274, 275**instance/3 predicate**, 220**interfaceMB.perl file**, 250**interfaceTP.perl file**, 241**interfaceTPandMB.perl file**, 251**K****kellerStorage.pl file**, 126, 259**knowledgeableCurt.pl file**, 283**L****lam/2 term**, 73**lexEntry/2 predicate**, 90, 91**lexicalKnowledge/3 predicate**,
288

list2string/2 predicate, 298,
 299
literal/1 predicate, 186

M

mb.out file, 250
mbTestSuite/0 predicate, 252
member/2 predicate, 36
memberList/2 predicate, 36–38,
 223
menu/0 predicate, 310
model/2 term, 30
modelChecker1.pl file, 40, 43
modelChecker2.pl file, 43
modelCheckerTestSuite.pl file,
 39
modelCheckerTestSuite/0
 predicate, 39

N

nnf/2 predicate, 185, 186, 232
nnf2cnf/3 predicate, 186
nonRedFact/2 predicate, 234
nonRedundantFactors/2
 predicate, 233, 234
not/1 term, 32
notatedFormula/3 predicate, 220
numberReadings/4 predicate, 279
numbervars/3 predicate, 83, 144

O

or/2 term, 32

P

parent/2 predicate, 145
plugHoles/3 predicate, 146
plugUSR/2 predicate, 147
prefix/0 predicate, 310
propResolution.pl file, 191
propTableau.pl file, 173
propTestSuite.pl file, 172, 191

Q

Qdepth variable, 253
que/3 term, 294

R

readings/1 predicate, 261
readLine/1 predicate, 261
realiseAnswer/4 predicate, 299
realiseString/4 predicate, 298
refute/2 predicate, 189
removeClosedBranches/2
 predicate, 172
removeDuplicates/2 predicate,
 187
removeFirst/3 predicate, 171,
 223
resolve/3 predicate, 190, 233
resolveClauseList/4 predicate,
 190
resolveList/3 predicate, 189
retract/1 predicate, 261
rprove/1 predicate, 188, 233
rproveTestSuite/0 predicate,
 191, 234
rugratCurt.pl file, 265

S

satisfy/4 predicate, 33–38
scrupulousCurt.pl file, 278
selectFromList/3 predicate, 191
semLex/2 predicate, 92, 117, 142,
 143
semLexHole.pl file, 142, 149
semLexLambda.pl file, 87, 93
semLexStorage.pl file, 117, 121,
 126, 127
semRulesCooper.pl file, 118, 121
semRulesHole.pl file, 143, 149
semRulesKeller.pl file, 127
semRulesLambda.pl file, 87–89
sensitiveCurt.pl file, 273
setCnf/2 predicate, 187
setof/3 predicate, 36
skolemFunction/2 predicate,
 220, 222, 232, 254
skolemise/3 predicate, 232
some/2 term, 33
sRetrieval/2 predicate, 119, 126
substitute/4 predicate, 220, 232

T

t/1 term, 169
test/4 predicate, 39
top/1 predicate, 146
tp.out file, 241
tpmb.out file, 251
tpmbTestSuite/0 predicate, 252
tprove/1 predicate, 170
tprove/2 predicate, 224
tproveTestSuite/0 predicate,
 173, 224
tpTestSuite/0 predicate, 242,
 252

U

unary/2 predicate, 172, 223
unaryExpansion/2 predicate,
 171, 222
unify_with_occurs_check/2
 predicate, 215, 221, 253
unionSets/3 predicate, 189
universal/1 predicate, 223
universalExpansion/4
 predicate, 223
url2srl/2 predicate, 147

Subject Index

Symbols

\exists , 6, 7
 \forall , 6, 7
 λ , 66
 \leftrightarrow , 15
 \neg , 6, 7
 \perp , 15
 \rightarrow , 6, 7
 T , 15
 \vee , 6, 7
 \wedge , 6, 7

A

a , 61, 63, 94
abduction, 301
ABox, 303
abstraction, 66, 67, 98
accidental binding, 68–70, 79
adjective, 90
admissible plugging, 134, 136, 137
AI, xxii, 300
 all , 8, 11, 94
 α -conversion, 68, 80, 82
Prolog implementation of, 78–81
 α -equivalence, 68, 83
alphabetic variants, 68
filtering out, 120
antecedent, 10
application, 67, 70
argument, 67, 73

arity, 2
Artificial Intelligence, *see* AI
assignment, 12
assignment function, 12
associative laws, 176, 177, 186
associative operator, 27
atomic MGU closure rule, 218
automated reasoning, 45, 156, 315
auxiliary verb, 91

B

background knowledge, 45, 282, 285
belief revision, 303
 β -conversion, 66–68, 70, 71, 80, 82
Prolog implementation of, 75–78
 β -reduction, 67
bi-implication, 15, 173
expansion rules for, 166
binary resolution rule, 180
binding operator, 113
Bliksem, 238–242, 251, 252, 255, 268, 309, 315
formula syntax, 239
boolean connectives, 6
precedence conventions, 10
 both , 94
bound variable, 8, 9, 11
inductive definition of, 8

renaming, 80
 branching expansion rule, 162

C

CADE, 248
 characteristic function, 100
 Church's Thesis, 243, 256
 classical logic, 44, 47
 clause, 175
 clause normal form, 225
 clause set, 179, 181
 closed class word, 285
 closed tableau, 160
 closed term, 16
 CNF, *see* conjunctive normal form
 co-NP complete, 198
 commutative operator, 27
 completeness, 193–195

- of first-order resolution, 225
- of first-order tableau, 208
- of propositional resolution, 193
- of propositional tableau, 193

 complexity

- of proof methods, 200
- of propositional validity, 197
- of querying task, 51

 compositional semantics, 57
 compositionality, 55–59, 94
 compound name, 91
 computational linguistics, xv,

- xxii, 304

 computational semantics, xi
 conjunct, 10
 conjunction, 6, 7

- Prolog representation of, 32

 conjunctive expansion rule, 164,

- 165

 conjunctive normal form,

- 175–179, 228
- Prolog implementation of, 185

 consequent, 10
 consistency, 21

- relation to informativity, 28
- with respect to other formulas, 29

consistency checking task, 21, 23,

- 155, 156, 196, 208, 243, 244,
- 247, 250, 265, 269

 proof-theoretic perspective on, 23
 undecidability of, 23, 52

constant, 4, 6
 Cooper storage, 113–122

- overgeneration, 124

 Prolog implementation of, 116–121

coordination

- noun phrase, 88
- semantic rule for, 90

 copula, 93
Curt

- Baby Curt, 259
- background knowledge, 284
- Clever Curt, 267
- consistency checking, 268
- Helpful Curt, 52, 248, 293
- informativity checking, 272
- Knowledgeable Curt, 281
- questions, 293
- reserved commands, 260
- Rugrat Curt, 264
- Scrupulous Curt, 277
- Sensitive Curt, 272

 Cyc, 302

D

DCG, xvi, xvii, 57–62, 72, 74, 75,

- 84–88
- left-recursive rules, 88

 De Morgan laws, 28, 177, 185
 decidability, 194
 default logic, 303
 Definite Clause Grammar, *see* DCG
 description logic, xxiii, 46, 52,

- 302

 determiner, 58, 65, 91

- lambda expression for, 74
- semantic macro for, 92

 disagreement pair, 212

- Discourse Representation
 Structures, 86, 103, 304
- Discourse Representation Theory,
see DRT
- disjunct, 10
- disjunction, 6, 7
 Prolog representation of, 32
- disjunctive expansion rule, 162,
 164, 165
- distributive laws, 28, 176, 177,
 186, 228
- ditransitive verb, 84, 264
- dominance constraint, 132
- DPLL procedure, 200
- DRT, xiv, xxii, 53, 86, 103, 304
- Dynamic Predicate Logic, 54
- dynamic Prolog predicate, 261
- E**
- either... or, 11
- entity, 5
 anonymous, 5
- epistemic states, 47
- equality, 17–19, 236, 237
 Prolog representation of, 32
- equality axioms, 293
- EuroWordNet, 302
- events, 47, 48
- every, 8, 50, 61, 69
- everybody, 8, 11
- everyone, 11
- everything, 8, 264
- existential quantifier, 6
 Prolog representation of, 32
- existential rules, 206, 216, 217
- expansion rules, *see* tableau
 expansion rules
- F**
- Fact, 303
- FDPLL, 315
- few, 50
- File Change Semantics, 54
- Finder, 316
- finite model, 246
- first-order language, 3, 6–11, 48
 syntax of, 6, 9
 with equality, 17
- first-order logic, 1–19, 44, 48
 and natural language, 44–50
 inference tools, 45
 inferential capabilities, 44–47
 introduction to, 50
 representational capabilities,
 47–50
- two-variable fragment of, 53
- first-order model, *see* model
- first-order resolution, 225–231,
 254
 implementation of, 231–234
- first-order syntax
 readability of, 9
- first-order term, 6
- first-order validity, 207
- first-order variable, *see* variable
- formal semantics, xi
- formula, 2
 atomic, 6
 basic, 6
 informative, 25
 invalid, 24, 25
 matrix of, 7
 Prolog representation of, 32
 quantified, 7
 uninformative, 25
 valid, 25
 well formed, 7
- formula tree, 129
- free variable, 8, 9, 11, 12, 14, 79
 analogy with pronouns, 12
 inductive definition of, 8
- free-variable tableau, 215–219,
 253
 implementation of, 219–224
- free-variable tableau prover, 265,
 266
- function symbols, 16–17
- functional application, 67, 72, 99
- functor, 67, 73
- G**
- Gandalf, 315, 316

generalized quantifier theory, 53
 generalized quantifiers, 50
 generation, 304
 grammar, 86
 shortcomings of, 89
 grammar architecture, 86
 grammar engineering, 86–93
 basic principles of, 86

H

higher-order logic, 49, 101, 305
 hole semantics, 127–149
 implementation, 137–148
 semantic macros for, 142
 semantic rules for, 142
 hybrid logic, 46, 52
 hyperresolution, 256

I

idempotency, 214
 idempotent MGUs, 214
 iff, 13, 15
 implication, 6, 7
 Prolog representation of, 32
 inconsistency, 21
 with respect to other formulas, 29, 156, 196
 inconsistent description, 21
 inconsistent formula, 29
 indexed binding operator, 113
 inference, xii, 19, 155
 in natural language semantics, 47
 inference tasks, 19
 infix notation, 33, 74, 260
 informative formula, 26
 informativity, 25, 26
 in communication, 52
 relation to consistency, 28
 with respect to other formulas, 26
 informativity checking task, 24–26, 155, 156, 196, 208, 243, 245, 247, 250, 272
 proof-theoretic perspective on, 27

undecidability of, 27, 52
 inheritance, 286
 interpretation function, 4, 5
 intervals, 47
 intransitive verb, 58, 69
 invalid argument, 24, 25, 276
 invalid formula, 24, 25

K

Keller storage, 122–127
 Prolog implementation of, 125–126
 knowledge representation, 286, 287

L

labelled formula, 132
 lambda abstraction, 66
 lambda calculus, 66–72
 as glue language, 69
 implementation of, 73–84
 typed, 97
 untyped, 97
 lambda expression, 67, 69, 70
 model-theoretic interpretation, 97
 Prolog representation of, 73
 λ-conversion, 67
 LeanTAP, 253
 Learn Prolog Now, xix, 310
 lexical entry, 61, 86
 lexical item, 57, 72
 lexical knowledge, 282, 284, 301, 303
 lexical rule, 88
 lexical semantics, 86, 285, 301
 lexicon, 57, 86, 90–92
 literal, 175

M

Mace, 247–251, 255, 256, 268, 309, 316
 many, 50
 matrix, 204
 MGU, 211, 214, 219
 minimal model, 270

modal logic, 46, 51–53
 modal phenomena, 47
 model, 2–5, 9, 13, 47, 48
 domain of, 4
 entity in, 5
 finite, 21
 infinite, 21
 Prolog representation of, 30
 model builder, 242–251, 268, 270, 301
 model building, 153, 246, 247, 255
 approaches to, 256
 by iteration, 248
 SAT solving, 201
 vs. model checking, 245
 model checker, 21, 29–43, 295
 handling bound variables, 40
 Prolog implementation of, 29
 weak points of, 42
 model checking, 51
 vs. model building, 245
 model representation, 249
 model theory, 23, 25, 44
 Montague semantics, xii, 96
 Montague's rule of quantification, 109
m
 most, 50, 53
 most general unifier, *see* MGU

N

natural language argument, 275
 natural language generation, 304
 natural language syntax, 57, 59
 negation, 6, 7
 Prolog representation of, 32
 negation normal form, 176–179, 226
 Prolog implementation of, 185
 nested stores, 124
 NNF, *see* negation normal form
 no, 264
 nobody, 264
 non-logical symbol
 Prolog representation of, 32
 non-monotonic logic, 303

non-redundant factor, 230, 231
 notation, 319
 noun, 58, 69, 90
 semantic macro for, 92
 noun phrase, 56, 58
 NP-complete, 51, 198, 200
 nuclear scope, 63, 65

O

occurs check, 209
 ontology, 285, 286
 open class word, 285
 OpenCyc, 302
 Otter, 238–242, 248, 251, 252, 255, 309, 316
 formula syntax, 239

P

Paradox, 247–251, 256, 309, 317
 parallel inference, 242, 250
 parameter, 205, 216
 parser, 59
 parsing, 58
 partiality, 49
 Perl, 241, 242, 250, 251, 256, 309
 pigeon hole principle, 198
 plugging, 134–137
 plugging algorithm, 143–147
 plural noun phrases, 53
 plural nouns, 94
 polynomial time, 199, 307
 possible world semantics, 48
 pragmatics, xiv, xxi, xxii, xxiv, 52, 96, 103, 107–109
 prefix notation, 33, 74
 preposition, 94
 Prolog
 introduction to, 310
 Prolog cut, 172, 188
 Prolog interpreter, 309
 Prolog unification, 209
 pronoun, 9
 proof method, 156, 157
 proof theory, 45, 156, 199
 and natural language semantics, 201

proper name, 58, 90
 lambda expression for, 71
 semantic macro for, 92

property, 2

proposition symbol, 312

propositional inference, 155

propositional language, 158

propositional logic, 10, 158, 312

propositional model, 164

propositional resolution, 174–199
 completeness, 193
 Prolog implementation of, 185–191
 soundness, 193

propositional tableau, 158–168
 completeness, 193
 Prolog implementation of, 168–174
 Prolog representation of, 169
 rule for negation, 164
 rules for binary connectives, 165
 soundness, 192

propositional validity
 complexity of, 197
 decidability of, 194

PSPACE-complete, 51

Q

Q-depth, 220, 224, 225, 240

quantified noun phrase, 61, 65

quantified sentence, 12
 matrix of, 12

quantifier, 6, 50
 counting, 50
 generalized, 50
 Prolog representation of, 32

quantifier raising, 109

quantifier scope ambiguity, 105
 complex noun phrases, 122

quantifier storage, 112

quantifier-free formula, 311

quantifier-free fragment, 10, 311

quasi-logical form, 293, 294

querying task, 20
 complexity of, 51

questions, 294, 303

R

Racer, 303

recursively enumerable, 257

refutation proof method, 159, 181

relation, 2
 binary, 2
 unary, 2

relation symbol, 4

resolution, *see* propositional resolution, *see* first-order resolution
 complementary clauses, 180
 complementary pair, 180
 resolvents, 180

resolution rule, 179–184, 229

restriction, 63, 65

retrieval rule
 Cooper, 116
 Keller, 124

robot, 301

rule-saturated tableau, *see* tableau

S

SAT solver, 200, 201

SAT solving, 200

Satchmo, 317

satisfaction, 12

satisfaction definition, xxi, 11–16
 Prolog implementation of, 33

satisfiability, 22

satisfiable formula, 22

Schubert's Steamroller, 236, 242, 254

scope ambiguities
 in arguments, 276

scope ambiguity, 105, 107, 260

Scott, 316

second-order entity, 48

second-order logic, 49, 53
 first-order interpretation of, 49

second-order quantification, 48, 49

Sem, 256

- Semantic Deduction Theorem, 25–27, 156, 196
 semantic lexicon, 86, 92–93
 semantic macro, 89, 92, 94
 semantic rules, 86, 89–90
 semantic tableau, 157
 semantic underspecification, 128
 Semantic Web, 302
 sentence
 definition of truth, 14
 logical equivalence, 27
 of first-order logic, 9
 truth for, 12
 sentence symbol, 158, 312
 set CNF, 178, 179, 181
 Sicstus Prolog, 309
 signature, *see* vocabulary
 signed formula, 159, 166, 204
 signs, 159
 simultaneous unification
 problem, 214
 situational knowledge, 282, 284, 289, 303
 Skolem constant, 217
 Skolem function, 226
 Skolem function symbol, 217
 Skolem term, 217, 226
 skolemisation, 144, 226, 227
 some, 8, 11, 50
 somebody, 8
 someone, 8, 10
 something, 8
 sorted first-order logic, 18–19, 131
 sorted variables, 18
 soundness, 191, 194, 195
 of first-order resolution, 225
 of first-order tableau, 207
 of propositional resolution, 193
 of propositional tableau, 192
 Spass, 316
 spoken dialogue, 301
 SRL, 129
 SRL formula tree, 129
 stack, 75
 use in β -conversion, 75
 storage, 112
 limitations, 127
 nested stores, 124
 storage rule
 Cooper, 114
 Keller, 124
 store, 113
 Prolog representation for, 117, 126
 subformula, 8, 10
 subformulahood
 inductive definition of, 8, 11
 substitution, 209
 subsume, 234
 SWI Prolog, 309
 symbol
 logical, 17
 non-logical, 6
 synset, 302
 syntactic analysis, 57
 syntactic category, 69
 syntactic structure, 56, 57
 syntax rules, 86–89
 syntax-semantics interface, 87
- T**
- tableau, *see* propositional tableau
 tableau, free-variable tableau
 closed, 160
 first-order tableau, 204–209
 initial tableau, 166, 167
 rotation of branches, 221
 rule-saturated, 160, 166
 signed, 252
 unexpanded nodes, 166
 unsigned, 252
 tableau branch, 162, 166
 tableau expansion rules, 159, 164, 204
 tableau proof, 158, 161, 167
 tableau systems, 199
 TBox, 303
 temporal logic, 51, 53
 temporal semantics, 48
 term

closed, 16
 interpretation of, 17
 test suite, 39, 82, 310
 for β -conversion, 84
 for CNF, 187
 for grammar, 93
 for model checker, 39, 40
 for propositional resolution, 191
 for propositional tableau, 172
 pigeon hole principle, 198
 with first-order problems, 224,
 242
the, 19, 94, 264
theorem prover, 28, 156, 268, 270
 off-the-shelf, 235–242
three, 50
time, 47
TPTP syntax, 248, 316
transitive verb, 56, 58, 91
 lambda expression for, 72
truth, 11, 12
truth definition, 12
truth table, 158, 159, 168, 174,
 196, 197, 200, 311, 312
two, 50
two-variable fragment, 53
typed lambda calculus, 98

U

unary expansion rule, 164
undecidability, 207
 of consistency checking task, 23
 of first-order logic, 197, 241,
 243, 256
 of informativity checking, 27
underspecification, 128
underspecified representation
 language, 129
underspecified semantic
 representation, 129
unifiable, 211
unification, 209–215
 algorithm, 211, 213
 definition, 211
 occurs check, 209
 substitution, 209

unifier, 211
uninformative formula, 29
uninformativity, 25, 26
 with respect to other formulas,
 25, 156, 196
universal quantifier, 6
 Prolog representation of, 32
universal rules, 205, 208, 216
unsatisfiability, 22
untyped lambda calculus, 97
URL, 129
USR, 129

V

valid argument, 24, 25, 276
valid formula, 24, 25, 29
valid inference, 24
validity, 24, 25, 156, 157
 and truth tables, 312
validity checking, 156
Vampire, 316
variable, 6
 Prolog representation of, 32
 sorted, 18
variable assignment, 12
variant assignment, 13
verb phrase, 56, 58
vocabulary, 2–4, 6, 13, 48
 choice of, 6

W

weak vs. strong readings, 107–109
well formed formula, 7
wff, 7
wh-question, 293
 representation of, 294
wide vs. narrow scope, 106
WordNet, 302
world knowledge, 282, 284, 288,
 303

Z

zChaff, 316

Can meaning be represented in ways that computers can make use of? Can computers distinguish coherent from incoherent utterances, and judge whether a sentence contains new information? This textbook—the first wholly devoted to *computational semantics*—takes such questions as its starting point. It provides a thorough introduction to the underlying theoretical issues, and full implementations of the techniques it describes. Clearly written, it is a point of entry to contemporary computational semantics that will be accessible to computer scientists, linguists, and logicians—and indeed, to anyone curious about the role of meaning in human languages.

Patrick Blackburn is a Directeur de Recherche (Director of Research) at INRIA, France's national organization for research in computer science. **Johan Bos** is a senior researcher at the School of Informatics of the University of Edinburgh.

"This book is the established textbook on the logic programming approach to natural language interpretation. It provides an exciting combination of standard Montague techniques, modern approaches to underspecification and the use of first order theorem provers, all in a book that can be used by advanced undergraduates or graduate students."

Robin Cooper
Professor of Computational Linguistics
Göteborg University

"A clear and elegant introduction to computational semantics, of equal value for its presentation of basic theory and its demonstration of good language engineering practice."

Bonnie Webber
Professor of Intelligent Systems
University of Edinburgh

