Computational Semantics for Natural Language

Course Notes for NASSLLI 2003
Indiana University, June 17–21, 2003

Patrick Blackburn & Johan Bos

May 28, 2003

Contents

Preface

These are the course notes for "Computational Semantics for Natural Language", a five lecture course given by Patrick Blackburn and Johan Bos at NASSLLI 2003, the second North American Summer School in Logic, Language and Information, June 17-21, 2003. As you can see, the notes have four chapters. Each is taken from the book "Representation and Inference for Natural Language: A First Course in Computational Semantics" by Patrick Blackburn and Johan Bos, which will soon be published by CSLI Press. The chapters of the course notes are draft versions of four of the six chapters of our book. We have omitted from these notes the chapters on propositional and first-order inference methods, the bibliographical notes and bibliography, and the appendices.

The goal of the NASSLLI course (and indeed, the book) is to introduce the basics of natural language semantics—but to do so from a computational perspective, and to do so in a way that emphasizes the role of inference. Traditional introductions to formal semantics tend to focus on how to build semantic representations for natural language expressions. Now, we also devote a lot attention to this (important) topic: we discuss first-order logic, lambda calculus, and methods for coping with scope ambiguities in detail. But we present this material in a non-traditional way. In particular, we always take care to raise (and answer) the question "How do we make computational sense of these ideas?" Furthermore, we don't just discuss how representations are built: we also discuss (computational) methods for performing inference, and draw the themes of representation and inference together. In particular, in the last chapter of these notes we take the reader on a guided tour of the CURT family. This is a series of simple dialogue systems which illustrate how tools for building semantic representations can be combined with inference tools, and why this might be an interesting thing to do.

Further material of relevance to the course can be found at our website at *www.comsem.org*. There you will find all the Prolog source code for the programs discussed in these notes. You will also find a pointer to the book "Learn Prolog Now!", by Patrick Blackburn, Johan Bos, and Kristina Striegnitz. Now, we believe you can get a lot out of this course even if you don't know Prolog: the course attempts to make a number of general concepts and approaches clear, and these are independent of Prolog (and indeed, independent of any programming language). Nonetheless, if you do know Prolog, you will almost certainly gain a deeper understanding of the computational issues involved (as well as being able to have some fun in the practical sessions). So we recommend that anyone planning on taking this course should brush up on rusty Prolog skills—or indeed, consider acquiring them from scratch. "Learn Prolog Now!" was written with the needs of beginners (indeed, beginners learning Prolog by self study) in mind. If Prolog is new to you, it may be a good way to start getting to know it.

Well—that's enough introduction. It's time to get on with some computational semantics for natural language. Enjoy the course!

Patrick Blackburn[1]
INRIA Lorraine
France
patrick@aplog.org

Johan Bos
University of Edinburgh
Scotland
jbos@cogsci.ed.ac.uk

[1]Patrick Blackburn would like to thank the *L*angue et Dialogue (Language and Dialogue) group at the LORIA research institute in Nancy, France, for covering his travel costs to Bloomington, Indiana.

Chapter 1

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 informativeness 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 exactly how first-order languages are to be interpreted in models. Following this, we introduce two useful extensions of the basic first-order formalism: *function symbols* and *equality*.

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 idealizations 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, if 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, and we are 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* allow us to avoid such problems: they tell us which first-order languages and models belong together.

Here is our first vocabulary:

$$\begin{array}{l}\{\ (\textsc{love},2),\\ \ \ (\textsc{customer},1),\\ \ \ (\textsc{robber},1),\\ \ \ (\textsc{mia},0),\\ \ \ (\textsc{honey-bunny},0),\\ \ \ (\textsc{vincent},0),\\ \ \ (\textsc{yolanda},0)\ \}\end{array}$$

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 *loving* (the 2 indicates that loving is taken to be a two-place relation) and the properties (or 1-place relations) of *being a customer* and *being a robber*. In addition to these relations we're going to be talking about four special entities named *Mia*, *Honey Bunny*, *Vincent*, and *Yolanda* (the 0s indicate that these are the names of entities).

Second, the vocabulary also tells us *how* we can talk about these things. In the above case it tells us that we will be using a symbol LOVE of arity 2 (that is, a 2-place symbol) for talking about loving, two symbols of arity 1 (CUSTOMER and ROBBER) for talking about customers and robbers, and four constant symbols (or names), namely MIA, VINCENT, HONEY-BUNNY, and YOLANDA for naming certain entities of special interest.

In short, 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.

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) consist-

ing 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 names, each constant should be interpreted as an element of D. (That is, for each constant symbol C in the vocabulary, $F(\text{C}) \in D$.) 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. (That is, $F(\text{R})$ 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:

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

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 Vincent and Mia 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 emphasize that the interpretation function has changed, we'll use a different symbol (namely F_2) for it.

$F_2(\text{MIA}) = d_2$
$F_2(\text{HONEY-BUNNY}) = d_1$
$F_2(\text{VINCENT}) = d_4$
$F_2(\text{YOLANDA}) = d_3$

$F_2(\text{CUSTOMER}) = \{d_1, d_2, d_4\}$
$F_2(\text{ROBBER}) = \{d_3\}$
$F_2(\text{LOVE}) = \emptyset$

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

One point is worth emphasizing. Both models just defined are special in the following way: every entity in D is named by exactly one constant. But models don't have to be like this. Consider the model with $D = \{d_1, d_2, d_3, d_4, d_5\}$ and the following interpretation function F:

$F(\text{MIA}) = d_2$
$F(\text{HONEY-BUNNY}) = d_1$
$F(\text{VINCENT}) = d_4$
$F(\text{YOLANDA}) = d_1$
$F(\text{CUSTOMER}) = \{d_1, d_2, d_4\}$
$F(\text{ROBBER}) = \{d_3, d_5\}$
$F(\text{LOVE}) = \{(d_3, d_4)\}$

In this model, not every entity has a name: d_3 is 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.

Exercise 1.1.2 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 1.1.3 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, ..., 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, incidentally, are named after George Boole, a 19th century pioneer of modern mathematical logic.

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 $\tau_1, \ldots, \tau_n$ are terms, then $R(\tau_1, \cdots, \tau_n)$ 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 $R(\tau_1, \cdots, \tau_n)$ is that the entities named by the terms $\tau_1, \cdots, \tau_n$ stand in the relation (or have the property) named by the symbol R. For example

LOVE(PUMPKIN,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

ROBBER(HONEY-BUNNY)

means that the entity named HONEY-BUNNY had 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

$\neg$LOVE(PUMPKIN,HONEY-BUNNY)

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

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

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

$$(\textsc{love}(\textsc{pumpkin},\textsc{honey-bunny}) \vee \textsc{love}(\textsc{vincent},\textsc{mia}))$$

means Either Pumpkin loves Honey-Bunny or Vincent loves Mia. Formulas built using $\rightarrow$ correspond to expressions of the form If ... then ...; for example

$$(\textsc{love}(\textsc{pumpkin},\textsc{honey-bunny}) \rightarrow \textsc{love}(\textsc{vincent},\textsc{mia}))$$

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

$$\exists x\textsc{love}(x,\textsc{honey-bunny})$$

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

$$\forall x\textsc{love}(x,\textsc{honey-bunny})$$

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

$$\exists x\forall y\textsc{love}(x,y)$$

means Someone loves everybody, and

$$\forall x\exists y\textsc{love}(x,y)$$

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

$$\neg\forall y\textsc{person}(y)$$

are $\textsc{person}(y)$, $\forall y\textsc{person}(y)$, and $\neg\forall y\textsc{person}(y)$. We leave it to the reader to give a precise inductive definition of subformulahood (see Exercise 1.1.6), and turn 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. No occurrence of any variable is bound in any atomic formula.
3. 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 $(\phi \rightarrow \psi)$.
4. If an occurrence of any variable is bound in ϕ or in ψ, then that same occurrence is bound in $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $(\phi \rightarrow \psi)$. Moreover, that same occurrence is bound in $\forall y\phi$ and $\exists y\phi$, for any choice of variable y.
5. In any formula of the form $\forall y\phi$ or $\exists y\phi$ (here y can be any variable at all) the occurrence of y that immediately follows the initial quantifier symbol is bound.
6. If an occurrence of a variable x is free in ϕ, then that same occurrence is free in $\forall y\phi$ and $\exists y\phi$, for any variable y distinct from x. On the other hand, all occurrences of x that are free in ϕ, are bound in $\forall x\phi$ and in $\exists x\phi$, and so is the occurrence of x that immediately follows the quantifier.

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 $\forall x \textsc{robber}(x)$. This is a claim that *every* individual is a robber. Roughly speaking, the bound variable x in $\textsc{robber}(x)$ acts as a sort of placeholder. In fact, the choice of x as a variable here is completely arbitrary: for example, the sentence $\forall y \textsc{robber}(y)$ 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 centers on the correct handling of free and bound variables). But before turning to semantic issues, one final remark. 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

$$(\textsc{customer}(\textsc{vincent}) \wedge \textsc{robber}(\textsc{pumpkin})),$$

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

$$\textsc{customer}(\textsc{vincent}) \wedge \textsc{robber}(\textsc{pumpkin}).$$

To help further reduce the bracket count, 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 that $\rightarrow$. What this means, for example, is that the formula

$$\forall x(\neg\textsc{customer}(x) \wedge \textsc{robber}(x) \rightarrow \textsc{robber}(x))$$

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.

One final remark. 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 the course of the book, in many interesting respects the *quantifier free fragment* of a first-order language is far 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 **??** discusses it and introduces the notational simplifications standardly used when working with this important sublogic.

Exercise 1.1.4 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 either Jules is happy or Butch is happy.
3. Either everyone is happy, or Butch and Pumpkin are fighting, or Vincent had 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 either a leather jacket or a dog collar.

Exercise 1.1.5 Which occurrences of variables are bound, and which are free, in the following formulas:

1. $\text{ROBBER}(y)$
2. $\text{LOVE}(x,y)$

3. LOVE(x,y) → ROBBER(y)
4. ∀y(LOVE(x,y) → ROBBER(y))
5. ∃w∀y(LOVE(w,y) → ROBBER(y))

Exercise 1.1.6 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 1.1.7 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 ∀xROBBER(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. 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, ∀xROBBER(x) is a sentence, but its matrix ROBBER(x) is not. Thus an inductive truth definition defined solely in terms of sentences couldn't explain why ∀xROBBER(x) was true in a model, for there are no sentential subformulas 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 idealized) 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, notably in Discourse Representation Theory and other forms of dynamic semantics.

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 of values to variables 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 of values to variables in some model, and let x be a variable. If g' is an assignment of values to variables in the same model, and for all variables y distinct from x such that $g'(\mathrm{y}) = g(\mathrm{y})$

holds, 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 of values to variables in M. Then the relation $M, g \models \phi$ (ϕ is satisfied in M with respect to the assignment of values to variables g) is defined inductively as follows:

$$
\begin{array}{lll}
M, g \models R(\tau_1, \cdots, \tau_n) & \textit{iff} & (I_F^g(\tau_1), \cdots, I_F^g(\tau_n)) \in F(R) \\
M, g \models \neg\phi & \textit{iff} & \text{not } M, g \models \phi \\
M, g \models \phi \wedge \psi & \textit{iff} & M, g \models \phi \text{ and } M, g \models \psi \\
M, g \models \phi \vee \psi & \textit{iff} & M, g \models \phi \text{ or } M, g \models \psi \\
M, g \models \phi \rightarrow \psi & \textit{iff} & \text{not } M, g \models \phi \text{ or } M, g \models \psi \\
M, g \models \exists \mathrm{x}\phi & \textit{iff} & M, g' \models \phi, \text{ for some x-variant } g' \text{ of } g \\
M, g \models \forall \mathrm{x}\phi & \textit{iff} & M, g' \models \phi, \text{ for all x-variants } g' \text{ of } g
\end{array}
$$

(Here 'iff' is shorthand for 'if and only if'.) Note the crucial—and indeed, intuitive—role played by the x-variants in the clauses for the quantifiers. For example, what the clause for the existential quantifier boils down to is this: $\exists \mathrm{x}\phi$ 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.

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 1.1.11.

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 conceptually prior. By making *explicit* the role of variable assignments, it holds up an (admittedly imperfect) mirror to the process of evaluating descriptions in situations while making use of contextual information.

Exercise 1.1.8 Consider the model with $D = \{d_1, d_2, d_3, d_4, d_5\}$ and the following interpretation function F:

$F(\text{MIA}) = d_2$
$F(\text{HONEY-BUNNY}) = d_1$
$F(\text{VINCENT}) = d_4$
$F(\text{YOLANDA}) = d_1$
$F(\text{CUSTOMER}) = \{d_1, d_2, d_4\}$
$F(\text{ROBBER}) = \{d_3, d_5\}$
$F(\text{LOVE}) = \{(d_3, d_4)\}$

Are the following sentences true or false in this model?

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

Exercise 1.1.9 Give a model that makes all the following sentences true:

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

Exercise 1.1.10 When we informally discussed the semantics of bound variables we claimed that in any model of appropriate vocabulary, $\forall \text{x}\text{ROBBER}(\text{x})$ and $\forall \text{y}\text{ROBBER}(\text{y})$ 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 1.1.11 We claimed that when evaluating *sentences*, 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 1.1.12 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 $M \models \phi[\mathrm{x} \leftarrow d_1, \mathrm{y} \leftarrow d_4]$ (assign d_1 to the free variable x, and d_4 to the free variable y) or $M \models \phi[\mathrm{x} \leftarrow d_7, \mathrm{y} \leftarrow d_2]$.

Function Symbols, Equality, and Sorting

We have now presented the core ideas of first-order logic, but before moving on let us discuss three extensions of the basic formalism: first-order logic with function symbols, first-order logic with equality, and sorted first-order logic. Function symbols will play an important technical role when we discuss first-order inference in Chapter **??**, and equality is a fundamental tool in natural language semantics.

Let's first deal with 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 $\tau_1, \ldots, \tau_n$ are terms, then $f(\tau_1, \ldots, \tau_n)$ 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 $f(\tau_1, \ldots, \tau_n)$, then we define $I_F^g(\tau)$ to be $F(f)(I_F^g(\tau_1), \ldots, I_F^g(\tau_n))$. 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 a key role in Chapter **??**, where they will help us formulate an inference system for first-order logic suitable for computational implementation.

Let's 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 $\tau_1 = \tau_2$ rather than 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).

- Add that function symbols redundant (but nice!) with equality.
- Sorting — again redundant, but nice, and used in Chapter 3.

Exercise 1.1.13 Use function symbols and equality to say that Butch's mother is Mia's grandmother.

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 introduce three inference tasks: *querying*, *consistency checking*, and *informativeness checking*. The three tasks are fundamental ones, and can be combined in various ways to deal with interesting problems in the semantics of natural language. To give a classic example, a key part of the van der Sandt algorithm for handling presupposition (discussed in the advanced companion to this book), is a clever blend of consistency and informativeness checking.

By the end of the section the reader should have a good understanding of what the 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 informativeness 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 specialized 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 natural 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 4) it is possible to translate some kinds of natural language questions into first-order logic. And if we do this, we have a simple way of answering them: we simply see if their translations are satisfied in the model. In Chapter 4 we construct a simple question answering system which works this way.

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 realizable'. 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 semantics. Suppose we are analyzing 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 pin down 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 idealized situations, this means that satisfiable formulas are those which describe 'conceivable', or 'possible', or 'realizable' situations. For example, ROBBER(MIA)

describes a realizable 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 'unrealizable' situations. For example, $\textsc{robber}(\textsc{mia}) \wedge \neg\textsc{robber}(\textsc{mia})$ describes something that is unrealizable: 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 $\{\phi_1, \ldots, \phi_n\}$ is satisfiable if $\phi_1 \wedge \cdots \wedge \phi_n$ 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 $\{\phi_1, \ldots, \phi_n\}$ is unsatisfiable if $\phi_1 \wedge \cdots \wedge \phi_n$ 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 realizable 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, and from time to time we shall use this terminology. But we prefer to talk about consistency checking, for it emphasizes the pre-theoretic notion we are trying to capture. Moreover, as we shall learn in Chapter 4, there is a also a mathematical precise (but *non* model-theoretic) concept called consistency that turns out to correspond *exactly* with the (model-theoretic) notion of satisfiability. All in all, 'consistency checking' is a good terminological choice. 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's 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-analyze consistency checking from a perspective that emphasizes 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 the computational semantics.

Exercise 1.2.1 Consider the following formula:

$$\forall x \exists y \textsc{neighbour}(x,y) \wedge \forall x \neg \textsc{neighbour}(x,x)$$
$$\wedge \forall x \forall y \forall z (\textsc{neighbour}(x,y) \wedge \textsc{neighbour}(y,z) \rightarrow \textsc{neighbour}(x,z)).$$

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

The Informativeness Checking Task

The main goal of this section is to get to grips with pre-theoretic concept of informativeness (and uninformativeness). 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 informativeness.

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 ROBBER(x) $\vee$ $\neg$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 ROBBER(x) is an invalid formula: it is not satisfied 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 $\phi_1, \ldots, \phi_n$, and ψ are a finite collection of first-order formulas. Then we say that the argument with *premises* $\phi_1, \ldots, \phi_n$ 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

$$\phi_1, \ldots, \phi_n \models \psi$$

means that the argument with premises $\phi_1, \ldots, \phi_n$ 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 $\phi_1, \ldots, \phi_n$, or that ψ is a *logical consequence* of $\phi_1, \ldots, \phi_n$, or that ψ is a *semantic consequence* of $\phi_1, \ldots, \phi_n$.

An argument that is not valid is called *invalid*. That is, an argument with premises $\phi_1, \ldots, \phi_n$ and conclusion ψ is 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

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

Validity and valid arguments are closely related. For example, the argument with premises $\forall$x(ROBBER(x) $\rightarrow$ CUSTOMER(x)) and ROBBER(MIA) and the conclusion CUSTOMER(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, for they embody the abstract patterns that underly logical truth, and they regard valid arguments as important because of the perspective they offer on correct mathematical reasoning. (In a nutshell: for logicians, validities 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 informativeness and uninformativeness.

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 informativeness turns out to be linguistically relevant, we shall define the following task:

> **The Informativeness 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 uninformativeness too. If $\phi_1, \ldots, \phi_n \models \psi$, and we already know that $\phi_1, \ldots, \phi_n$, 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 $\phi_1, \ldots, \phi_n \models \psi$, then we shall say that ψ is *uninformative with respect to* $\phi_1, \ldots, \phi_n$. On the other hand, suppose that $\phi_1, \ldots, \phi_n \not\models \psi$, and that we already know that $\phi_1, \ldots, \phi_n$. Then if we are told ψ, we clearly *have* learned something new. Hence, if $\phi_1, \ldots, \phi_n \not\models \psi$, then we shall say that ψ is *informative with respect to* $\phi_1, \ldots, \phi_n$.

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

But *why* should linguists care about informativeness? The point is this: like inconsistency, uninformativeness 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 informativeness 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 uninformativeness 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 informativeness checking.

And this brings us to the next question: is informativeness checking computable? And once again the answer is *no*, not fully. Like the consistency checking task, the informativeness checking task is undecidable. We're not going to prove this result, but given our previous discussion it is probably clear that informativeness checking is likely to be tough. After all, informativeness checking is a highly abstract task: validity means satisfiable in *all* models, and there's an awful lot of awfully big of 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 informativeness checking. Instead of viewing informativeness in terms of satisfaction in all models, it is possible to reanalyze it in terms of certain kinds of symbol manipulation. This proof-theoretic perspective makes it possible to create software that offer effective partial solutions to the informativeness checking problem, and (as we shall see in Chapter 6) we can apply this software to linguistic issues.

Exercise 1.2.2 The *Semantic Deduction Theorem* for first-order logic says that $\phi_1, \ldots, \phi_n \models \psi$ if and only if $\models (\phi_1 \wedge \cdots \wedge \phi_n) \rightarrow \psi$. (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 sentences 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 $\psi \wedge (\phi \wedge \theta)$, and that $(\phi \vee \psi) \vee \theta$ is logically equivalent to $\psi \vee (\phi \vee \theta)$ (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 $\vee$).
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 commutative and 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 $\psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4$. We often adopt this convention.

Relating Consistency and Informativeness

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 informativeness checking task) are extremely difficult. Indeed, in order to make progress with them, in Chapters 4 and 5 we're going to have to rethink them from a symbol manipulation perspective. 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.

These relationships have practical importance. 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) or not. 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.

We will say more about these relationships, and about the tools that can help us establish them, in Chapters 4 and 5.

Exercise 1.2.4 In the text we gave a simple argument establishing the first of the relationships listed above. Establish the truth of the remaining three relationships.

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 informativeness 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 the `model/2` term. 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. Observe that it also gives us a lot of negative information about Yolanda: she's not a customer, she's not a robber, and she neither loves nor is loved by any of the others. 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. As this example makes clear, our Prolog representation format for models really does cover all the options allowed for by the set-theoretic definition.

```
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 are 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,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 to us 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. $D = \{d_1, d_2, d_3\}$,
   $F(\text{THE-WOLF}) = d_1$,
   $F(\text{MARSELLUS}) = d_2$,
   $F(\text{BUTCH}) = d_3$,
   $F(\text{BOXER}) = \{d_3\}$,
   $F(\text{PROBLEM-SOLVER}) = \{d_1\}$.

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

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 functor of the term is the `model/2` predicate, 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 (for languages 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.*

This advantage of this is that it allows us to use Prolog's inbuilt 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 Pro-

log term `love(vincent,mia)`, and HATE(BUTCH,X) would be represented by `hate(butch,X)`. Note that first-order atomic sentences (for example BOXER(BUTCH)) are represented by exactly the same Prolog term (namely `boxer(butch)`) that is used to represent the fact that Butch is a boxer in our Prolog representation of models. This will help keep the implementation of the satisfaction clause for atomic formulas simple.

Recall that there is also a special two-place relation symbol, namely the equality symbol =. We shall use the functor `eq/2` as its Prolog representation. 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

```
ex(X,Phi).
```

The Satisfaction Definition in Prolog

We turn to the final task: evaluating (representations of) formulas in (representations of) models with respect to (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`?

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 1.1.12: 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 model 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 make a particular subformula true or false in a model. If subformula is flagged `pos` it means we are trying to make it true, and if it is flagged `neg` it means we are trying to make it 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. Just for the fun of it, 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,Pol):-
   satisfy(or(not(Formula1),Formula2),Model,G,Pol).
```

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 1.3.6 below.

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

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

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

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 rechristened 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 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` 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` will simply be the domain `D` itself. In short, obtaining `D` as the result of our `setof` 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(ex(X,not(Formula))),Model,G,Pol).
```

Needless to say, a pair of clauses which directly handle the universal quantifier could be given. Exercise 1.3.7 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 course of 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 to turn the formula into a list, and then call `i/4` to interpret the term. We then use `memberList` 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 most convenient to have a separate file containing a number of example models. In the file `exampleModels.pl` you will find the following three examples:

```
example(1,
        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)])])).

example(2,
        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,[])])).

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)])])).
```

Note that the first two of these examples are the models used in the text to introduce our Prolog representation format for models. The third is new, and in a different vocabulary.

So, let's now write a driver 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,pos), !,
        printStatus(pos)
    ;
        printStatus(neg)
    ).

printStatus(pos):- write('Satisfied in model. ').
printStatus(neg):- write('Not satisfied in model. ').
```

For convenience, we have also included a predicate `evaluate/2`, defined as follows:

```
evaluate(Formula,Example):-
   evaluate(Formula,Example,[]).
```

That is, it calls `evaluate/3` with an empty assignment list. This saves typing if we want to evaluate a sentence.

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

```
test(ex(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 ex(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 `evaluate/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 `evaluate` 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 Try the model checker on example model 1 with the examples

```
ex(X,and(customer(X)),ex(Y,robber(Y)))
ex(X,and(customer(X),ex(Y,robber(Y))))
ex(X,and(customer(X)),ex(X,robber(X)))
ex(X,and(customer(X),ex(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 1.3.6 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 1.3.7 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.

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. And there are some interesting ways of using it. For example, we can use it to

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.

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,customer,[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 dialogue 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', one of whose subformulas is a Prolog variable, 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 1.3.8 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(forall(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 `Not satisfied in 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 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 1.3.8 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 `ex(X,or(customer(X),Z))` and `ex(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 commanding "`?- trace.`" and deactivated by "`?- notrace.`").

Exercise 1.3.9 What happens if you use the basic model checker to evaluate constants as formulas, and why? If this is unclear, perform a trace.

Exercise 1.3.10 [hard] Modify the basic model checker so that it classifies the kind of example examined in the previous two exercises as ill-formed input.

Exercise 1.3.11 Try out the new model checker (`modelChecker2.pl`) on the formulas `customer(X)` and `not(customer(X))`. Why (exactly) 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 from a computational perspective? We now discuss this. First we take an inferential perspective, and then we consider first-order logic's representational capabilities.

Inferential Capabilities

First-order logic is sometimes called classical logic, and it merits the description: 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 powers of first-order logic in extraordinary mathematical detail.

Nor have the inferential properties of first-order logic been neglected: the discipline called proof theory explores 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 informativeness checking tasks (which were defined in model-theoretic terms) is to reformulate them as concrete symbol manipulation tasks. Proof theorists and researchers in automated reasoning have devised many ways of doing this, and explored their properties and interrelationships 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 informativeness 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. So one reason for being interested in first-order logic is simply this: if you use first-order logic, a wide range of sophisticated inference tools are 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* (such as *world knowledge* and *lexical knowledge*) into the inference process.

Here's an example. When discussing informativeness 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 intimately 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 piece of lexical knowledge (that is, knowledge of the meanings of words and how they are inter-related), expressed here as a formula of first-order logic:

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

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

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

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 our lexical and world knowledge) then the conclusion is true also. In short, the uninformativeness of our little discourse actually rests on an inference that draws on both lexical knowledge and world knowledge. Modeling 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 informativeness 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.

But there is a price to pay. The interesting thing about first-order logic is not merely that it is well behaved inferentially, but that (as we shall shortly see) it offers expressive capabilities capable of handling a significant portion of natural language semantics. Propositional modal, hybrid and description logics don't have the expressivity for such detailed semantic analysis. They are likely to play an increasingly important role in computational semantics, but they are probably best suited for giving efficient solutions to relatively specialized 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 inferential techniques (such as tableau and resolution) are not parochial. Although initially developed for first-order logic, they have been successfully transferred to other logical systems (such as the modal/hybrid/description logic family). So studying first-order inference techniques is a useful first step towards understanding inference in other potentially useful 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 shortcomings as a natural

language representational formalism, and we shall draw attention to an important example. Nonetheless, first-order logic is capable of doing a lot of work for us. It's not called classical logic for nothing: it is an extremely flexible tool.

It is quite 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 idealizations 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:

$F(\text{MIA}) = d_1$
$F(\text{MONDAY}) = d_2$
$F(\text{TUESDAY}) = d_3$
$F(\text{WEDNESDAY}) = d_4$
$F(\text{PERSON}) = \{d_1\}$
$F(\text{DAY}) = \{d_2, d_3, d_4\}$
$F(\text{PRECEDE}) = \{(d_2, d_3), (d_3, d_4), (d_2, d_4)\}$
$F(\text{HAPPY}) = \{(d_1, d_2), (d_1, d_3)\}$

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, you are then free (if you so desire) to talk about these models using a first-order language of appropriate signature. 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 signature. Or perhaps you feel that the 'possible world semantics' introduced by philosophical logicians to analyze such concepts as necessity and belief is a promising way to handle 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 signature. 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.

Clever use of models sometimes enables first-order logic to provide useful approximate solutions to expressivity demands 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, elements of the 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:

$$\forall \mathrm{P} \forall \mathrm{x}((\textsc{good-boxer}(\mathrm{x}) \rightarrow \mathrm{P}(\mathrm{x})) \rightarrow \mathrm{P}(\textsc{butch})).$$

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, it seems we've 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, ...) 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 of second-order logic. Moreover, in some respects this first-order treatment of second-order logic is better behaved that the 'standard' interpretation (in which properties are domain subsets, one-place relations are sets of pairs of domain subsets, and so on). 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 interesting aspects of second-order logic. It may be an approximation, but it is a good 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 expressivity demands 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 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 to work with 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.

First-order 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.

Chapter 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?

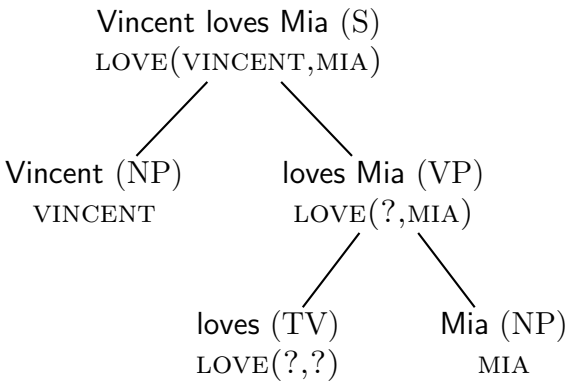
The key to answering this is to be more precise about what we mean by 'systematic'. Consider 'Vincent loves Mia'. Its semantic content is at least

partially captured by the first-order formula LOVE(VINCENT,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, to 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 syntactic 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 ultimately leads to the lexical items. (That is, our notion of syntactic structure should allow us to classify the sentence into subparts, sub-subpart, 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).

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 are 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.

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`.)

```
s --> np, vp.
np --> pn.
np --> det, noun.
pn --> [vincent].
pn --> [mia].
det --> [a].
det --> [every].

noun --> [woman].
noun --> [foot,massage].
vp --> iv.
vp --> tv, np.
iv --> [snorts].
iv --> [walks].
tv --> [loves].
tv --> [likes].
```

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 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 extra arguments (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 have chosen to fill in the syntactical 'black box' 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 answer 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 `N`th 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))
```

Also give query that yields the infix representation, and the response.

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)))
```

Also give infix form.

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 short cuts. 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 contribution 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 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 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:

```
s(Sem)--> np(X,SemVP,Sem), vp(X,SemVP).
```

Hence when we have a noun phrase of the form `np(SemPN,Sem,Sem)`, two things happen. First, because of the doubled `Sem` variable, the semantic representation of the verb phrase (that is, `SemVP`) becomes the semantic reprsentation 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 maneuver 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 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:

$$\lambda x.\textsc{man}(x)$$

The prefix λx. binds the occurrence of x in MAN(x). We often say that the prefix λx. *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 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:

λx.MAN(x)@VINCENT.

This compound expression consists of the abstraction λx.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 λx.MAN(x)@VINCENT is the application of the functor λx.MAN(x) to the argument VINCENT.

But what is such an expression good for? It is an instruction to throw away the λx. 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:

MAN(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 λx. in λx.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: λy.$\exists$x(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 λx.MAN(x), λu.MAN(u), and λv.MAN(v) are intended to be equivalent, and so are λu.$\exists$x(WOMAN(x)$\wedge$u@x) and λv.$\exists$x(WOMAN(x)$\wedge$v@x). All these expressions are functors which when applied to an argument $\mathcal{A}$, replace the bound variable by the argument. No matter which argument $\mathcal{A}$ we choose, the result of applying any of the first three expressions to $\mathcal{A}$ and then β-converting should be MAN($\mathcal{A}$), and the result of applying either of the last two expressions to $\mathcal{A}$ should be $\exists$x(WOMAN(x)$\wedge\mathcal{A}$@x). To put it another way, replacing the bound variables in a lambda expressions (for example, replacing the variable x in λx.MAN(x) by u to obtain λu.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 $\forall$xROBBER(x) and $\forall$yROBBER(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 $\mathcal{E}$ can be obtained from a lambda expression $\mathcal{E}'$ by α-conversion then we say that $\mathcal{E}$ and $\mathcal{E}'$ are *α-equivalent* (or that they are *alphabetic variants*). Thus λx.MAN(x), λy.MAN(y), and λz.MAN(z) are all α-equivalent, and so are the expressions λy.$\exists$x(WOMAN(x)$\wedge$y@x) and λz.$\exists$y(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 $\mathcal{E}$, but when we actually work out some semantic construction, we might use an α-equivalent expression $\mathcal{E}'$ instead of $\mathcal{E}$ 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 $\mathcal{F}$ in $\lambda x.\mathcal{F}$ is a complex expression containing many occurences of λ, $\forall$ and $\exists$. It could happen that when we apply $\lambda x.\mathcal{F}$ to an argument $\mathcal{A}$, some occurrence of a variable that is free in $\mathcal{A}$ becomes bound by a lambda operator or a quantifier when we substitute it into $\mathcal{F}$. 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 $\lambda x.\mathcal{F}$ as the functor: any α-equivalent expression will do. By relabelling all the bound variables in $\lambda x.\mathcal{F}$ we can always obtain an α-equivalent functor that doesn't bind any of the variables that occur free in $\mathcal{A}$, 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 useful (we discuss it briefly in the Notes at the end of the chapter, and present it in more detail in Appendix **??**) 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 '<sans-serif>every boxer walks</sans-serif>' using lambdas? The first step is to assign lambda expressions to the different basic syntactic categories. We assign the determiner '<sans-serif>every</sans-serif>', the common noun '<sans-serif>boxer</sans-serif>', and the intransitive verb '<sans-serif>walks</sans-serif>' the following lambda expressions:

'every': λu.λv.$\forall$x(u@x$\rightarrow$v@x)

'boxer': λy.BOXER(y)

'walks': λz.WALK(z)

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

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

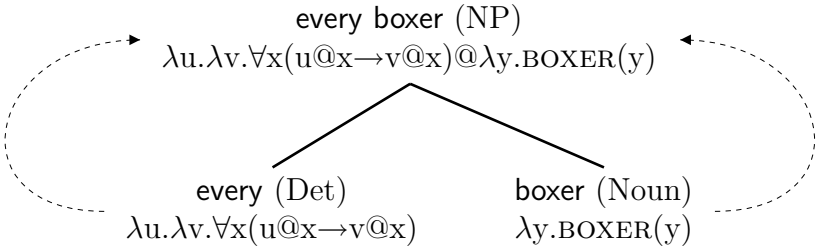
If we use the Prolog variable `U` instead of `Restr`, and `V` instead of `Scope` this becomes

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

which is quite similar to λu.λv.$\forall$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.



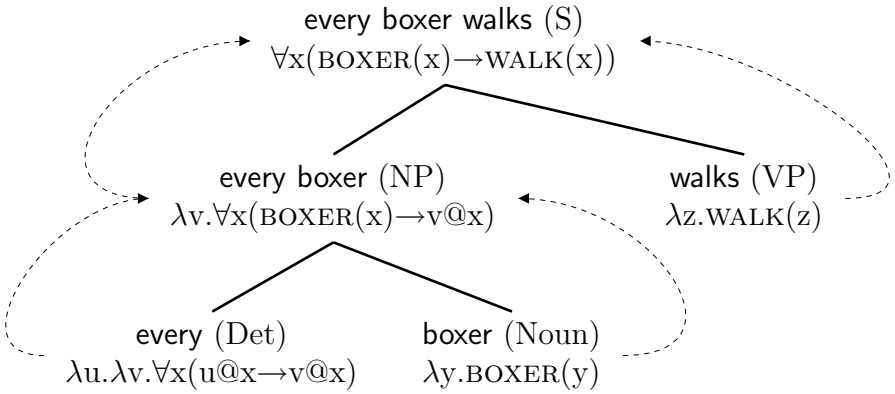
Now, applications are instructions to carry out β-conversion, so let's do what is required. (Note that as there are no free occurrences of variables in the argument expression, there is no risk of accidental variable capture, so we don't need to α-convert the functor.) Performing the demanded substitution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

'every boxer': $\lambda v.\forall x(\lambda y.\textsc{boxer}(y)@x \rightarrow v@x)$

But this expression contains the subexpression $\lambda y.\textsc{boxer}(y)@x$. Now, the argument x is a free variable, but x does not occur bound in the functor $\lambda y.\textsc{boxer}(y)$, 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': $\lambda v.\forall x(\textsc{boxer}(x) \rightarrow v@x)$

We can't perform any more β-conversions, so lets carry on with the analysis of the sentence. The following tree shows the final representation we obtain:



Why is the S node associated with $\forall x(\textsc{boxer}(x) \rightarrow \textsc{walk}(x))$? It's certainly what we want, but where does it come from?

In fact we obtain it by a procedure analogous to that 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': $\lambda v.\forall x(\textsc{boxer}(x) \rightarrow v@x)@\lambda z.\textsc{walk}(z)$

Performing β-conversion yields:

'every boxer walks': $\forall x(\text{BOXER}(x) \rightarrow \lambda z.\text{WALK}(z)@x)$

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

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

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': $\lambda u.u@\text{MIA}$

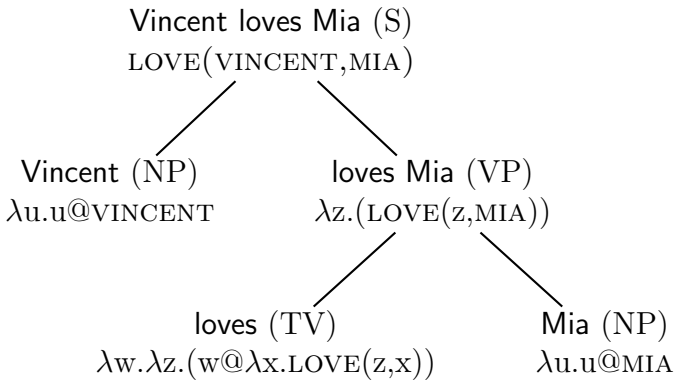
'Vincent': $\lambda u.u@\text{VINCENT}$

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:

λw.λz.(w@λx.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 understand the role-reversing idea used in the TV's semantic representation.

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 λx.$\mathcal{E}$ 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 applications $\mathcal{F}$@$\mathcal{A}$ 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.

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 `app` to build semantic representations out of the component semantic representations in the manner suggested by task 3. Compared with the code in 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 λx.x@VINCENT is the translation of 'Vincent', and λx.x@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 λu.λv.$\forall$x.(u@x$\rightarrow$v@x) and λu.λv.$\exists$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` explicitly records how it all fits together. Here is an example query:

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

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

Show the infix form

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 it's 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)`:

Formula	**Stack**	**Comment**
`app(lam(X,smoke(X)),mia)`	`[]`	start with empty stack
`lam(X,smoke(X))`	`[mia]`	applications push argument onto stack
`smoke(mia)`	`[]`	abstractions pop element from stack

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` 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:

Formula	**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 three, 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 `mia` for `Y`. In short, the stack keeps track of which functor corresponds to which argument.

Let's look at a final example. This shows what happens when the semantic representation of a proper name is applied to a complex argument:

Formula	**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)`	`[]`

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 seen, the stack is needed because functors are not always in the required syntactic form (they don't always have lambda as the outermost operator), so we sometimes need to β-convert the functor first. Bearing this is 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 `(Functor@Argument)`), 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 in the first place.) 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, we supply a driver predicate which initialises the β-conversion process with an empty list:

```
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 binding 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':

'every man' (NP): λu.$\forall$y(MAN(y)$\rightarrow$u@y)
'loves a woman' (VP): λx.$\exists$y(WOMAN(y)$\wedge$LOVE(x,y))

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

λu.$\forall$y(MAN(y)$\rightarrow$u@y)@λx.$\exists$y(WOMAN(y)$\wedge$LOVE(x,y)).

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

$$\forall y(\textsc{man}(y) \rightarrow \lambda x.\exists y(\textsc{woman}(y) \wedge \textsc{love}(x,y))@y).$$

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:

$$\lambda x.\exists y(\textsc{woman}(y) \wedge \textsc{love}(x,y))@y$$

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:

$$\forall y(\textsc{man}(y) \rightarrow \exists y(\textsc{woman}(y) \wedge \textsc{love}(y,y)))$$

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

$$\lambda x.\exists y(\textsc{woman}(y) \wedge \textsc{love}(x,y))@y$$

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

$$\lambda x.\exists z(\textsc{woman}(z) \wedge \textsc{love}(x,z))@y.$$

Doing it this way yields

$$\forall y(\textsc{man}(y) \rightarrow \exists z(\textsc{woman}(z) \wedge \textsc{love}(y,z)))$$

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.

Following text probably needs changing. Emphasize that we want alpha-conversion for other purposes (elimanting alphabetic variants) not just for beta conversion.

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, and is introduced by the driver clause for α-conversion:

```
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 existential 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,lam(Y,F2)):-
   nonvar(Expression),
   Expression = lam(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`. 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 = lam(_,_),
   compose(F1,Symbol,Args1),
   alphaConvertList(Args1,Subs,Free1-Free2,Args2),
   compose(F2,Symbol,Args2).
```

And `alphaConvertList/3` 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` and the `all` 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).
```

```
X = _G504
R = some(_G595,and(man(_G595),all(_G639,woman(_G639))))

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. We have created 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 and the output will be a series of entries of the following form:

```
Expression: app(lam(_G2,app(_G2,mia)),lam(_G3,walk(_G3)))
Expected: walk(mia)
Converted: walk(mia)
Result: ok
```

This tells us that our `betaConversion` 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(_G5,lam(_G8,like(_G8,_G5))),mia)
Expected: lam(_G4,like(_G4,mia))
Converted: lam(_G1,like(_G1,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? With this code: **Comment**

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

The code introduced in this section is fundamental to the rest of the book. It is important that the reader gets to grips with it. A good place to start is with the following exercises.

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`. 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 Some readers may still wonder whether we really need α-conversion. After all (they might argue) all we really need is a guarantee that we will never encounter two distinct occurrences of a binding operator ($\forall$, $\exists$ or λ) binding the same variable. And couldn't we write our lexicons in such a way that every distinct occurrence of a binding operator binds a distinct variable?

Well, we could do this, but apart from being an unwieldy (lexicons can be very big so we would need a lot of distinct variables) writing lexicons in this way is *not* enough to ensure that distinct occurences of binding operators bind distinct variables. Why not? (Hint: 'and' is translated by the following lambda expression:

$$\lambda \mathrm{u}.\lambda \mathrm{v}.\lambda \mathrm{x}.(u@x \wedge v@x).$$

Use this to build a semantic representation for 'Vincent and Mia dance'. What do you observe?)

Exercise 2.4.3 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` (the one that doesn't call on `alphaConvert`) 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 the semantic representation for 'and' given in the previous question) and 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 file `betaConversionTestSuite.pl` with α-conversion turned off. You will encounter some surprising behaviour (for example, in one case Prolog fails to terminate). Explain these failures.

Exercise 2.4.4 [intermediate] Give the Prolog code for lexical entries of ditransitive verbs such as 'offer' in 'Vincent offers Mia a drink'.

Exercise 2.4.5 [intermediate] Find a suitable lambda expression for the lexical entry of the determiner 'no', and then give the corresponding Prolog code.

Exercise 2.4.6 [intermediate] Extend the DCG so that it covers sentences such as 'everyone dances' and 'somebody snorts'.

Exercise 2.4.7 [hard] Pereira and Shieber 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 2.4.8 [intermediate] In our discussion in the text we assumed that β-conversion was carried out only after the DCG had been built for the entire sentence. Change the DCG so that β-conversion is interleaved with semantic construction. This can be done by making β-conversion part and parcel of the grammar. For instance, change the rule for sentences to

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

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`
Lots of examples to test our implementation of β-conversion.

`comsemPredicates.pl`
Definitions of some auxiliary predicates.

2.5 Grammar Engineering

With our notation for lambda abstraction and functional application, and our implementation of β-conversion, we have the basic semantic construction tools needed in this book. 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 representation language).

Grammar engineering principles have strongly influenced the design of the larger grammar we shall now discuss—and *not*, we would like to stress, purely for pedagogic reasons. We have experimented with many ways of computing semantic representations (for example, in the following chapter we will consider four different techniques for coping with scope ambiguities). Moreover we have checked that our approach is compatible with other semantic representation languages (notably Discourse Representation Structures) 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.

We have opted for a fairly simple two-dimensional grammar architecture consisting of a collection of *syntax rules* and corresponding *composition rules*, and a *lexicon* with a set of *semantic macros*.

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 most syntactic categories in an easily extractable form; again, this component will stay fixed throughout the book. The syntax rules together with the lexical entries describe the *syntactic* part of the grammar. (Actually, strictly speaking, this is not true. As we will see, the lexical entries also contain some simple facts about what is usually called *lexical semantics*; for example, that 'big' is the opposite of 'small'.)

The *semantic* part of the grammar consists of the composition rules and the crucial semantic macros. The composition rules state how semantic representations of a complex category is computed out of the semantic repre-

sentations of its parts. The semantic macros are where we write the lambda definitions that tell us how the word combines level at which 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 and the Prolog predicates that implement it.

	grammar	lexicon
syntax	syntax rules (`-->/2`)	lexical entries (`lexEntry/2`)
semantics	composition rules (`combine/2`)	semantic macros (`semMacro/2`)

The DCG and the lexical entries are stored in two separate files. The semantic rules and macros, because they are dependent on the choice of semantic formalism, and we will change this as we gradually add more and more sophistication to our semantic analysis, are part of the main file.

The Syntax Rules

Here are some of the core DCG rules that we are using. (We won't show all the rules but instead explain only some of them, as they bear all striking similarities. Please consult the file `englishGrammar.pl` for the complete set of rules.) These rules license a number of semantically important constructions, such as proper names, determiners, pronouns, relative clauses, the copula construction, and coordination. The DCG rules were using 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 feature value pairs expressing this grammatical information. And secondly, all syntax rules come with a hook to a semantics rule, in the form of the `combine/2` notation. Here is an example illustrating these two extensions:

```
s([coord:no,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 → NP VP rule, but it tells us slightly more: 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.

Now consider some rules that show how coordination is dealt with. Implementing coordination in DCGs requires special attention because DCGs should stay away from lect-recursive syntax rules (that is, rules of the form X → X Y), as this will cause Prolog to fall in an infinite loop. The way we solve this is simple and effective, and introduces a feature `coord` that can either have the value `yes` or `no`.

```
np([coord:yes,num:pl,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])}.
```

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

Finally, we've got the lexical rules. These are rules that apply to terminal symbols, the actual strings in the input of the parser, and need to call the lexicon to check if a string belongs to the syntactic category searched for.

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

In this example for nouns, the semantic macro is used to construct the actual semantic representation for a noun. Each lexical category is associated with such a macro, and this enables us to abstract away from specific types

of structures. So we have set up the syntax rules in such a way that we are independent from the semantic theory we want to work with, and this is an attractive property as we are going to change the underlying semantic representations in the following chapters.

All these rules will work for us unchanged throughout the book. The reader can find a complete list in the file `englishGrammar.pl`. Furthermore, the file `lambda.pl`, that contains the implementation of the lambda calculus of this chapter, already uses these rules in the way described here.

One shortcoming should be mentioned. We implemented 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 Composition Rules

Let's turn to the composition rules. 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]).
```

Of course, because `combine/3` offers us extreme flexibility in implementing semantic construction, we can also choose to apply β-conversion directly:

```
combine(t:Converted,[s:Sem]):-
   betaConvert(Sem,Converted).
```

At this point we would like to stress once more the modularity of this approach to grammar engineering. For instance, if one were to replace the lambda calculus with another tool for computing semantic representations (which we think is probably not a good idea), the `combine/2` rules would be the right place to do it.

The Lexicon

Our lexicon lists information about the words belonging to most syntactic categories in a form useful for the interface component that we shall shortly define. The general format of a lexical entry is `lexEntry(Cat,Features)` where `Cat` is the syntactic category, and `Features` a list of feature-value pairs. This list contains specific information depending on the type of lexical entry, such as sortal properties for nouns, gender information proper names, and inflectional status for verbs.

We will go through a few examples to get an idea how the lexicon is set up. Let's first consider nouns:

```
lexEntry(noun,[symbol:beverage,syntax:[beverage],hypernyms:[drinkable,food]]).
lexEntry(noun,[symbol:burger,syntax:[burger],hypernyms:[edible,food]]).
lexEntry(noun,[symbol:boxer,syntax:[boxer],hypernyms:[person]]).
lexEntry(noun,[symbol:boss,syntax:[boss],hypernyms:[person]]).
```

Nouns in our lexicon come with three features: `symbol`, the relation symbol used to compute the semantic representation), `syntax`, a list of words describing the appearance of the lexical entry in a phrase (note that we use a list here to deal with multi-word phrases would we like to do so), and finally `hypernyms`, information about the noun's hypernym relation to other symbols of our semantic vocabulary. The latter information will be used to compute a simple ontology that will be useful for generating background knowledge for inference tasks. In Chapter 6 we will explore this information.

Other lexical entries are much simpler of nature. A case in point are the entries for determiners, containing no semantic information at all:

```
lexEntry(det,[syntax:[every],type:uni]).
lexEntry(det,[syntax:[a],type:indef]).
lexEntry(det,[syntax:[the],type:def]).
lexEntry(det,[syntax:[which],type:wh]).
```

The semantic contribution of determiners is not simply a constant or predicate 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 macros.

Proper names are coded in the lexicon with gender information. Note the use of the multi-word lexeme here for 'Vincent Vega'.

```
lexEntry(pn,[symbol:vincent,syntax:[vincent,vega],gender:male]).
lexEntry(pn,[symbol:yolanda,syntax:[yolanda],gender:female]).
```

Another example of lexical entries encoding background knowledge are adjectives. Here we introduce antonyms relations, expressing disjointness between semantic concepts. Again, we will explain in Chapter 4 how we will explore this information.

```
lexEntry(adj,[symbol:big,syntax:[big],antonyms:[small]]).
lexEntry(adj,[symbol:kahuna,syntax:[kahuna],antonyms:[]]).
lexEntry(adj,[symbol:male,syntax:[male],antonyms:[female]]).
lexEntry(adj,[symbol:married,syntax:[married],antonyms:[single]]).
```

As a final example (please refer to `englishLexicon.pl` for a complete listing of the lexicon), consider the lexical entries for the transitive verb to clean:

```
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]).
```

This way of setting up a lexicon offers natural expansion options. For example, if decided to develop a grammar that dealt with both singular and plural forms of nouns (which we won't do so in this book), it is just a matter of extending the general format of entries with one or more fields.

The Semantic Macros

We now come to arguably the most important part of the grammar, the semantic macros. As the introduction of `combine/3` 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 macros is where the real semantic work will be done. Let's consider some examples of semantic macros right away.

```
semMacro(pn,M):-
   M = [symbol:Sym,
        sem:lam(U,app(U,Sym))].

semMacro(pn,M):-
   M = [symbol:Sym,
        sem:lam(U,app(U,Sym))].

semMacro(noun,M):-
   M = [symbol:Sym,
        sem:lam(X,Formula)],
   compose(Formula,Sym,[X]).

semMacro(tv,M):-
   M = [symbol:Sym,
        sem:lam(K,lam(Y,app(K,lam(X,Formula))))],
   compose(Formula,Sym,[Y,X]).

semMacro(tv,M):-
   M = [symbol:Sym,
        sem:lam(K,lam(Y,app(K,lam(X,F))))],
   compose(F,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 `lambda(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 a the required lambda expression. For example,

the symbol `boxer` would give the formula `lambda(X,boxer(X))`. The macro for transitive verbs is similar to that of the macro for nouns, apart from handling two variables rather than just one.

As we've already mentioned, the interface also contains self-contained entries for the determiners. Here they are:

```
semMacro(det,M):-
   M = [type:uni,
        sem:lam(U,lam(V,all(X,imp((app(U,X)),(app(V,X)))))))].

semMacro(det,M):-
   M = [type:indef,
        sem:lam(U,lam(V,some(X,and((app(U,X)),(app(V,X)))))))].

semMacro(det,M):-
   M = [type:indef,
        sem:lam(U,lam(V,some(X,and(app(U,X),app(V,X))))))].
```

These, of course, are just the old-style 'lexical entries' we are used to.

Exercise 2.5.1 [easy] Find out how copula verbs are handled in the lexicon and grammar, and how the semantic representation for sentences like 'Mia is a boxer' and 'Mia is not Vincent' are generated.

Exercise 2.5.2 [moderate] Extend the grammar such that is allows expressions of the form Vincent is male, Mia and Vincent are cool, and Marsellus or Butch is big.

Exercise 2.5.3 [moderate] Extend the grammar such that is allows expressions of the form 'Vincent and Jules are in a restaurant' and 'Butch is without a weapon'. (Represent the meaning of the preposition 'without' as a simple two-place relation symbol.)

Exercise 2.5.4 [hard] 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 $\exists X(\text{MAN}(X) \wedge \neg \exists Y(\text{WEAPON}(Y) \wedge \text{WITH}(X,Y)))$.

From now on, we will always use the syntax rules and the lexicon (with the exceptions of a few pedagogic explorations of other formats). To put it another way: *from now on the primary locus of change will be the semantic*

macros and the composition rules. For example, it is here that we will develop treatments of quantifier scope, pronoun resolution, and presupposition accommodation. For a complete listing of the macros we have just been discussing, see the file `lambda.pl`, but we shall see many more types of semantic macros as we work our way through the book.

Programs for the full grammar fragment

`lambda.pl`
This is the main file for the implementation of the lambda calculus using the extended grammar

`betaConversion.pl`
The predicates that implement β-conversion.

`alphaConversion.pl`
The predicates that implement β-conversion.

`readLine.pl`
Module for converting strings into lists.

`sentenceTestSuite.pl`
Test suite with sentences.

`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.

Chapter 3

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.

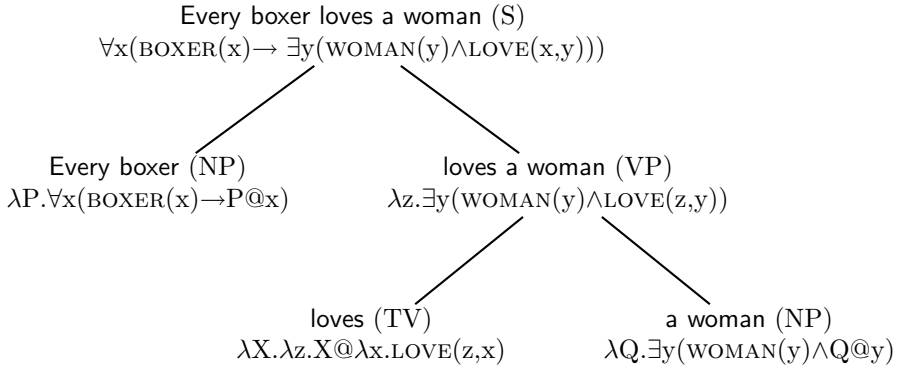
We are going to investigate four different approaches to scope ambiguities: Montague's original method, two storage based methods, and a modern underspecification based approach called hole semantics. We will implement the storage based approaches and hole semantics.

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 representation 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 usual

to say that in the first reading 'every boxer' *has scope over* (or *outscopes*) 'a woman', while in reading (2) 'a woman' has scope over 'every boxer'.

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' outscopes '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 clearly strikes at the very heart of our semantic construction methodology. As scope ambiguities are extremely common, so we urgently need a solution.

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 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 15 readings. Some of these readings turn out to be logically equivalent, but even after we take this into account we are still left with 11 distinct readings. Here they are:

Node 1: (reading 1)
1:∀A(criminal(A) & male(A) → ∃B(bkburger(B) & (neuter(B) & ∀C(∀D(hashbar(D) & (neuter(C) & (neuter(D) & (owner(C) & of(C,D)))) → give(C,A,B))))))

Node 2: (reading 2)
2:∀A(criminal(A) & male(A) → ∃B(∃C(bkburger(C) & (hashbar(B) & (neuter(B) & (neuter(C) & ∀D(neuter(D) & (owner(D) & of(D,B)) → give(D,A,C)))))))

Node 3: (reading 3)
3:∀A(criminal(A) & male(A) → ∃B(hashbar(B) & (neuter(B) & ∀C(neuter(C) & (owner(C) & of(C,B)) → ∃D(bkburger(D) & (neuter(D) & give(C,A,D)))))))

Node 4: (reading 4 and 14)
4:∀A(criminal(A) & male(A) → ∀B(∀C(hashbar(C) & (neuter(B) & (neuter(C) & (owner(B) & of(B,C)))) → ∃D(bkburger(D) & (neuter(D) & give(B,A,D))))))
14:∀A(∀B(hashbar(B) & (neuter(A) & (neuter(B) & (owner(A) & of(A,B)))) → ∀C(criminal(C) & male(C) → ∃D(bkburger(D) & (neuter(D) & giveA(C,D))))))

Node 5: (reading 5)
5:∃A(bkburger(A) & (neuter(A) & ∀B(criminal(B) & male(B) → ∃C(hashbar(C) & (neuter(C) & ∀D(neuter(D) & (owner(D) & of(D,C)) → give(D,B,A)))))))

Node 6: (reading 6 and 7)
6:∃A(bkburger(A) & (neuter(A) & ∀B(criminal(B) & male(B) → ∀C(∀D(hashbar(D) & (neuter(C) & (neuter(D) & (owner(C) & of(C,D)))) → give(C,B,A))))))
7:∃A(bkburger(A) & (neuter(A) & ∀B(∀C(hashbar(C) & (neuter(B) & (neuter(C) & (owner(B) & of(B,C)))) → ∀D(criminal(D) & male(D) → give(B,D,A))))))

Node 8: (reading 8)
8:∃A(hashbar(A) & (neuter(A) & ∀B(criminal(B) & male(B) → ∃C(bkburger(C) & (neuter(C) & ∀D(neuter(D) & (owner(D) & of(D,A)) → give(D,B,C)))))))

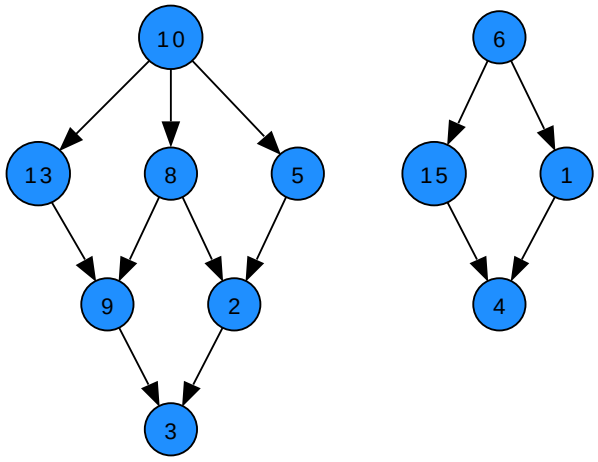
Node 9: (reading 9 and 12)
9:∃A(hashbar(A) & (neuter(A) & ∀B(criminal(B) & male(B) → ∀C(neuter(C) & (owner(C) & of(C,A)) → ∃D(bkburger(D) & (neuter(D) & give(C,B,D)))))))
12:∃A(hashbar(A) & (neuter(A) & ∀B(neuter(B) & (owner(B) & of(B,A)) → ∀C(criminal(C) & male(C) → ∃D(bkburger(D) & (neuter(D) & give(B,C,D)))))))

Node 10: (reading (10 and 11)
10:∃A(∃B(bkburger(B) & (hashbar(A) & (neuter(A) & (neuter(B) & ∀C(criminal(C) & male(C) → ∀D(neuter(D) & (owner(D) & of(D,A)) → give(D,C,B)))))))
11:∃A(∃B(bkburger(B) & (hashbar(A) & (neuter(A) & (neuter(B) & ∀C(neuter(C) & (owner(C) & of(C,A)) → ∀D(criminal(D) & male(D) → give(C,D,B)))))))

Node 13: (reading 13)
13:∃A(hashbar(A) & (neuter(A) & ∀B(neuter(B) & (owner(B) & of(B,A)) → ∃C(bkburger(C) & (neuter(C) & ∀D(criminal(D) & male(D) → give(B,D,C)))))))

Node 15: (reading 15)
15:∀A(∀B(hashbar(B) & (neuter(A) & (neuter(B) & (owner(A) & of(A,B))))) → ∃C(bkburger(C) & (neuter(C) & ∀D(criminal(D) & male(D) → give(A,D,C))))))

If we examine these readings closely and determine their logical relationships we discover that they are partitioned into two logically unrelated groups, as shown the following diagram:



The arrows in the diagram represent logical implication. Note that each group has a weakest reading (namely 10 and 6) 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 kind of inference tools discussed in Chapter 5. In Chapter 6 we show the reader exactly how this can be done.

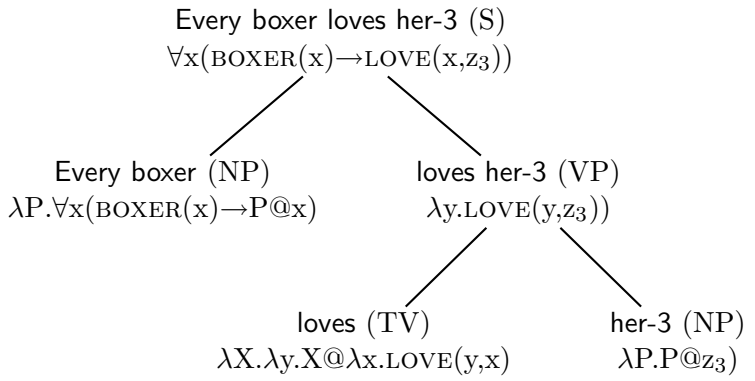
3.2 Montague's Approach

Montague semantics makes use of (indeed, is the source of) the direct method of constructing semantic representations for quantified NPs 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:

(3)



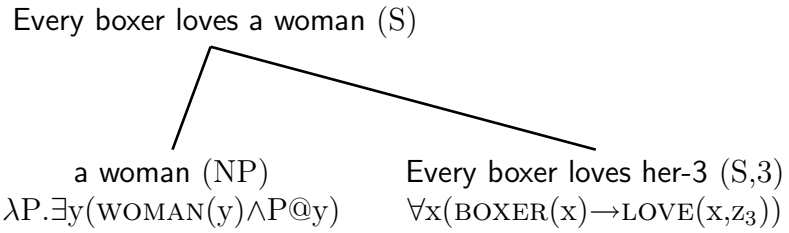
Instead of combining '**loves**' with the quantifying term '**a woman**' we have combined it with the placeholder pronoun '**her-3**'. This pronoun bears an 'index', namely the numeral '3'. The placeholder pronoun is associated with a 'semantic placeholder', namely $\lambda P.P@z_3$. As we shall see, 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 $\lambda P.P@z_3$, the semantic representation associated with the placeholder pronouns, should look familiar. Recall that the semantic representation for '**Mia**' is $\lambda P.P@MIA$. 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, we have 'raised'

both the syntactic and semantic placeholders high into the tree, and we have done so in a completely orthodox way.

Now for the next step. We want to ensure that '**a woman**' outscopes '**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:

(4)



But what's happening semantically? We know what formula we want to be assigned to the top S node (namely, the formula 2) but how can we ensure it gets there? Let's think the matter through.

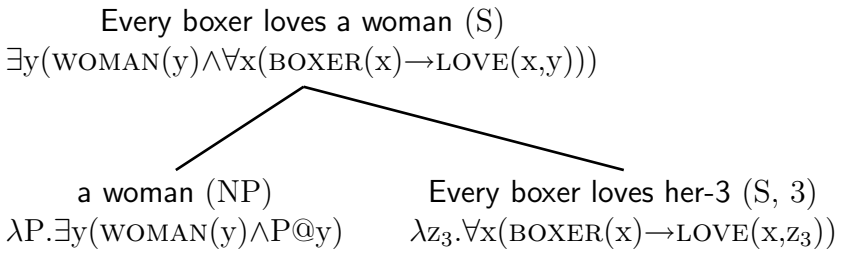
We want '**a woman**' to take wide scope over '**every boxer**' semantically. Hence we should use the semantic representation associated with '**a woman**' as the function. (To see this, simply look at the form of its semantic representation. When we apply it to an argument and perform β-conversion, 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:

(5) $\lambda z_3.\forall x(\text{BOXER}(x)\rightarrow\text{LOVE}(x,z_3))$

Why is this? Well, right at the bottom of the tree we made use of the semantic placeholder $\lambda P.P@z_3$. 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'. Lambda conversion will glue this record into its rightful place, and, as the following tree shows, everything will work out just right:

(6)



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 semantical 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 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 grammar 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 $\langle$ and $\rangle$. 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 elements (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 're-package' the information it contains 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 $\langle\phi, (\beta, j), \ldots, (\beta', k)\rangle$ is a semantic representation for a quantified NP, then the store $\langle\lambda\text{P.P@z}_i, (\phi, i), (\beta, j) \ldots (\beta', k)\rangle$, 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 $\lambda\text{P.P@z}_i$. (After all, if we decide to store ϕ away for later use, it's sensible to keep track of what its argument is.)

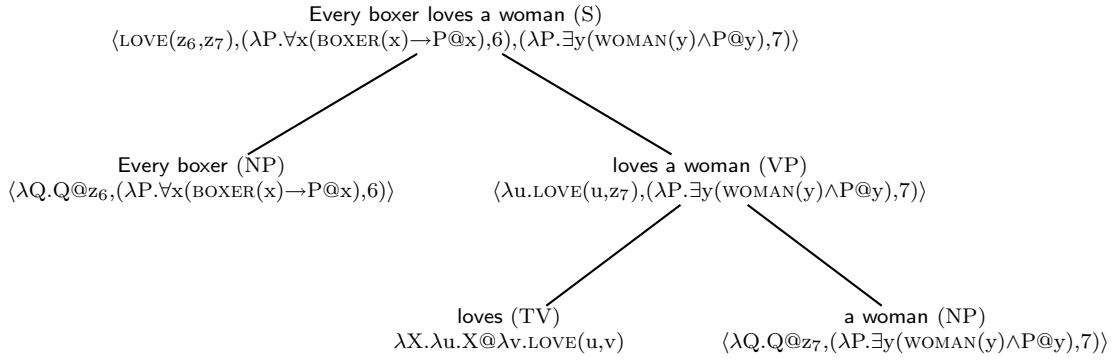
In short, from now on when we encounter a quantified NP we will be faced by a choice. We can either pass on ϕ (together with any previously stored information) straight on up the tree, or we can decide to use $\lambda\text{P.P@z}_i$ as the core representation, store the quantifier ϕ on ice for later 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 $\lambda\text{P.P@z}_i$ 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 $\langle\phi, (\beta, j), \ldots, (\beta', k)\rangle$) or use the storage rule to form

$$\langle \lambda \mathrm{P.P@z}_i, (\phi, i), (\beta, j) \ldots (\beta', k) \rangle$$

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 $\langle \lambda \mathrm{P.P@z}_m, (\lambda \mathrm{P.P@z}_i, m), (\phi, i), (\beta, j), \ldots, (\beta', k) \rangle$. 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. Here's (part of) the relevant tree.



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 λP.$\exists$y(WOMAN(y)$\wedge$P@y). In our new representations-are-stores world view, this means that the 1-place store

$$\langle \lambda \mathrm{P.}\exists \mathrm{y}(\textsc{woman}(\mathrm{y}) \wedge \mathrm{P@y}) \rangle$$

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 above tree. We've picked a brand new free variable (namely z_7), used the placeholder λQ.Q@z_7 as the first item in the store, and 'iced' λP.$\exists$y(WOMAN(y)$\wedge$P@y), first recording the fact that z_7 is the variable relevant to this expression. 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 $\langle \mathcal{F}', (\beta, j), \ldots, (\beta', k) \rangle$ and its argument

node $\mathcal{A}$ is associated with the store $\langle \mathcal{A}', (\beta'', l), \ldots, (\beta''', m)\rangle$, then store associated with the node $\mathcal{R}$ whose parts are $\mathcal{F}$ and $\mathcal{A}$ is

$$\langle \mathcal{F}'@\mathcal{A}', (\beta, j), \ldots, (\beta', k), (\beta'', l), \ldots, (\beta''', m)\rangle.$$

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 $\mathcal{F}'@\mathcal{A}'$ 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 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 (we hope!).

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:

$$\langle \lambda P.\forall x(\textsc{boxer}(x) \rightarrow P@x)@\lambda z_6.\textsc{love}(z_6,z_7),(\lambda P.\exists y(\textsc{woman}(y) \wedge P@y),7)\rangle$$

Using β-conversion, this simplifies to:

$$\langle \forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x,z_7)),\ (\lambda P.\exists y(\textsc{woman}(y) \wedge P@y),7)\rangle$$

No more β-conversions are possible, but there's still a quantifier left in store. Retrieving it produces:

$$\langle \lambda P.\exists y(\textsc{woman}(y) \wedge P@y)@\lambda z_7.\forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x,z_7))\rangle$$

The result is the reading where '**a woman**' outscopes '**every boxer**', as becomes clear if we perform two more β-conversions to obtain:

$$\langle \exists y(\textsc{woman}(y) \wedge \forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x,y))\rangle$$

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.

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)`. 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(lambda(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 have to modify our semantic macros and composition rules in such a way that they are dealing with stores. Here are some examples of the macros for storage:

```
semMacro(noun,M):-
   M = [symbol:Sym,
        sem:[lambda(X,Formula)]],
   compose(Formula,Sym,[X]).

semMacro(iv,M):-
   M = [symbol:Sym,
        sem:[lambda(X,Formula)]],
   compose(Formula,Sym,[X]).
```

Most of the composition rules work as before, the only difference being that they work with stores rather than plain lambda expressions. Sometimes a composition rule needs to append stores—coordination is a case in point:

```
combine(np:[app(app(B,A),C)|S3],[np:[A|S1],coord:[B],np:[C|S2]]):-
   appendLists(S1,S2,S3).
```

More interesting are the composition rules are those for noun phrases formed by determiners and nouns. Here we either put the quantified noun phrase on the store, or leave it where it is:

```
combine(np:[lam(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]]).
```

Two remarks should be made. First, storage should be a possibility *only* available to quantified noun phrases (in particular, we can't store other kinds of noun phrases such as proper names). Second, we have made pushing a quantifier onto the store an *optional* operation; that's why there are two composition 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 3.3.5). But we will point out that making it optional can lead to a lot of (sometimes 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. Still, as optionality is the better path to follow, we'll just have to find ways of coping

with such redundancies. For example, in this section we'll add a filter to eliminate semantic representations that are alphabetic variants of readings already produced.

One composition rule remains: the sentential one, which licenses retrieval:

```
combine(s:S,[np:[A|S1],vp:[B|S2]]):-
   appendLists(S1,S2,S3),
   sRetrieval([app(A,B)|S3],Retrieved),
   betaConvert(Retrieved,S).
```

With this last rule to hand we have all we need to create stores—but how is retrieval to be implemented?

The Prolog predicate that copes with retrieval is `sRetrieval/2`. Its first argument is a store, its second argument 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 an element from the store using the `selectFromList/3`, lambda abstracts `Sem` (the first item in the list) with respect to the retrieved variable, and applies the retrieved quantifier to it, yielding a new semantic representation that will be β-converted later. Note that the `sRetrieval` 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(bo(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` produces (via the Prolog backtracking mechanism) all the scoped representation possible.

The driver is transparent (as promised, we filter out alphabetic variants):

```
cooperStorage:-
   readLine(Sentence),
```

```
   setof(Sem,t([sem:Sem],Sentence,[]),Sems1),
   filterAlphabeticVariants(Sems1,Sems2),
   printRepresentations(Sems2).
```

The filter in defined using the `alphabeticVariants/2` predicate discussed in the previous chapter:

```
filterAlphabeticVariants(L1,L2):-
   selectFromList(X,L1,L3),
   memberList(Y,L3),
   alphabeticVariants(X,Y), !,
   filterAlphabeticVariants(L3,L2).

filterAlphabeticVariants(L,L).
```

And here's an example of our program at work:

```
?- cooperStorage.

> Every boxer loves a woman.

1 exists(A,woman(A)&forall(B,boxer(B)>love(B,A)))
2 forall(A,boxer(A)>exists(B,woman(B)&love(A,B)))

yes
```

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 grammar rules. Indeed, when you get down to it, all we really needed to do to implement Cooper storage was to define the composition rules and provide the additional predicate `sRetrieval`, which gave us the ability to manipulate our new representations in ways that plain old functional application couldn't. Cooper storage is indeed a natural and useful technique.

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 What does the store (before retrieval) for 'every piercing is done with a needle' look like? And after retrieval?

Exercise 3.3.5 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?

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 (7) and prepositional phrases in complex noun phrases (8):

(7) Every piercing that is done with a gun goes against the entire idea behind it.

(8) Mia knows every owner of a hash bar.

Both examples give rise to scope ambiguities. For example, in (8) 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 (8) more closely and find out. This is the store (as the reader should verify):

$$\langle \textsc{know}(\textsc{mia},z_2),\ (\lambda P.\forall y(\textsc{owner}(y)\wedge \textsc{of}(y,z_1)\rightarrow P@y),2),\ (\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x),1)\rangle$$

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):

$$\langle \forall y(\textsc{owner}(y)\wedge \textsc{of}(y,z_1)\rightarrow \textsc{know}(\textsc{mia},y)),\ (\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x),1)\rangle$$

Retrieving the existential quantifier then yields (again, after lambda conversion):

(9) $\langle \exists x(\textsc{hashbar}(x)\wedge\forall y(\textsc{owner}(y)\wedge \textsc{of}(y,x)\rightarrow \textsc{know}(\textsc{mia},y)))\rangle$

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:

$$\langle \exists x(\textsc{hashbar}(x)\wedge \textsc{know}(\textsc{mia},z_2)),\ (\lambda P.\forall y(\textsc{owner}(y)\wedge \textsc{of}(y,z_1)\rightarrow P@y),2)\rangle$$

Pulling the remaining quantifier off the store then yields:

(10) $\langle \forall y(\textsc{owner}(y)\wedge \textsc{of}(y,z_1) \rightarrow \exists x(\textsc{hashbar}(x)\wedge \textsc{know}(\textsc{mia},y)))\rangle$

But this is not at all 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.

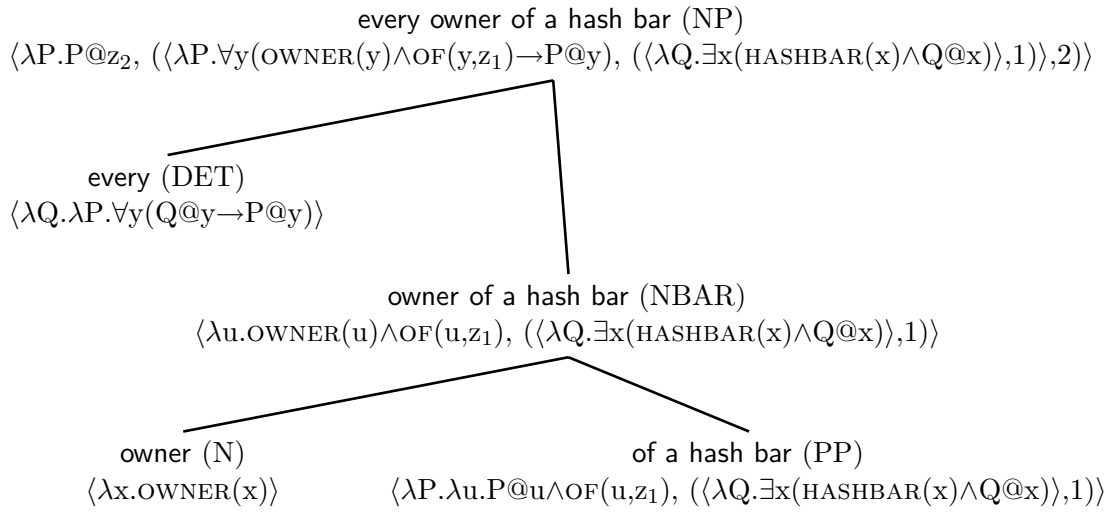
But this isn't very principled; it deals with the symptoms, not the cause. The heart of the problem is that Cooper storage rides roughshod over the hierarchical structure of NPs; we should try to deal with this head on. (Incidentally, arguably there's also an empirical problem: the free variable solution 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. As an added bonus, nesting is easier to implement than a free variable check.

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 $\langle \lambda P.P@z_i, (\langle \phi, \sigma \rangle, i) \rangle$, for some unique index i, is also an interpretation for this NP.

To see how this new storage rule works, consider how we assemble the representation associated with the complex noun phrase '<u>every owner of a hash bar</u>'.



As for the retrieval rule, it will now look like this.

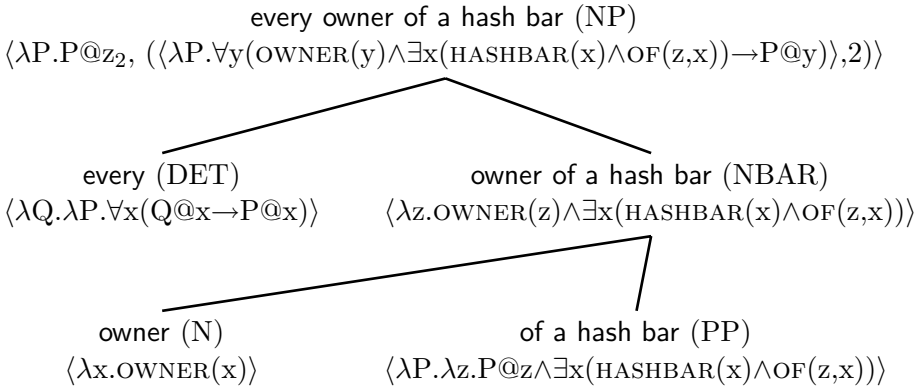
Retrieval (Keller)
Let σ, σ_1 and σ_2 be (possibly empty) sequences of binding operators. If the (nested) store $\langle \phi, \sigma_1, (\langle \beta, \sigma \rangle, i), \sigma_2 \rangle$ is an interpretation for an expression of category S, then $\langle \beta@\lambda z_i.\phi, \sigma_1, \sigma, \sigma_2 \rangle$ 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

$$\langle \text{KNOW}(\text{MIA},z_2), (\langle \lambda P.\forall y(\text{OWNER}(y)\wedge\text{OF}(y,z_1)\rightarrow P@y), (\langle \lambda Q.\exists x(\text{HASHBAR}(x)\wedge Q@x)\rangle,1)\rangle,2)\rangle$$

There is only one way to perform retrieval: first pulling of the universal quantifier, followed by the existential quantifier, resulting in (9). Since this is the only possibility, the unwanted reading (10) 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':

$$\langle \text{KNOW}(\text{MIA},z_2),(\langle \lambda P.\forall y(\text{OWNER}(y)\wedge\exists x(\text{HASHBAR}(x)\wedge\text{OF}(y,x))\rightarrow P@y)\rangle,2)\rangle$$

There is only one operator the store. Retrieving it yields the reading we want.

$$\langle \forall y(\text{OWNER}(y)\wedge\exists x(\text{HASHBAR}(x)\wedge\text{OF}(y,x))\rightarrow\text{KNOW}(\text{MIA},y))\rangle$$

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)`. 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([lambda(P,all(Y,imp(boxer(Y),app(P,X))))],X)]
```

(You should compare this with the example of a Cooper-style store we gave earlier.)

With this underlying change made, very little needs to be changed in our earlier code. Here's the new version of `sRetrieval/2` we need:

```
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 is the call to `appendLists/3` to merge stores.

Finally, we must alter the composition rules for noun phrases (these are only rules that change):

```
combine(np:[lam(P,app(P,X)),bo([app(A,B)|S],X)],[det:[A],n:[B|S]]).
combine(np:[app(A,B)|S],[det:[A],n:[B|S]]).
```

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 exists(A,hashbar(A)&forall(B,of(B,A)&owner(B)>know(mia,B)))
2 forall(A,exists(B,hashbar(B)&of(A,B))&owner(A)>know(mia,A))
```

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 (in fact, nonsensical 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 Prolog code.

But 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

(11) One criminal knows every owner of a hash bar

it doesn't allow us to insist that '**every owner**' must 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, and this is precisely what most modern underspecified representations let us do.

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 3.3.5 we saw that Cooper storage couldn't generate all the readings of 'Every boxer doesn't love a woman'. 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 as another motive for turning to the more abstract view offered by current work on underspecification.

Exercise 3.3.6 In Exercise 3.3.5 we saw that Cooper storage works better with negated sentences if storage 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 'a man likes every woman with a five-dollar shake'. You might have noticed that the number of correct readings for this example is less than the combinatorial possibilities of the quantifiers that are involved. Naively, one would expect to have $3! = 6$ readings for this example. Why is one reading excluded?

3.4 Hole Semantics

In recent eyars there has been a lot 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 as we have already remarked, 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 (and other 'funny') representations seem to have been regarded with some unease. They were (it was conceded) certainly 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

Programs for quantifier storage methods

`cooperStorage.pl`
Implementation of Cooper Storage. Defines storage of noun phrases and retrieval of the store.

`kellerStorage.pl`
Implementation of Nested Storage (Keller). Defines storage of noun phrases and retrieval of the store.

`alphaConversion.pl`
Contains the predicate for checking for alphabetic variants of readings obtained by the storage implementations.

`betaConversion.pl`
The predicates that implement β-conversion.

`englishGrammar.pl`
Our standard grammatical rules for a fragment of English.

`sentenceTestSuite.pl`
Lots of natural language examples to test our implementation.

`readLine.pl`
Contains predicates that convert a string into a list of words.

`comsemPredicates.pl`
Definitions of some auxiliary predicates.

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 semantic representations elegantly 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.

In this book, we shall explore an approach to underspecification called *hole semantics*. We have chosen hole semantics because it is the approach 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. Rferences to other approaches to underspecification can be found in the Notes at the end of the chapter.

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 semantic representation. Rather, they will be associated with abstract representations from which the desired representation can be read off. Viewed closer up, however, it is clear that hole semantics adopts a far more radical perspective on representation than storage does.

Hole semantics is essentially a constraint-based approach to semantic representation. That is, at the end of the parsing process the method delivers a set of *constraints*: any first-order representation which fulfills these constraints—which govern how the various bits and pieces produced during the parsing process 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 like this!*) and the source of much of hole semantic's power.

Let's get started. Our first task will be to break the underlying representation language (that is, first-order logic over some vocabulary) down into convenient chunks, and then we define a simple constraint language governing how these chunks can be assembled into bigger units. The resulting representations we call underspecified formulas, UFs for short.

Our semantic representation languages (SRL) are always first-order languages (with equality, let's say) over some vocabulary. Given such an SRL, the vocabulary of the underspecified representation language (URL) for the

SRL we are working with consist of the following items:

- The 2-place predicates :NOT and $\leq$.
- The 3-place predicates :IMP, :AND, :ALL, :SOME and :EQ.
- Every symbol in the SRL vocabulary is a URL constant.
- If the SRL vocabulary contains any constant symbols, then CONSTANT is a 2-place URL symbol. **CHANGE THIS**
- If the SRL vocabulary contains a relation symbol PRED of arity n, then :PRED is a $n+1$-place URL symbol.

The URL is a three sorted language. It contains variables called *holes* (written h, h′, h_1, h_2, and so on), variables called *labels* (written l, l′, l_1, l_2, and so on) and variables called *metavariables* (written v, v′, v_1, v_2, and so on). A *node* is a hole or a label. A meta-term is meta-variables or a URL constant (that is, a symbol from the SRL vocabulary).

We are not interested in all the first-order formulas that can be constructed over this vocabulary—we're only interested in the existentially closed conjunctive formulas. More precisely, we're interested in what we call UFs. Basic UFs are defined as follows:

1. If l is a label, and h is a hole, then l$\leq$h is a basic UF;
2. If l is a label, and n and n′ are nodes, then l:NOT(n), l:IMP(n,n′), l:AND(n,n′), l:OR(n,n′) are basic UFs;
3. If l is a label, t and t′ are terms, then l:EQ(t,t′) is a basic UF;
4. If l is a label, S is a symbol in the SRL language with arity n, and $t_1 \ldots t_n$ are terms, then l:S,t_1,...,t_n) is a basic UF.
5. If l is a label, v a metavariable, and n a hole or label, then l:SOME(v,n) and l:ALL(v,n) are basic UFs.
6. Nothing else is a basic UF.

Now for arbitrary UFs:

1. All basic UFs are UFs;

2. If ϕ is a UF, and n is a node then $\exists n\phi$ is a UF;

3. If ϕ is a UF, and v is a meta-variable then $\exists v\phi$ is a UF;

4. If ϕ and ψ are UFs,then $(\phi \wedge \psi)$ is a UF;

5. Nothing else is a UF.

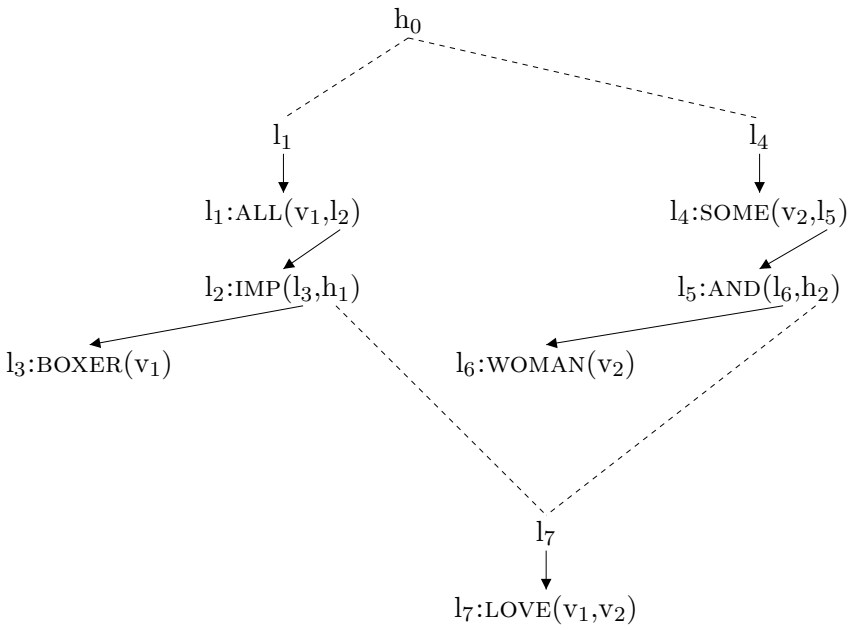
Here's an example. As we have already mentioned, sentences are going to be associated with an UF. Here's the UF associated with 'Every boxer loves a woman':

(12) $\exists l_1 \exists l_2 \exists v_1 (l_1{:}\textsc{all}(v_1,l_2) \wedge \exists l_3 \exists h_1 (l_2{:}\textsc{imp}(l_3,h_1) \wedge l_3{:}\textsc{boxer}(v_1) \wedge$
$\exists l_4 \exists l_5 \exists v_2 (l_4{:}\textsc{some}(v_2,l_5) \wedge \exists l_6 \exists h_2 (l_5{:}\textsc{and}(l_6,h_2) \wedge l_6{:}\textsc{woman}(v_2) \wedge$
$\exists l_7 (l_7{:}\textsc{love}(v_1,v_2) \wedge l_7 \leq h_1 \wedge l_7 \leq h_2 \wedge \exists h_0 (l_1 \leq h_0 \wedge l_4 \leq h_0))))))$

Simplify this underspecified formula by raising all quantifiers to the front:

$\exists h_0 \exists h_1 \exists h_2 \exists l_1 \exists l_2 \exists l_3 \exists l_4 \exists l_5 \exists l_6 \exists l_7 \exists v_1 \exists v_2 (l_1{:}\textsc{all}(v_1,l_2) \wedge l_2{:}\textsc{imp}(l_3,h_1)$
$\wedge\ l_3{:}\textsc{boxer}(v_1) \wedge l_4{:}\textsc{some}(v_2,l_5) \wedge l_5{:}\textsc{and}(l_6,h_2) \wedge l_6{:}\textsc{woman}(v_2)$
$\wedge\ l_7{:}\textsc{love}(v_1,v_2) \wedge l_7 \leq h_1 \wedge l_7 \leq h_2 \wedge l_1 \leq h_0 \wedge l_4 \leq h_0)$

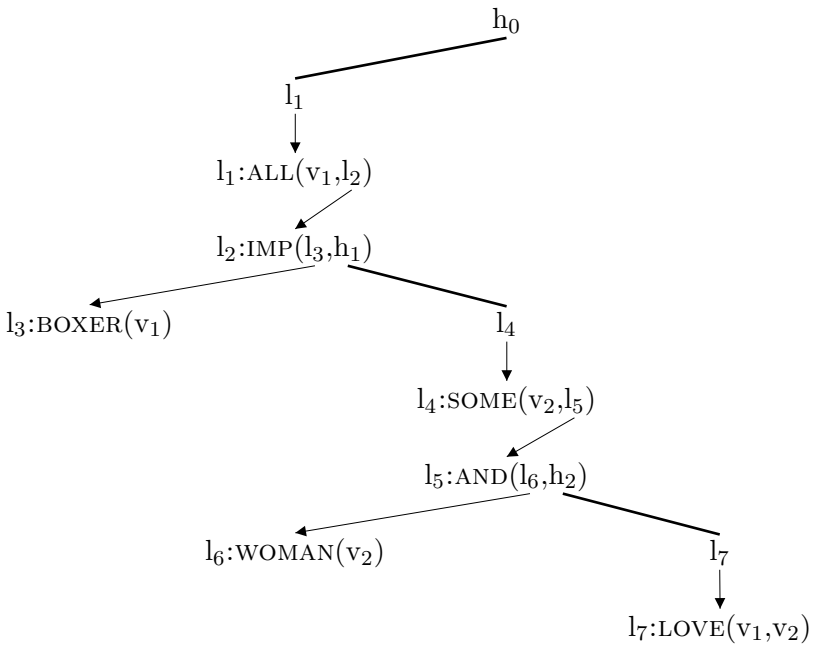
But what does this UF mean? Essentially it tells us how we are permitted to plug together the labeled formulas to build a first-order formula. The hole h_0 is a variable that stands for the first-order formula that we will eventually build. This formula will be built by plugging together three labeled formulas given in the middle component of the UF. Plugging together simply means substituting one of these three formulas in another. (Incidentally, when we carry out such a substitution we only substitute the formula, not its label as well. The label is simply an identifier that enables us to state the right hand side constraints succinctly). Any plugging will do—*as long as it satisfies all constraints given*. These constraints are essentially constraints on *subformulas*, and are best thought of visually. Each dashed line in the following diagram corresponds to a $\leq$ expression in an obvious way:



The constraints are set up in such a way that the formula l_7: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.

Now it's time to say a little more about *resolving* these ambiguous representations. The holes underspecify scope, and in order to give us non-ambiguous interpretations, 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. Obviously, no label can be plugged into two different holes at the same time. Therefore, a so-called *plugging* is a one-to-one correspondence from holes to labels (i.e., a bijective function, with the set of holes as domain and the set of labels as codomain). A plugging for a proper UF is admissible if the instantiations of the holes with labels result in a representation in which there is no contradiction spelled out by the constraints.

There are two pluggings, 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 $P_2(h_0)=l_4$, $P_2(h_1)=l_7$, $P_2(h_2)=l_1$. Plugging P_1 interprets (12) as giving the universal quantifier wide scope, out-scoping the existential quantifier. The corresponding formula is:

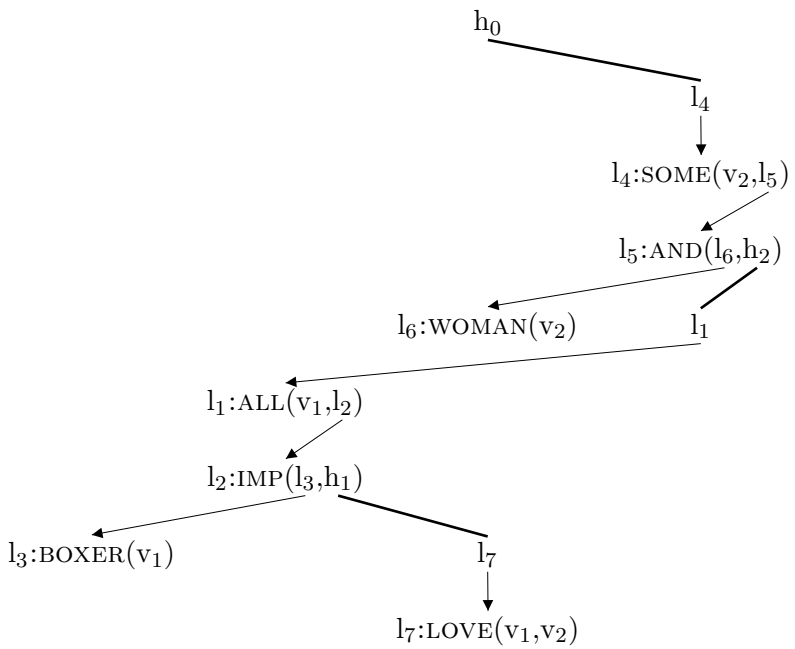


This tree structure obviously describes the formula:

$$\forall x(\text{BOXER}(x) \rightarrow \exists y(\text{WOMAN}(y) \wedge \text{LOVE}(x,y)))$$

This formula is true in a model where all boxers love a woman, but not necessarily the same woman.

Plugging P_2 interprets (12) as follows:



Here the existential quantifier out-scopes the universal quantifier. In a model where there is some woman that is loved by all boxers, this interpretation denotes truth. The corresponding formula in predicate logic is:

$$\exists y(\text{WOMAN}(y) \wedge \forall x(\text{BOXER}(x) \rightarrow \text{LOVE}(x,y)))$$

Translation of a UF and a plugging to FOL:

$(h,P)^{\text{uf2fol}} = (P(h))^{\text{uf2fol}}$ iff h is a hole
$(l,P)^{\text{uf2fol}} = \exists v(n,P)^{\text{uf2fol}}$ iff l:SOME(v,n)
$(l,P)^{\text{uf2fol}} = \forall v(n,P)^{\text{uf2fol}}$ iff l:ALL(v,n)
$(l,P)^{\text{uf2fol}} = ((n,P)^{\text{uf2fol}} \wedge (n',P)^{\text{uf2fol}})$ iff l:AND(n,n′)
$(l,P)^{\text{uf2fol}} = C(v)$ iff l:C(v)

and so on....

Underspecified Representations in Prolog

Explain new Prolog notation required for UFs.

In addition, to distinguish holes and labels from each other (and from other meta-variables), we will use `hole(H)` to designate a hole h and `label(L)` for a label l.

Here is an example UF for ''.

$\exists h_0\ \exists l_1\ \exists h_1\ \exists l_2\ \exists l_3\ \exists l_4\ \exists v_1(l_3{:}\textsc{some}(v_1,l_4) \wedge l_4{:}\textsc{and}(l_2,h_1) \wedge l_1{\leq}h_1 \wedge l_3{\leq}h_0 \wedge l_2{:}\textsc{boxer}(v_1) \wedge l_2{\leq}h_0 \wedge l_1{:}\textsc{collapse}(v_1) \wedge l_1{\leq}h_0)$

And here is its Prolog equivalent:

```
ex(A,and(hole(A),ex(B,and(label(B),ex(C,and(hole(C),ex(D,and(label(D),
     ex(E,and(label(E),ex(F,and(label(F),ex(G,and(E:ex(G,F),and(F:and(D,C),
        and(leq(B,C),and(leq(E,A),and(and(D:boxer(G),leq(D,A)),
            and(B:collapse(G),leq(B,A))))))))))))))))))))
```

Note, in the Prolog notation too there is no confusion possible between meta- and object level of representation.

Computing Underspecified Formulas

Building UFs is done with the tools we're already familiar way: using lambdas and β-conversion. For UFs, we are going to introduce lambdas for holes, labels and meta-variables as well. Use of colon.

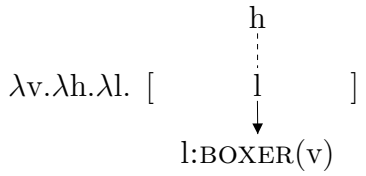
The way we construct UFs is basically an extension to our normal systematic way of constructing representations. To keep track of the holes and labels, we will decorate *each* UF-representation with two additional lambdas: one binding the *top hole*, and one binding the *main label*. The top hole is the hole that closes the current scope domain. The main label is the label within the current scope domain that is out-scoped by all elements within its scope domain.

Abstracting away from these two concepts, it is straightforward to combine UFs (incidentally, macro definitions might get complicated—drawing a picture often eases matters).

Start with a simple example. As notational convention we use indexs for existentially quantified variables, and no indexes for lambda-bound variables. Take the noun 'boxer' for instance:

$\lambda v.\lambda h.\lambda l.(\textsc{boxer}(l,v) \wedge l{\leq}h).$

Drawn as a tree:

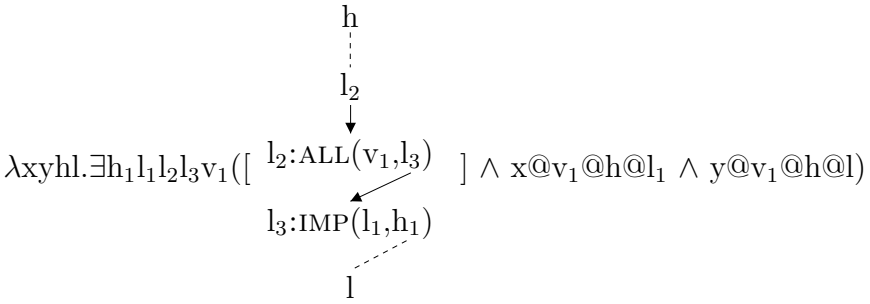


Determiner 'every':

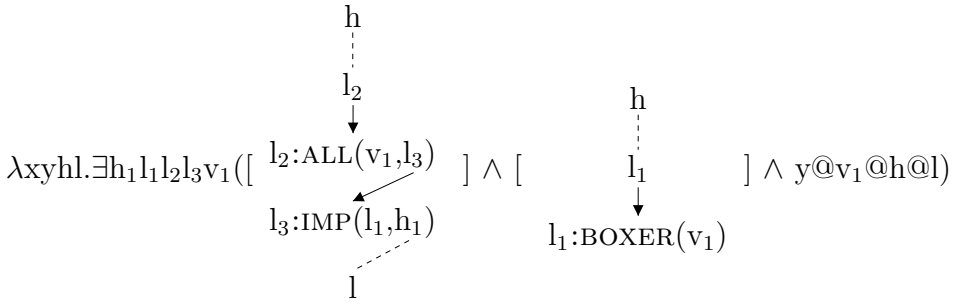
$\lambda x.\lambda y.\lambda h.\lambda l.\exists h_1 \exists l_1 \exists l_2 \exists l_3 \exists v_1 (l_2{:}\textsc{all}(v_1,l_3) \wedge l_3{:}\textsc{imp}(l_1,h_1) \wedge l \leq h_1 \wedge l_2 \leq h \wedge x@v_1@h@l_1 \wedge y@v_1@h@l)$.

The two additional lambdas binding h and l define the scope domain of '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—these associate the restriction label of the quantifier with the label of the noun representation, and the top and hole label of the verb phrase argument.

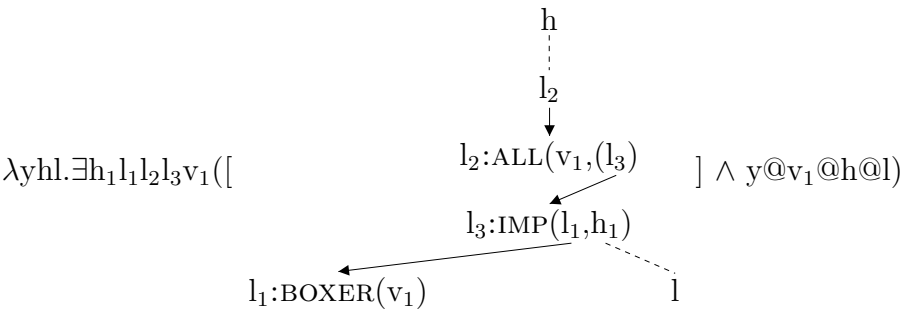
Drawn as a tree:



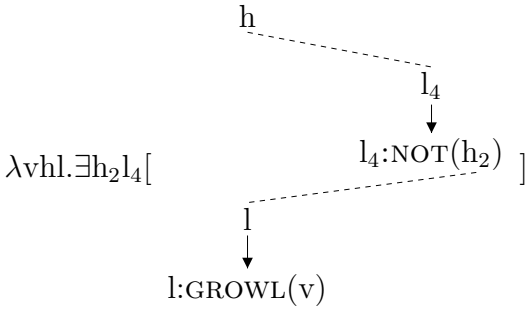
Drawn as two trees ('every boxer'):



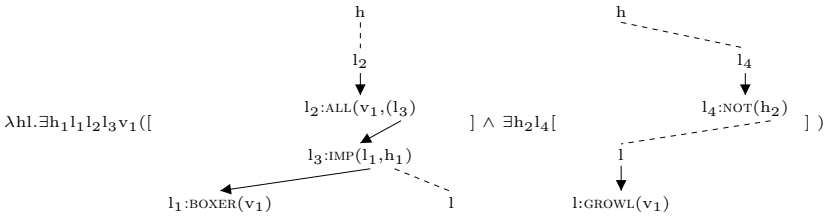
Next step. Drawn as a tree ('every boxer'):



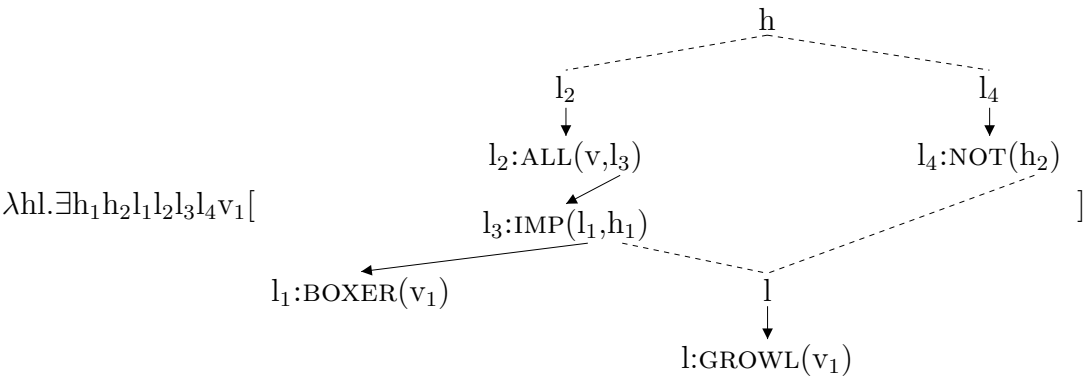
Drawn as a tree, the verb phrase ('does not growl'):



Drawn as two trees ('every boxer does not growl'):



Drawn as a tree ('every boxer does not growl'):



In Prolog, nouns are dealt with by the following macro:

```
semMacro(noun,M):-
   M = [symbol:Sym,
        sem:lam(X,lam(H,lam(L,and(pred1(L,Sym,X),leq(L,H)))))].
```

Let's introduce the semantic macro for the determiner 'every':

```
semMacro(det,M):-
   M = [type:uni,
        sem:lam(N,lam(V,lam(H,lam(L,ex(H1,ex(L1,ex(L2,ex(L3,ex(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))))))))))))))))))].
```

Proper names, on the other hand do not introduce holes, and therefore have no $\leq$-constraints. A new top hole `H` and main label `L` are introduced and applied to its argument `V`.

```
semMacro(pn,M):-
   M = [symbol:Sym,
        gender:_,
        sem:lam(V,lam(H,lam(L,app(app(app(V,Sym),H),L))))].
```

Verbs introduce the 'top' hole and a leq-constraint between itself and the top. This is of importance for sentences where no quantifiers (or other scope-introducing elements) appear, as it establishes a clean link from the top element to the main label, that of the verb. Consider the entries for intransitive and transitive verbs for example:

```
semMacro(iv,M):-
   M = [symbol:Sym,
        sem:lam(X,lam(H,lam(L,and(pred1(L,Sym,X),
                                  leq(L,H))))),].

semMacro(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)))))))),].
```

Underspecification allows us to be very flexible. We can use one and the same technique for both quantifiers and negation. Here is the semantic macro for auxiliary verbs:

```
semMacro(av,M):-
   M = [pol:neg,
        sem:lam(V,lam(X,lam(H,lam(L,ex(S,ex(N,and(hole(S),
                and(label(N),and(not(N,S),and(leq(N,H),and(leq(L,S),
                    app(app(app(V,X),H),L)))))))))))),];
```

Similar macros are defined for the quantified noun phrases, copula, relative pronouns, adjectives, prepositions, and coordination. We won't discuss them all in detail here.

The composition rules are mostly like that of the implementation of the lambda calculus. There are two exceptions: complex sentences (see exercise), and the following rule:

```
combine(t:U,[s:S]):-
   betaConvert(ex(T,and(hole(T),ex(L,and(label(L),
                    app(app(S,T),L))))),U).
```

This is the rule that closes off the outermost scoping domain. Here `S`, the representation for the sentence, is of the form $\lambda h.\lambda l.\phi$. Now the above rule, being of the form $\exists h_0 \exists l_1 S@h_0@l_1$, results in $\exists h_0 \exists l_1 \lambda h.\lambda l.\phi@h_0@l_1$ which can be reduced to a UF without any occurrences of lambda-bound variables.

The Plugging Algorithm

The steps: 1. skolemize variables—this will be easier for the implementation because we can make maximal use of unification. (numbervars/3) 2. break

down UF and assert to database. 3. calculate a plugging 4. compute top and construct the formula.

Break down the UF and assert it to the Prolog database. We therefore introduce the following dynamic predicates (predicates which definition can change during runtime).

```
:- dynamic plug/2, leq/2, hole/1, label/1.
:- dynamic ex/3, all/3, que/4.
:- dynamic not/2, or/3, imp/3, and/3.
:- dynamic pred1/3, pred2/4, eq/3.
```

Assert UF to Prolog database. Remember that UFs are existentially quantified conjunctive formulas, so we need only to recurse on those. All other expressions are asserted to the database.

```
assertUF(ex(_,F)):-
    assertUF(F).

assertUF(and(F1,F2)):-
    assertUF(F1),
    assertUF(F2).

assertUF(F):-
    \+ F=and(_,_),
    \+ F=ex(_,_),
    assert(F).
```

Once a UF is broken down into bits and asserted to the database, 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):- ex(A,_,B).
```

In addition, we every plug is also a parent.

```
parent(A,B):- plug(A,B).
```

We use the parent and the leq of the UF to define dominance. Transitive closure of dominance:

```
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).
```

Once we have defined dominance, we can also define the top of a UF:

```
top(X):-
   dom(X,_),
   \+ dom(_,X), !.
```

Check whether a plugging is proper. First retract previous attempt to plugging from the database, then assert the new pluggings. Check whether the plugging causes cycles, and whether two nodes with the same parent dominate another node.

```
properPlugging(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)).
```

Plug recursively all holes with the available labels.

```
plugHoles([],_,Plugs):-
   properPlugging(Plugs).

plugHoles([H|Holes],Labels1,Plugs):-
   properPlugging(Plugs),
   selectFromList(L,Labels1,Labels2),
   plugHoles(Holes,Labels2,[plug(H,L)|Plugs]).
```

Finally, given a plugging (still asserted to the database!), we need to reconstruct the formula. This is done by `uf2fol/2`, which recursively converts a UF into an ordinary first-order formula with respect to a plugging.

If a UF-term is a hole, it inspects the plugging for that hole and continues converting the label:

```
uf2fol(H,F):-
   hole(H),
   plug(H,L),
   uf2fol(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:

```
uf2fol(L,all(X,F)):-
   all(L,X,H),
   uf2fol(H,F).
```

For the booleans, again this is straightforward. Consider for instance the definition for plugging an implication:

```
uf2fol(L,imp(F1,F2)):-
   imp(L,H1,H2),
   uf2fol(H1,F1),
   uf2fol(H2,F2).
```

Finally, basic formulas:

```
uf2fol(L,F):-
   pred1(L,Symbol,Arg),
   compose(F,Symbol,[Arg]).

uf2fol(L,F):-
   pred2(L,Symbol,Arg1,Arg2),
   compose(F,Symbol,[Arg1,Arg2]).
```

The main wrapper.

```
plugUF(UF,Sem):-
   numbervars(UF,0,_),  % assuming distinct variables
   initUF,
   assertUF(UF),
   findall(H,hole(H),Holes),
   findall(L,(label(L),\+ parent(_,L)),Labels),
   top(Top),
   plugHoles(Holes,Labels,[]),
   uf2fol(Top,Sem).
```

As usual we implement a driver that grasps all the important predicates together. This one combines the plugging algorithm and the construction of UFs, displays the UF on the current output device as well as all the readings it has.

```
holeSemantics:-
   readLine(Sentence),
   parse(Sentence,UF),
   printRepresentations([UF]),
   setof(Sem,plugUF(UF,Sem),Sems),
   printRepresentations(Sems).
```

Here is an example session, showing the UF for 'every woman does not snort' and the two obtained readings.

```
?- holeSemantics.

> Every woman does not snort
```

```
1 ex(A,and(hole(A),ex(B,and(label(B),ex(C,ex(D,ex(E,ex(F,ex(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)),ex(H,ex(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))))

yes
```

Programs for Hole Semantics

`holeSemantics.pl`
Main file of Predicate Logic Unplugged.

`pluggingAlgorithm.pl`
The plugging algorithm for hole semantics.

`sentenceTestSuite.pl`
Lots of natural language examples to test our implementation.

`readLine.pl`
Contains predicates that convert a string into a list of words.

`comsemPredicates.pl`
Definitions of some auxiliary predicates.

Exercise 3.4.1 The Prolog definition of the transitive closure of the dominance relation is perhaps, with the inclusion of `memberList/3` and an accumulator list of previously unaccounted immediate dominance relations, slightly more complicated than you had 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(a,b). parent(b,c). parent(c,a). parent(d,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.

Evaluating Underspecification

So far we have formalised semantic underspecified representations and are able to deal with (quantifier) scope ambiguities. But how does this proposal relate to earlier accounts of scope ambiguities? Are there any benefits in using the new streamlined underspecified representation rather than stick to the old fashioned storage methods? The basic answer is that underspecification languages are more powerful: they exhibit far more descriptive potential than storage methods. To make this statement more clear, let's consider a concrete example sentence including three quantifiers:

(13) One criminal knows every owner of a hash bar.

With the tools developed in this chapter it is perfectly possible to represent the relative scopes of the three quantifiers fully underspecified. Moreover, it is possible to partially specify relative scopes. For some (linguistic) reason, one might wish to express that '*one criminal*' out-scopes '*every owner*' and '*a hash bar*', but that the relative scopes of the latter two quantifying constructs does not matter. We need not add new machinery to our formalism: we can use the $\leq$-constraint to state such requirements.

How do other approaches to quantifier scope fare with such requirements? Storage techniques [**?**, **?**], to begin with, are not able to express such constraints. A store is a sequence of the core semantic representation and a list of the binding operators (the quantifiers) which scope is not fixed yet. In the process of constructing a store, quantified noun phrases can either enter the store or combine with the core semantic representation (and form a new core semantic representation, by means of in-situ quantification). At the end of the parse, the different ways in which the quantifiers are retrieved from the store and combined with the core semantic representation result in different readings (the last quantifier which is pulled off the store gets widest scope).

Hence, to give a quantifier obligatory wide scope, the only way to pursue is to put it on the store, while performing in-situ application for the remaining quantifiers. But this presumes that the scope order of the remaining quantifiers is fixed!

Chapter 4

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 user extends 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.

4.1 Baby Curt

Baby Curt 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`

program from Chapter 3. Here is an example of a dialogue with Baby Curt. The text preceded by the > symbol are 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. (Reserved commands are not interpreted by Curt.) One of these is the commands `readings` which has the following effect:

```
> readings

1 (love(vincent,mia) &
     forall A (woman(A) > exists B (boxer(B) & know(A,B))))
2 (love(vincent,mia) &
     exists A (boxer(A) & forall 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.

Normally Curt takes the first of the readings in its memory and combines that with the new sentences that are entered; it simply forgets about any other interpretations. But there is another reserved command that allows us to force Curt to choose a specific reading. This is the `select` command:

```
> select 2

> readings

1 (love(vincent,mia) &
     exists A (boxer(A) & forall B (woman(B) > know(B,A))))
```

Here we see that Curt has kept the narrow scope reading for a boxer in its memory. 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 readings 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 do so using two *dynamic* Prolog predicates. The first, `readings/1`, will be used to store the interpretations of the sentence 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` and `retract` 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, which will be unified with the Prolog variable `Input`. Next `curtUpdate/3` takes the newly entered sentence and updates it with respect to the 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 your 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 straightforwardly coded:

```
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 the reserved 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 whatever we entered doesn't match any of the reserved commands, Curt attempts to parse the input using `kellerStorage/3`. 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),
   readings(OldReadings),
   combine(Readings,OldReadings,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-commands. 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-commands 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 concretely realized 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 the Exercise 4.1.2 to find out how they work. But `combine/3`, which is used by `curtUpdate/3` to combine the readings of the new sentence with the readings of the previous discourse, is worth looking at right away:

```
combine(New,[],New).

combine(Readings,[Old|_],Updated):-
   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 4.1.1 Load Baby Curt and enter some sentences. Try all the reserved commands.

Exercise 4.1.2 Inspect the file `curtPredicates.pl` and check how the predicates `updateReadings/1`, `updateHistory/1`, `clearHistory/0`, and `selectReadings/3` are defined.

Programs for Baby Curt

`babyCurt.pl`
Loading this file loads everything needed to run Baby Curt.

`curtPredicates.pl`
Auxiliary predicates used by the Curt family.

4.2 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 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 in 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

$$\text{Discourse-So-Far} \models \neg\phi$$

If the theorem can prove this, then the latest sentence is inconsistent with the previous discourse. Now, note the message from the free-variable theorem tableau prover in the second line of our example: our 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` 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),
   readings(OldReadings),
   combine(Readings,OldReadings,Combined),
   consistentReadings(Combined-NewReadings,Moves),
   updateReadings(NewReadings),
   updateModels(Models).
```

The consistent readings are selected as follows. For each reading in Curt's memory, Curt tries to prove whether it is inconsistent using the predicate `consistent/2`. All readings that are consistent are stored in the variable `Consistent`. If there are no consistent readings, then reply-move is unified with `contradiction`, otherwise the reply-move is set to `accept`.

```
consistentReadings(Readings-Consistent,[Move]):-
   findall(Reading,
           (
              memberList(Reading,Readings),
              consistent(Reading)
           ),
          Consistent),
   (
       Consistent=[],
       Move=contradiction
   ;
       \+ Consistent=[],
       Move=accept
   ).
```

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(Formula):-
   prove(not(Formula),75), !,
   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 realization 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 4.2.1 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 4.2.2 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.

Programs for Rugrat Curt

`rugratCurt.pl`
Loading this file loads everything needed to run Rugrat Curt.

4.3 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 copula constructions) 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

```
exists A (man(A) & vincent = A)
```

Note the way the equality symbol is used to predicate the 'is a man' property of 'Vincent'. But the free-variable tableau system of Chapter 5 doesn't handle equality, and 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 very difficult task computationally (indeed, general first-order reasoning is undecidable, and even propositional reasoning, which looks so simple, is NP-complete). 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 via the predicate `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, but we urge the reader to experiment with other combinations—the results are often fascinating.

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 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 inconsistencies 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 builder, 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 *positive check for consistency*. If the model builder can show that

$$\text{Discourse-So-Far} \cup \{\phi\}$$

has a model, 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 better all-round performance. Bearing this in mind, let's take a look at Clever Curt 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 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 theorem prover and 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` which is now a three-place predicate (because we want to keep track of the models returned by the model builder):

```
consistentReadings(Readings-Consistent,Models,[Move]):-
        findall((Reading,Model),
                (
                    memberList(Reading,Readings),
                    consistent(Reading,Model)
                ),
                Interpretations),
        (
            Interpretations=[],
            Move=contradiction
        ;
            \+ Interpretations=[],
            Move=accept
        ),
        findall(R,memberList((R,_),Interpretations),Consistent),
        findall(M,memberList((_,M),Interpretations),Models).
```

The predicate `consistent/2` is defined as follows. It sets the domain size, and then makes a call to `callTPandMB/4` (recall Chapter 5), prints the result and succeeds only if the model returned is non-empty.

```
consistent(Formula,Model):-
   DomainSize=15,
   callTPandMB(Formula,DomainSize,Model,Engine),
   format('~nMessage (consistency checking): ~p found a result.',[Engine]),
   \+ Model=model([],[]).
```

And that's pretty much it. Once again, we suggest the reader thinks hard about the following exercises before moving on.

Exercise 4.3.1 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 4.3.2 In Exercise 4.2.2 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?

Exercise 4.3.3 Try playing with Clever Curt when the chosen theorem prover is Otter. Can you find examples for which the behaviour of Curt is significantly different when this prover is chosen?

Programs for Clever Curt

`cleverCurt.pl`
Loading this file loads everything needed to run Clever Curt.

4.4 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 informativeness 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 realize this. Now, in many 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 sensitize Curt to this distinction?

We learnt two important answers in Chapter 5 (a different answer is explored in Exercise 4.4.2). First, we learnt that theorem provers provide us with a *negative check for informativeness*. If a theorem prover can show that

$$\text{Discourse-So-Far} \models \phi$$

then the latest sentence is not informative with respect to the previous discourse. We also learnt that model builder provide us with a *positive check for informativeness*. If a model builder can show that

$$\text{Discourse-So-Far} \cup \{\neg\phi\}$$

has a model then the latest sentence is informative with respect to with the previous discourse.

Both checks are built into Sensitive Curt. 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 (informativeness): 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/3`:

```
curtUpdate(Input,Moves,run):-
   kellerStorage(Input,Readings), !,
   updateHistory(Input),
   readings(OldReadings),
   combine(Readings,OldReadings,Combined),
   consistentReadings(Combined-NewReadings,Models,TempMoves),
   informativeReadings(NewReadings,TempMoves,Moves),
   updateReadings(NewReadings),
   updateModels(Models).
```

The predicate, `informativeReadings/3`, is defined as follows:

```
informativeReadings([],Moves,Moves):- !.

informativeReadings(Readings,_,[obvious]):-
   memberList(and(Old,New),Readings),
   \+ informative(Old,New), !.

informativeReadings(_,Moves,Moves).
```

There are three clauses for 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 informativeness. The second clause checks if there are no informative readings among the set of consistent readings. It does so with the help of the predicate `informative/2`. This returns the reply-move `obvious` if it succeeds. Otherwise, in the third clause, the reply-moves keeps its old value.

So, how is `informative/2` implemented? As follows:

```
informative(Old,New):-
   callTP(imp(Old,New),Proof,Engine),
   (
    Proof=proof,
    format('~nMessage (informativeness checking): ~p found a result.',[Engine])
    fail
   ;
    \+ Proof=proof
   ).
```

Finally, because we have added a new reply-move, we need to add a clause for the realization 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 in to this distinction (or at least, tuned to them up to the limits imposed by theorem prover and 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:

(1)
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:

(2)
If Mia snorts, then Vincent smokes.
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:

(3)
A woman loves every man.
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 reading are intended. We leave the reader to sort out the details in the following exercise.

Exercise 4.4.1 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 a reserved command `select` to allow us to choose which reading is carried through.)

Exercise 4.4.2 In the text we used the approach discussed in Chapter 5 (namely running a theorem prover and model builder in parallel) to determine informativeness. But it is interesting to think about other methods of establishing informativeness.

Clever Curt carries out consistency checks before it checks for informativeness, 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 informativeness. What are the advantages and disadvantages of this approach to informativeness checking compared to the approach in the text?

Programs for Sensitive Curt

`sensitiveCurt.pl`
Loading this file loads everything needed to run Sensitive Curt.

4.5 Scrupulous Curt

Although Sensitive Curt can perform both consistency and informativeness 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 exists A (boxer(A) & exists B (woman(B) & love(A,B)))
2 exists A (woman(A) & exists 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 can move the innermost quantifier to the outside without changing the meaning:

```
1 exists A (exists B (boxer(A) & (woman(B) & love(A,B))))
2 exists A (exists 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 exists A (exists 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 exists B (exists A (woman(B) & (boxer(A) & love(A,B))))
```

But this is α-equivalent to

```
2 exists A (exists 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 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 $\psi \rightarrow \pi$ (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.

Curt: OK.

> readings

1 exists A (boxer(A) & exists B (woman(B) & love(A,B)))
2 exists A (woman(A) & exists B (boxer(B) & love(B,A)))

> summary

Message (eliminating equivalent readings): there are 2 readings:
1 exists A (boxer(A) & exists B (woman(B) & love(A,B)))
2 exists A (woman(A) & exists B (boxer(B) & love(B,A)))
Readings 1 and 2 are equivalent (bliksem).

> readings

1 exists A (woman(A) & exists B (boxer(B) & love(B,A)))
```

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`:

```
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,1,N,Numbered),
   format('~nMessage (eliminating equivalent readings): there are ~p readings:',[N]),
   printRepresentations(Readings),
   elimEquivReadings(Numbered,[],Unique).
```

Numbering the readings is easy:

```
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:

```
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 (~p).~n',[N1,N2,Engine]),
      elimEquivReadings(Readings,Diff,Unique)
   ;
      format('Readings ~p and ~p are probably 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. Or is 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): there are 2 readings:
1 forall A (boxer(A) > exists B (woman(B) & love(A, B)))
2 exists A (woman(A) & forall B (boxer(B) > love(B, A)))
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 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 explore this idea in Exercise 4.5.4

Exercise 4.5.1 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 4.5.2 [hard] 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 favors 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 4.5.3 [hard] 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 $\phi_1, \ldots, \phi_n$, 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 4.5.4 Extend `eliminateEquivalentReadings/3` by calling a model builder in parallel to explicitly demonstrate the non-equivalence of readings.

Programs for Scrupulous Curt

`scrupulousCurt.pl`
Loading this file loads everything needed to run Scrupulous Curt.

4.6 Knowledgeable Curt

Curt can now do so many clever things (perform consistency and informativeness 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:

```
> Mia is a woman

Message (consistency checking): mace found a result.

Curt: OK.

> Mia is a plant
```

```
Message (consistency checking): mace found a result.

Curt: OK.
```

A human being, of course, immediately sees the conflict between the claims that Mia 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 Chapter 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

$$\mathsf{Discourse\text{-}So\text{-}Far} \models \neg\phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\mathsf{Discourse\text{-}So\text{-}Far} \cup \{\phi\}$$

has a model (the positive test). As for informativeness checking, this boils down to using a theorem prover to test whether

$$\mathsf{Discourse\text{-}So\text{-}Far} \models \phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\mathsf{Discourse\text{-}So\text{-}Far} \cup \{\neg\phi\}$$

has a model (the 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 two kinds: *lexical knowledge* and *world knowledge*. Then consistency checking boils down to using a theorem prover to test whether

$$\mathsf{Lexical\text{-}Knowledge} \cup \mathsf{World\text{-}Knowledge} \cup \mathsf{Discourse\text{-}So\text{-}Far} \models \neg\phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\mathsf{Lexical\text{-}Knowledge} \cup \mathsf{World\text{-}Knowledge} \cup \mathsf{Discourse\text{-}So\text{-}Far} \cup \{\phi\}$$

has a model (the positive test). As for informativeness checking, this boils down to using a theorem prover to test whether

$$\mathsf{Lexical\text{-}Knowledge} \cup \mathsf{World\text{-}Knowledge} \cup \mathsf{Discourse\text{-}So\text{-}Far} \models \phi$$

(the negative test) and simultaneously using the model builder to check whether

$$\mathsf{Lexical\text{-}Knowledge} \cup \mathsf{World\text{-}Knowledge} \cup \mathsf{Discourse\text{-}So\text{-}Far} \cup \{\neg\phi\}$$

has a model (the positive test).

Let's look at an example. Knowledgeable Curt has a small fund of lexical and world knowledge, and among the things it knows is that woman 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:

```
> Mia is a woman

Message (consistency checking): mace found a result.

Curt: OK.

> Mia 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 a 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.
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 (forall A (concrete(A) >  ~ abstract(A)) &
  (forall B (entity(B) > concrete(B)) &
  (forall C (entity(C) > thing(C)) &
  (forall D (living(D) >  ~ nonliving(D)) &
  (forall E (male(E) >  ~ female(E)) &
  (forall F (organism(F) > living(F)) &
  (forall G (organism(G) > entity(G)) &
  (forall H (animate(H) >  ~ inanimate(H)) &
  (forall I (human(I) >  ~ nonhuman(I)) &
  (forall J (person(J) > human(J)) &
  (forall K (person(K) > animate(K)) &
  (forall L (person(L) > organism(L)) &
```

```
(forall M (female(M) >  ~ male(M)) &
(female(mia) & (person(mia) &
(female(mia) & person(mia))))))))))))))))
```

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 and world knowledge. Why is this helpful? For a start, simply realizing that this distinction can be made (even if it is a little 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.

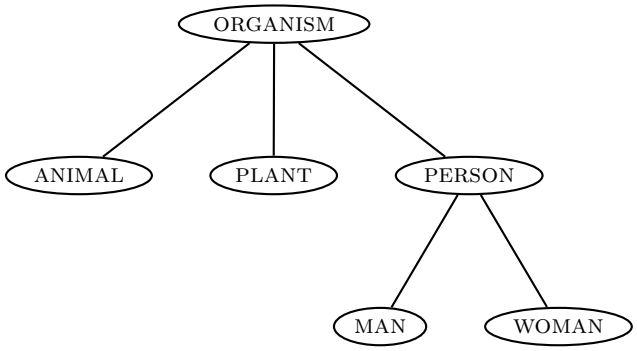
But haven't we already thought hard about the meaning of words? In one sense, yes. For example we have said that the meaning of '**man**' is λy.MAN(y), and that the meaning of '**every**' is λu.λv.$\forall$x(u@x$\rightarrow$v@x). These are important things to say about the meanings of these words. In 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 giving '**every**' the representation λu.λv.$\forall$x(u@x$\rightarrow$v@x) gives us the essence of its meaning; clearly there

are other things that 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 λy.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 detail 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* and *adjectives*. 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 make such knowledge available to Curt.

Let's first look at the nouns. Here is a tree structure made up of six nouns ('organism', 'animal', 'plant', 'animal', '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'.

Let's introduce some standard linguistic terminology: 'organism' is a *hypernym* of 'animal', and 'animal' is a *hyponym* of 'organism'. Incidentally, linguists do *not* view the hypernym or hyponym relations as transitive. That is, while 'object' is a *hypernym* of 'food', and 'food' is a *hypernym* of 'beverage', the linguist would not say that 'object' was a *hypernym* of 'beverage'. In graphical terms, to move to the hypernym we take one step up the tree to the mother mode, and to move to a hyponym we move one step down to a daughter.

Read this way, the classification presented by the above tree makes fairly good sense. Of course it 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.

Let's now consider adjectives. Several pairs of adjectives form *antonym* relations: big/small, male/female, old/young, quick/slow, and so on. We will use this lexical knowledge to define *disjointness* in our ontology. We will do this as follows. We introduce the antonyms 'male'/'female', 'animate'/'inanimate', 'human'/'nonhuman'. Next we define

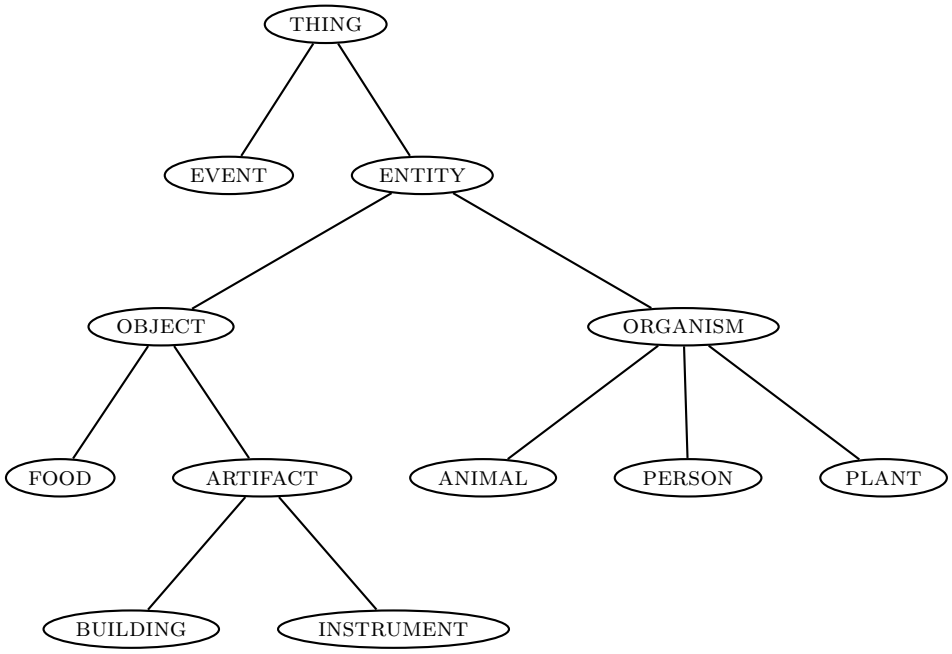
ANIMAL: ANIMATE, NONHUMAN.
PLANT: NONHUMAN, INANIMATE.
PERSON: HUMAN, ANIMATE.
MAN: MALE.
WOMAN: FEMALE.

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 hyponym of PERSON.

That's the basic idea. Let's now apply it on a bigger scale, and 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 to extend or otherwise modify the ontology provided below. 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:



At 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 (yes, you guessed right: a living entity). The remaining classifications should be fairly self-explanatory. We will use the following disjointness relationships:

ABSTRACT − CONCRETE
NONLIVING − LIVING

EDIBLE − INEDIBLE
IMMOBILE − MOBILE
ANIMATE − INANIMATE
NONHUMAN − HUMAN
MALE − FEMALE

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, 'episode' and 'joke' are events, 'foot massage' and 'piercing' are events, a 'five dollar shake' is a beverage, and so on. Incidentally, this approach also let's us deal with homonyms (words that have multiple unconnected meanings), as Exercise 4.6.1 asks the reader to show.

What exactly is involved in going through the lexicon and relating each noun to one of the nodes in the ontology? It means enriching the lexical entries along the following lines:

```
lexEntry(noun,[symbol:car,syntax:[car],hypernyms:[vehicle]]).
lexEntry(noun,[symbol:chainsaw,syntax:[chainsaw],hypernyms:[device]]).
lexEntry(noun,[symbol:criminal,syntax:[criminal],hypernyms:[person]]).
lexEntry(noun,[symbol:customer,syntax:[customer],hypernyms:[person]]).
lexEntry(noun,[symbol:drug,syntax:[drug],hypernyms:[food]]).
lexEntry(noun,[symbol:entity,syntax:[entity],hypernyms:[concrete,thing]]).
lexEntry(noun,[symbol:episode,syntax:[episode],hypernyms:[event]]).
lexEntry(noun,[symbol:event,syntax:[event],hypernyms:[abstract,unisex,thing]]).
lexEntry(noun,[symbol:fdshake,syntax:[five,dollar,shake],hypernyms:[beverage]]).
```

That is, for each noun in the lexicon, we fill the last slot with its hypernym noun. (Actually, in some cases we are abusing the linguistic terminology here, for not all of these are what a linguist would call true hypernyms. It would be more accurate if we spoke of superconcepts.) The hyponym relation doesn't need to be explicitly encoded; as it is just the converse of the hypernym relation it is easy to compute.

As we've already remarked, linguists do not view the hypernym relation as transitive. That is, while 'object' is a *hypernym* of 'food', and 'food' is a *hypernym* of 'beverage', 'object' is not a *hypernym* of 'beverage'. Nonetheless, it is certainly true that anything that is a beverage is an object, and when

performing inference we will be interested in drawing such conclusions, and this means that we will need to chain our our way through the hypernym relation. (To put it in mathematical terms: the relation that is really interesting from the inferential point of view is the transitive closure of the hypernym relation, not the hypernym relation itself.) But this leads to a practical point: if we have been careless, and specified the hypernym relation wrongly, we may get silly answers when we carry out such chaining. In particular we need to take care that there are no words w, v_1, v_2, ..., v_n in our lexicon such that w is a hypernym of v_1, v_1 is a hypernym of v_2, ..., and v_n is a hypernym of w. Now, if you have sat down and looked at all the lexical entries, you will have seen that encoding all this structure is a lot of work, and it would be easy to make such a mistake. Thus it would be useful to have a Prolog utility that checks that our ontological specifications don't contains cycles, and Exercise 4.6.2 asks the reader to provide one.

Adjectives are dealt with in much the same way. We give them entries like the following:

```
lexEntry(adj,[symbol:big,syntax:[big],antonyms:[small]]).
lexEntry(adj,[symbol:male,syntax:[male],antonyms:[female]]).
lexEntry(adj,[symbol:female,syntax:[female],antonyms:[male]]).
lexEntry(adj,[symbol:concrete,syntax:[concrete],antonyms:[abstract]]).
```

Once we have recorded our ontology in the lexicon, 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 compile the ontology into a collection of first-order axioms. We use the predicate `axiom/3` to record the results of this compilation process. For example, for the concept 'car' the compilation returns:

```
axiom(car,1,
      all(X,imp(car(X),vehicle(X)))).
```

And if you examine the lexical entry for the word '`car`', it will be pretty obvious where this came from:

```
lexEntry(noun,[symbol:car,syntax:[car],hypernyms:[vehicle]]).
```

It should clear that the compilation process has turned the information that 'vehicle' is a hypernym of 'car' into first-order logic in the obvious way: it has given us the axiom $\forall x(\text{CAR}(x) \rightarrow \text{VEHICLE}(x))$. But why did we write the symbol `car` as the first argument of `axiom`? We call `car` 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.

Let's look at another example. Here's the lexical entry for the adjective 'concrete':

```
lexEntry(adj,[symbol:concrete,syntax:[concrete],antonyms:[abstract]]).
```

The compilation process yields following:

```
axiom(concrete,1,
      all(X,imp(concrete(X),not(abstract(X))))).
```

That is, it returns $\forall x(\text{CONCRETE}(x) \rightarrow \neg\text{ABSTRACT}(x))$.

The lexical relations recorded in our lexicon (namely, the hypernym relation for nouns and proper names, and the antonym relation for adjectives) generate these axioms automatically; in fact, the compilation is performed when the file `backgroundKnowledge.pl` is consulted. Exercise 4.6.3 asks you to take a closer look at what is involved here.

A practical consequence of this approach to lexical knowledge representation is that if you want to change some of these axioms, the right way to do so is by changing the lexical entries; recall from Chapter 2 that these are stored in the file `englishLexicon.pl`. To inspect the axioms generated, give the command

```
listing(backgroundKnowledge:axiom).
```

after loading Knowledgeable Curt (and before giving the command `curt` to start the dialogue).

Exercise 4.6.1 [easy] Think of a way to put homonyms (words that have multiple unconnected meanings) in the ontology of nouns. For instance, according to WordNet, '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 4.6.2 Write a checker that browses through the lexicon, checking that it is never possible to cycle back to a word by moving along the hypernym relation for nouns.

Exercise 4.6.3 Inspect the file `backgroundKnowledge.pl` and find out how lexical knowledge is converted into axioms by the predicate `generateLexicalKnowledge`.

World Knowledge

The lexical knowledge we have added is useful, but Curt needs more than the knowledge generated by hypernym and antonym relations—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 are about to call 'world knowledge' could (and perhaps should) be thought of as 'lexical knowledge' (see Exercise 4.6.7). 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 our discussion of world knowledge will only illustrate basic issues of direct relevance to computational semantics: we shall give a few simple axioms, and show how they affect Curt's behaviour.

The file `backgroundKnowledge.pl` contains several world knowledge axioms. The axioms are stored in the same `axiom/3` based format used for compiled lexical knowledge. For example, one of our world knowledge axioms states that only persons can dance, and another states that only persons or buildings can collapse. These axioms are stored as follows:

```
axiom(dance,1,
      all(X,imp(dance(X),person(X)))).

axiom(collapse,1,
      all(X,imp(collapse(X),or(person(X),building(X))))).
```

We also have some slightly more complicated axioms, for example one stating that cleaning is something that people do to artifacts and another stating that drinking is something that people do to beverages:

```
axiom(clean,2,
      all(X,all(Y,imp(clean(X,Y),and(person(X),artifact(Y)))))).

axiom(drink,2,
      all(X,all(Y,imp(drink(X,Y),and(person(X),beverage(Y)))))).
```

Why is such knowledge relevant to computational semantics? For a start, with this knowledge at its disposal, Curt will be more choosy about which sentences and discourses it accepts as consistent. For example, Knowledgeable Curt will reject Mia drinks a restaurant as inconsistent. Observe that to reject this sentence Curt must make an inference that draws on both world knowledge (drinking is a relation that holds between beverages and people) and lexical knowledge (the ontology defined in the lexicon precludes restaurants from being beverages).

Furthermore, note that consistency tests that take world knowledge into account not only exclude sentences or discourses, they can also filter out scope ambiguities. For example, 'Every car has a radio' 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 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 (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 (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 forall A (car(A) > exists B (radio(B) & have(A, B)))
```

What lead Curt to reject the other reading? If you give the command `knowledge` and go carefully through the output you will find that Curt drew on (among other things) the following principle:

```
forall T forall U (exists V
     (object(T) & (object(U) & (object(V) &
     (have(T, V) & have(U, V))))) > T = U))
```

This says that if an object `V` is part of two objects `T` and `U`, then `T` and `U` must be the same thing (or to put it another way: no object can be part of two distinct objects). This is part of the world knowledge supplied to Knowledgeable Curt.

Exercise 4.6.4 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 world knowledge `backgroundKnowledge.pl` so that this sentence (and sentences like 'Mia cleans a plant') are accepted, but sentences like 'Mia cleans a footmassage' are rejected.

Exercise 4.6.5 The sentence 'Every boxer has a broken nose' has two logically distinct readings, but world knowledge rules one of these out as biologically implausible. By extending the lexicon and the world knowledge, ensure that Curt rejects the unwanted reading.

Exercise 4.6.6 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 4.6.7 It might be argued that some of the 'world knowledge' axioms in `backgroundKnowledge.pl` actually embody *lexical* knowledge. For example, the axiom stating that drinking is a relation that holds between people and beverages could be viewed this way, for it defines what linguists would call a sortal restriction on verbal arguments. Which other axioms are like this? Which are not? Find a way to include such information in the lexicon, and change the program `backgroundKnowledge` in such a way that it automatically generates such axioms from the lexicon.

Selecting Background Knowledge

One aspect of our representation of lexical and world knowledge remains unexplained: the role of trigger symbols. That is, instead of writing

```
axiom(car,1,
      all(X,imp(car(X),vehicle(X))))
```

why didn't we simply write

```
axiom(all(X,imp(car(X),vehicle(X)))
```

instead? We said earlier that `car` was the trigger symbol for this axiom, but what is it there for?

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 forms the conjunctions of all such formulas, 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.)

Exercise 4.6.8 Examine the `backgroundKnowledge/2` predicate (you will find it in the file `backgroundKnowledge.pl`).

Programs for Knowledgeable Curt

`knowledgeableCurt.pl`
Loading this file loads everything needed to run Knowledgeable Curt.

4.7 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 to 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`) a restriction (here `person(X)`) and a body (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 body.

The semantic role of the restriction and body should be intuitively clear: the restriction represents what is being asked for, and the body 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 return `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, the first time to check whether the question makes sense, then the second time (as an argument to `findall`) to gather together 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 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 checker 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 dialogue 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): ~p found result "~p".',[Engine,Proof])
```

Now we are ready to examine the final step: how 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 to Curt to generate a natural language 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, and Helpful Curt only generates two kinds of noun phrase, proper names and indefinite noun phrases (in Exercise 4.7.4 we ask the reader to extend Helpful Curt so that is 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 use the proper name it found) 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 of 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 then proper name, otherwise it

responds with the proper name alone. As `Answer` is declared in the lexicon to be a list of atoms, Helful 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 lexical look-up for nouns (again we use backtracking to find a match), then using the model builder 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 wrap the calls to `realiseString/4` up into a high level predicate:

```
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 where the case where no satisfying entity is found, the second where one is found, and the third where several is found. And that's Helpful Curt.

Exercise 4.7.1 Some of the answers (formed by nouns) that Curt gives to question are not as specific as they could be. Change the implementation of the realization of answers such that it gives the most specific possible answer.

Exercise 4.7.2 The generation of nouns as coded by `realiseString/3` doesn't necessarily generate the most informative answer. Play around with Helpful Curt and identify cases were funny answers are generated and explain the problem. Propose a solution to overcome this. Implement the solution proposed in the previous exercise.

Exercise 4.7.3 Check restriction of questions.

Exercise 4.7.4 The clause of `realiseString/3` 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/3` that implements this.

Exercise 4.7.5 Extend the grammar fragment and the driver predicate in such a way that yes-no questions are covered as well.

Exercise 4.7.6 Reimplement 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?' (For gap-threading, consult Pereira & Shieber's book.)

Programs for Helpful Curt

`helpfulCurt.pl`
Loading this file loads everything needed to run Helpful Curt.

Here's an overview of all the Curt code and its capabilities:

Programs for the Curt Family

`babyCurt.pl`
Baby Curt. No inference capabilities.

`rugratCurt.pl`
Rugrat Curt. Consistency checking with naive theorem prover (the free-variable tableau from Chapter 5).

`cleverCurt.pl`
Clever Curt. Consistency checking with sophisticated model builder and theorem prover.

`sensitiveCurt.pl`
Sensitive Curt. Consistency and informativeness checking with sophisticated model builder and theorem prover.

`scrupulousCurt.pl`
Scrupulous Curt. Eliminating logically equivalent readings.

`knowledgeableCurt.pl`
Knowledgeable Curt. Incorporating Background Knowledge.

`helpfulCurt.pl`
Helpful Curt. Question Answering.

`elimEquivReadings.pl`
Code for eliminating logically equivalent readings.

`curtPredicates.pl`
Auxiliary predicates used by the Curt family.

`callInference.pl`
Modify this file to experiment with different theorem prover/model builder combinations.