# 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:

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

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

First, the vocabulary tells us *what* we're going to be talking about. In the present case, we're going to be talking about *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* $\tau$ 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 $\phi$ and $\psi$ are wffs then so are $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $(\phi \rightarrow \psi)$.
3. If $\phi$ is a wff, and x is a variable, then both $\exists x\phi$ and $\forall x\phi$ are wffs. (We call $\phi$ 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 $\phi$ are $\phi$ itself and all the formulas used to build $\phi$. 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 $\phi$ or in $\psi$, 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 $\phi$ or in $\psi$, 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 $\phi$, 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 $\phi$, 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

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

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. $\textsc{robber}(y)$
2. $\textsc{love}(x,y)$

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 $\tau$ be a term. Then the *interpretation* of $\tau$ with respect to $M$ and $g$ is $F(\tau)$ if $\tau$ is a constant, and $g(\tau)$ if $\tau$ is a variable. We denote the interpretation of $\tau$ 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 $\phi$ 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 $M, g \models \phi$ ($\phi$ is satisfied in $M$ with respect to the assignment of values to variables $g$) is defined inductively as follows:

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

(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 $\phi$ in the model. That is, we have to try to find *some* value for x that satisfies $\phi$ 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 $\phi$ 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 $\phi$ 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* $\phi$ 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 $\phi$ is not a sentence but a formula containing free variables. Second, show that the claim is *true* if $\phi$ is a *sentence*.

**Exercise 1.1.12** For any formula $\phi$, given two assignments $g$ and $g'$ which differ only in what they assign to variables *not* in $\phi$, 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 $\phi$ 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 $\tau$ is a constant, and $g(\tau)$ if $\tau$ is a variable. On the other hand, if $\tau$ 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 $\tau_1$ and $\tau_2$ 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 $\tau_1$ and $\tau_2$, 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 $\tau_1$ and $\tau_2$ 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 $\phi$, is $\phi$ 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, ROBBER(MIA) $\wedge$ $\neg$ROBBER(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 $\phi$, is $\phi$ 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 $\phi$ is a valid formula, it is impossible to find a situation and a context in which $\phi$ 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 $\phi$ 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 $\phi$ 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 $\psi$ are a finite collection of first-order formulas. Then we say that the argument with *premises* $\phi_1, \ldots, \phi_n$ and *conclusion* $\psi$ 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 $\psi$ is valid. Incidentally, there are many ways of speaking of valid arguments. For example, it is also common to say that $\psi$ is a *valid inference* from the premises $\phi_1, \ldots, \phi_n$, or that $\psi$ is a *logical consequence* of $\phi_1, \ldots, \phi_n$, or that $\psi$ 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 $\psi$ 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 $\forall x(\text{ROBBER}(x) \rightarrow \text{CUSTOMER}(x))$ and $\text{ROBBER}(\text{MIA})$ and the conclusion $\text{CUSTOMER}(\text{MIA})$ is valid. That is:

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

But now consider the following (valid) formula:

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

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

Validity and valid arguments are central concepts in model theory. Logicians regard validities as important, 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 $\phi$, is $\phi$ 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 $\psi$ 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 $\psi$ 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 $\psi$, we clearly *have* learned something new. Hence, if $\phi_1, \ldots, \phi_n \not\models \psi$, then we shall say that $\psi$ is *informative with respect to* $\phi_1, \ldots, \phi_n$.

Of course, by appealing to the Semantic Deduction Theorem, we can reduce testing $\psi$ for informativeness (or uninformativeness) with respect to $\phi_1$, ..., $\phi_n$ to ordinary informativeness checking. For the theorem tells us that $\psi$ is informative with respect to $\phi_1$, ..., $\phi_n$ 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 $\phi$ and $\psi$ are *logically equivalent* if and only if $\phi \models \psi$ and $\psi \models \phi$. For all formulas $\phi$, $\psi$, and $\theta$:

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. $\phi$ is consistent (that is, satisfiable) if and only if $\neg\phi$ is informative (that is, invalid).

2. $\phi$ is inconsistent (that is, unsatisfiable) if and only if $\neg\phi$ is uninformative (that is, valid).

3. $\phi$ is informative (that is, invalid) if and only if $\neg\phi$ is consistent (that is, satisfiable).

4. $\phi$ 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 $\phi$ 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 $\phi$ 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 $\phi$ 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 $\phi$ 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 $\phi$ and $\psi$ 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 $\phi$ 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 $\phi$ 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:

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

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.