CHAPTER 8

FIRST-ORDER LOGIC

In which we notice that the world is blessed with many objects, some of which are related to other objects, and in which we endeavor to reason about them.

Propositional logic sufficed to illustrate the basic concepts of logic, inference, and knowledge-based agents. Unfortunately, propositional logic is limited in what it can say. In this chapter, we examine **first-order logic**,[1] which can concisely represent much more. We begin in Section 8.1 with a discussion of representation languages in general; Section 8.2 covers the syntax and semantics of first-order logic; Sections 8.3 and 8.4 illustrate the use of first-order logic for simple representations.

First-order logic

8.1 Representation Revisited

In this section, we discuss the nature of representation languages. Programming languages (such as C++ or Java or Python) are the largest class of formal languages in common use. Data structures within programs can be used to represent facts; for example, a program could use a 4×4 array to represent the contents of the wumpus world. Thus, the programming language statement $World[2,2] \leftarrow Pit$ is a fairly natural way to assert that there is a pit in square [2,2]. Putting together a string of such statements is sufficient for running a simulation of the wumpus world.

What programming languages lack is a general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain. This procedural approach can be contrasted with the **declarative** nature of propositional logic, in which knowledge and inference are separate, and inference is entirely domain independent. SQL databases take a mix of declarative and procedural knowledge.

A second drawback of data structures in programs (and of databases) is the lack of any easy way to say, for example, "There is a pit in [2,2] or [3,1]" or "If the wumpus is in [1,1] then he is not in [2,2]." Programs can store a single value for each variable, and some systems allow the value to be "unknown," but they lack the expressiveness required to directly handle partial information.

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation. Propositional logic has a third property that is desirable in representation languages, namely, **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For

Compositionality

[1] First-order logic is also called **first-order predicate calculus**; it may be abbreviated as **FOL** or **FOPC**.

example, the meaning of "$S_{1,4} \wedge S_{1,2}$" is related to the meanings of "$S_{1,4}$" and "$S_{1,2}$." It would be very strange if "$S_{1,4}$" meant that there is a stench in square [1,4] and "$S_{1,2}$" meant that there is a stench in square [1,2], but "$S_{1,4} \wedge S_{1,2}$" meant that France and Poland drew 1–1 in last week's ice hockey qualifying match.

However, propositional logic, as a factored representation, lacks the expressive power to *concisely* describe an environment with many objects. For example, we were forced to write a separate rule about breezes and pits for each square, such as

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) .$$

In English, on the other hand, it seems easy enough to say, once and for all, "Squares adjacent to pits are breezy." The syntax and semantics of English make it possible to describe the environment concisely: English, like first-order logic, is a structured representation.

8.1.1 The language of thought

Natural languages (such as English or Spanish) are very expressive indeed. We managed to write almost this whole book in natural language, with only occasional lapses into other languages (mainly mathematics and diagrams). There is a long tradition in linguistics and the philosophy of language that views natural language as a declarative knowledge representation language. If we could uncover the rules for natural language, we could use them in representation and reasoning systems and gain the benefit of the billions of pages that have been written in natural language.

The modern view of natural language is that it serves as a medium for *communication* rather than pure representation. When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence "Look!" represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the *context* in which the sentence was spoken. Clearly, one could not store a sentence such as "Look!" in a knowledge base and expect to recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented.

Natural languages also suffer from *ambiguity*, a problem for a representation language. As Pinker (1995) puts it: "When people think about *spring*, surely they are not confused as to whether they are thinking about a season or something that goes *boing*—and if one word can correspond to two thoughts, thoughts can't be words."

The famous **Sapir–Whorf hypothesis** Whorf (1956) claims that our understanding of the world is strongly influenced by the language we speak. It is certainly true that different speech communities divide up the world differently. The French have two words "chaise" and "fauteuil," for a concept that English speakers cover with one: "chair." But English speakers can easily recognize the category *fauteuil* and give it a name—roughly "open-arm chair"—so does language really make a difference? Whorf relied mainly on intuition and speculation, and his ideas have been largely dismissed, but in the intervening years we actually have real data from anthropological, psychological, and neurological studies.

For example, can you remember which of the following two phrases formed the opening of Section 8.1?

> "In this section, we discuss the nature of representation languages ..."
>
> "This section covers the topic of knowledge representation languages ..."

Wanner (1974) did a similar experiment and found that subjects made the right choice at chance level—about 50% of the time—but remembered the content of what they read with better than 90% accuracy. This suggests that people interpret the words they read and form an internal *nonverbal* representation, and that the exact words are not consequential.

More interesting is the case in which a concept is completely absent in a language. Speakers of the Australian aboriginal language Guugu Yimithirr have no words for relative (or *egocentric*) directions, such as front, back, right, or left. Instead they use absolute directions, saying, for example, the equivalent of "I have a pain in my north arm." This difference in language makes a difference in behavior: Guugu Yimithirr speakers are better at navigating in open terrain, while English speakers are better at placing the fork to the right of the plate.

Language also seems to influence thought through seemingly arbitrary grammatical features such as the gender of nouns. For example, "bridge" is masculine in Spanish and feminine in German. Boroditsky (2003) asked subjects to choose English adjectives to describe a photograph of a particular bridge. Spanish speakers chose *big*, *dangerous*, *strong*, and *towering*, whereas German speakers chose *beautiful*, *elegant*, *fragile*, and *slender*.

Words can serve as anchor points that affect how we perceive the world. Loftus and Palmer (1974) showed experimental subjects a movie of an auto accident. Subjects who were asked "How fast were the cars going when they contacted each other?" reported an average of 32 mph, while subjects who were asked the question with the word "smashed" instead of "contacted" reported 41mph for the same cars in the same movie. Overall, there are measurable but small differences in cognitive processing by speakers of different languages, but no convincing evidence that this leads to a major difference in world view.

In a logical reasoning system that uses conjunctive normal form (CNF), we can see that the linguistic forms "$\neg(A \vee B)$" and "$\neg A \wedge \neg B$" are the same because we can look inside the system and see that the two sentences are stored as the same canonical CNF form. It is starting to become possible to do something similar with the human brain. Mitchell *et al.* (2008) put subjects in an functional magnetic resonance imaging (fMRI) machine, showed them words such as "celery," and imaged their brains. A machine learning program trained on (word, image) pairs was able to predict correctly 77% of the time on binary choice tasks (e.g., "celery" or "airplane"). The system can even predict at above-chance levels for words it has never seen an fMRI image of before (by considering the images of related words) and for people it has never seen before (proving that fMRI reveals some level of common representation across people). This type of work is still in its infancy, but fMRI (and other imaging technology such as intracranial electrophysiology (Sahin *et al.*, 2009)) promises to give us much more concrete ideas of what human knowledge representations are like.

From the viewpoint of formal logic, representing the same knowledge in two different ways makes absolutely no difference; the same facts will be derivable from either representation. In practice, however, one representation might require fewer steps to derive a conclusion, meaning that a reasoner with limited resources could get to the conclusion using one representation but not the other. For *nondeductive* tasks such as learning from experience, outcomes are *necessarily* dependent on the form of the representations used. We show in Chapter 19 that when a learning program considers two possible theories of the world, both of which are consistent with all the data, the most common way of breaking the tie is to choose the most succinct theory—and that depends on the language used to represent theories. Thus, the influence of language on thought is unavoidable for any agent that does learning.

8.1.2 Combining the best of formal and natural languages

We can adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks. When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to **objects** (squares, pits, wumpuses) and verbs and verb phrases along with adjectives and adverbs that refer to **relations** among objects (is breezy, is adjacent to, shoots). Some of these relations are **functions**—relations in which there is only one "value" for a given "input." It is easy to start listing examples of objects, relations, and functions:

Object
Relation
Function

- Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...
- Relations: these can be unary relations or **properties** such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...
- Functions: father of, best friend, third inning of, one more than, beginning of ...

Property

Indeed, almost any assertion can be thought of as referring to objects and properties or relations. Some examples follow:

- "One plus two equals three."
  Objects: one, two, three, one plus two; Relation: equals; Function: plus. ("One plus two" is a name for the object that is obtained by applying the function "plus" to the objects "one" and "two." "Three" is another name for this object.)
- "Squares neighboring the wumpus are smelly."
  Objects: wumpus, squares; Property: smelly; Relation: neighboring.
- "Evil King John ruled England in 1200."
  Objects: John, England, 1200; Relation: ruled during; Properties: evil, king.

The language of **first-order logic**, whose syntax and semantics we define in the next section, is built around objects and relations. It has been important to mathematics, philosophy, and artificial intelligence precisely because those fields—and indeed, much of everyday human existence—can be usefully thought of as dealing with objects and the relations among them. First-order logic can also express facts about *some* or *all* of the objects in the universe. This enables one to represent general laws or rules, such as the statement "Squares neighboring the wumpus are smelly."

Ontological commitment

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of *reality*. Mathematically, this commitment is expressed through the nature of the formal models with respect to which the truth of sentences is defined. For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states—true or false—and each model assigns *true* or *false* to each proposition symbol (see Section 7.4.2). First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. (See Figure 8.1.) The formal models are correspondingly more complicated than those for propositional logic.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0,1]$
Fuzzy logic	facts with degree of truth $\in [0,1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

This ontological commitment is a great strength of logic (both propositional and first-order), because it allows us to start with true statements and infer other true statements. It is especially powerful in domains where every proposition has clear boundaries, such as mathematics or the wumpus world, where a square either does or doesn't have a pit; there is no possibility of a square with a vaguely pit-like indentation. But in the real world, many propositions have vague boundaries: Is Vienna a large city? Does this restaurant serve delicious food? Is that person tall? It depends who you ask, and their answer might be "kind of."

One response is to refine the representation: if a crude line dividing cities into "large" and "not large" leaves out too much information for the application in question, then one can increase the number of size categories or use a *Population* function symbol. Another proposed solution comes from **Fuzzy logic**, which makes the ontological commitment that propositions have a **degree of truth** between 0 and 1. For example, the sentence "Vienna is a large city" might be true to degree 0.8 in fuzzy logic, while "Paris is a large city" might be true to degree 0.9. This corresponds better to our intuitive conception of the world, but it makes it harder to do inference: instead of one rule to determine the truth of $A \wedge B$, fuzzy logic needs different rules depending on the domain. Another possibility, covered in Section 24.1, is to assign each concept to a point in a multidimensional space, and then measure the distance between the concept "large city" and the concept "Vienna" or "Paris."

Fuzzy logic

Degree of truth

Various special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) "first class" status within the logic, rather than simply defining them within the knowledge base. **Higher-order logic** views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about *all* relations—for example, one could wish to define what it means for a relation to be transitive. Unlike most special-purpose logics, higher-order logic is strictly more expressive than first-order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

Temporal logic

Higher-order logic

A logic can also be characterized by its **epistemological commitments**—the possible states of knowledge that it allows with respect to each fact. In both propositional and first-order logic, a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence.

Epistemological commitment

Systems using **probability theory**, on the other hand, can have any *degree of belief*, or *subjective likelihood*, ranging from 0 (total disbelief) to 1 (total belief). It is important not to confuse the degree of belief in probability theory with the degree of truth in fuzzy logic. Indeed, some fuzzy systems allow uncertainty (degree of belief) about degrees of truth. For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75 and in [2, 3] with probability 0.25 (although the wumpus is definitely in one particular square).

8.2 Syntax and Semantics of First-Order Logic

We begin this section by specifying more precisely the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations. Then we introduce the various elements of the language, explaining their semantics as we go along. The main points are how the language facilitates concise representations and how its semantics leads to sound reasoning procedures.

8.2.1 Models for first-order logic

Chapter 7 said that the models of a logical language are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values.

Models for first-order logic are much more interesting. First, they have objects in them!

Domain

Domain elements

The **domain** of a model is the set of objects or **domain elements** it contains. The domain is required to be *nonempty*—every possible world must contain at least one object. (See Exercise 8.EMPT for a discussion of empty worlds.) Mathematically speaking, it doesn't matter *what* these objects are—all that matters is *how many* there are in each particular model—but for pedagogical purposes we'll use a concrete example. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

Tuple

The objects in the model may be *related* in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

$$\{\langle \text{Richard the Lionheart, King John}\rangle, \langle \text{King John, Richard the Lionheart}\rangle\}. \quad (8.1)$$

(Here we have named the objects in English, but you may, if you wish, mentally substitute the pictures for the names.) The crown is on King John's head, so the "on head" relation contains just one tuple, ⟨the crown, King John⟩. The "brother" and "on head" relations are binary relations—that is, they relate pairs of objects. The model also contains unary relations, or properties: the "person" property is true of both Richard and John; the "king" property is true only of John (presumably because Richard is dead at this point); and the "crown" property is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function—a mapping from a one-element tuple to an object—that

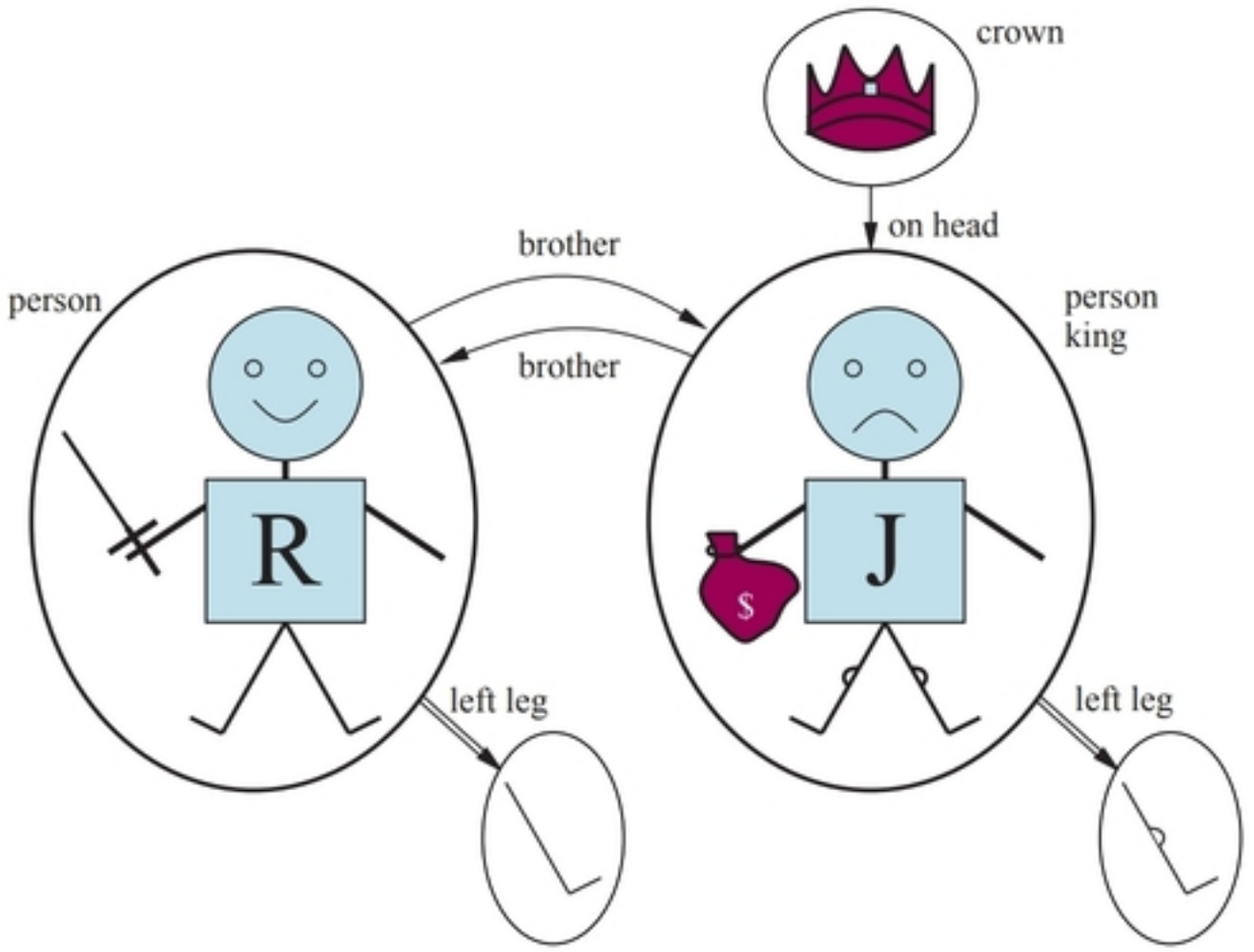

Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

includes the following mappings:

$$\begin{aligned}&\langle\text{Richard the Lionheart}\rangle \rightarrow \text{Richard's left leg}\\&\langle\text{King John}\rangle \rightarrow \text{John's left leg}.\end{aligned} \tag{8.2}$$

Strictly speaking, models in first-order logic require **total functions**, that is, there must be a value for every input tuple. Thus the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional "invisible" object that is the left leg of everything that has no left leg, including itself. Fortunately, as long as one makes no assertions about the left legs of things that have no left legs, these technicalities are of no import.

Total functions

So far, we have described the elements that populate models for first-order logic. The other essential part of a model is the link between those elements and the vocabulary of the logical sentences, which we explain next.

8.2.2 Symbols and interpretations

We turn now to the syntax of first-order logic. The impatient reader can obtain a complete description from the formal grammar in Figure 8.3.

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols *Richard* and *John*; the predicate symbols *Brother*, *OnHead*, *Person*, *King*, and *Crown*; and the function symbol *LeftLeg*. As with proposition symbols, the choice of names is entirely up to the user. Each predicate and function symbol comes with an **arity** that fixes the number of arguments.

Constant symbol

Predicate symbol

Function symbol

Arity

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \\
& \mid & \neg\, \mathit{Sentence} \\
& \mid & \mathit{Sentence} \wedge \mathit{Sentence} \\
& \mid & \mathit{Sentence} \vee \mathit{Sentence} \\
& \mid & \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
& \mid & \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
& \mid & \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term}, \ldots) \\
& \mid & \mathit{Constant} \\
& \mid & \mathit{Variable} \\
\mathit{Quantifier} & \rightarrow & \forall \mid \exists \\
\mathit{Constant} & \rightarrow & A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1030 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

Interpretation

Intended interpretation

Every model must provide the information required to determine if any given sentence is true or false. Thus, in addition to its objects, relations, and functions, each model includes an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for our example—which a logician would call the **intended interpretation**—is as follows:

- *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- *Brother* refers to the brotherhood relation—that is, the set of tuples of objects given in Equation (8.1); *OnHead* is a relation that holds between the crown and King John; *Person*, *King*, and *Crown* are unary relations that identify persons, kings, and crowns.
- *LeftLeg* refers to the "left leg" function as defined in Equation (8.2).

There are many other possible interpretations, of course. For example, one interpretation maps *Richard* to the crown and *John* to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols *Richard* and *John*.

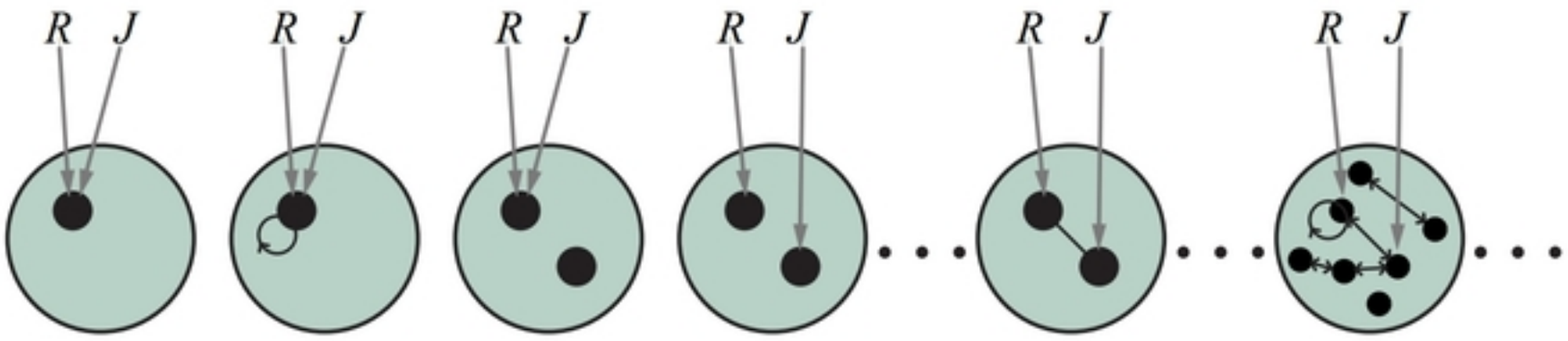

Figure 8.4 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

Notice that not all the objects need have a name—for example, the intended interpretation does not name the crown or the legs. It is also possible for an object to have several names; there is an interpretation under which both *Richard* and *John* refer to the crown.[2] If you find this possibility confusing, remember that, in propositional logic, it is perfectly possible to have a model in which *Cloudy* and *Sunny* are both true; it is the job of the knowledge base to rule out models that are inconsistent with our knowledge.

In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, function symbols to functions on those objects, and predicate symbols to relations. Just as with propositional logic, entailment, validity, and so on are defined in terms of *all possible models*. To get an idea of what the set of all possible models looks like, see Figure 8.4. It shows that models vary in how many objects they contain—from one to infinity—and in the way the constant symbols map to objects.

Because the number of first-order models is unbounded, we cannot check entailment by enumerating them all (as we did for propositional logic). Even if the number of objects is restricted, the number of combinations can be very large. (See Exercise 8.MCNT.) For the example in Figure 8.4, there are 137,506,194,466 models with six or fewer objects.

8.2.3 Terms

A **term** is a logical expression that refers to an object. Constant symbols are terms, but it is not always convenient to have a distinct symbol to name every object. In English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use $LeftLeg(John)$.[3] Term

In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a "subroutine call" that "returns a value." There is no *LeftLeg* subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing

---

[2] Later, in Section 8.2.8, we examine a semantics in which every object must have exactly one name.

[3] **λ-expressions** (lambda expressions) provide a useful notation in which new function symbols are constructed "on the fly." For example, the function that squares its argument can be written as $(\lambda x : x \times x)$ and can be applied to arguments just like any other function symbol. A λ-expression can also be defined and used as a predicate symbol. The `lambda` operator in Lisp and Python plays exactly the same role. Notice that the use of λ in this way does *not* increase the formal expressive power of first-order logic, because any sentence that includes a λ-expression can be rewritten by "plugging in" its arguments to yield an equivalent sentence.

that John must have one) without ever providing a definition of *LeftLeg*. This is something that cannot be done with subroutines in programming languages.

The formal semantics of terms is straightforward. Consider a term $f(t_1, \ldots, t_n)$. The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them $d_1, \ldots, d_n$); and the term as a whole refers to the object that is the value of the function F applied to $d_1, \ldots, d_n$. For example, suppose the *LeftLeg* function symbol refers to the function shown in Equation (8.2) and *John* refers to King John, then $LeftLeg(John)$ refers to King John's left leg. In this way, the interpretation fixes the referent of every term.

8.2.4 Atomic sentences

Atomic sentence

Atom

Now that we have terms for referring to objects and predicate symbols for referring to relations, we can combine them to make **atomic sentences** that state facts. An **atomic sentence** (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as

$$Brother(Richard, John).$$

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.[4] Atomic sentences can have complex terms as arguments. Thus,

$$Married(Father(Richard), Mother(John))$$

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).[5]

*An atomic sentence is **true** in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

8.2.5 Complex sentences

We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus. Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \wedge Brother(John, Richard)$$
$$King(Richard) \vee King(John)$$
$$\neg King(Richard) \Rightarrow King(John).$$

8.2.6 Quantifiers

Quantifier

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called *universal* and *existential*.

Universal quantification ($\forall$)

Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as "Squares neighboring the wumpus are smelly" and "All kings

[4] We usually follow the argument-ordering convention that $P(x, y)$ is read as "x is a P of y."

[5] This ontology only recognizes one father and one mother for each person. A more complex ontology could recognize biological mother, birth mother, adoptive mother, etc.

are persons" are the bread and butter of first-order logic. We deal with the first of these in Section 8.3. The second rule, "All kings are persons," is written in first-order logic as

$$\forall x \ \mathit{King}(x) \Rightarrow \mathit{Person}(x) .$$

The **universal quantifier** $\forall$ is usually pronounced "For all ...". (Remember that the upside-down A stands for "all.") Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, $\mathit{LeftLeg}(x)$. A term with no variables is called a **ground term**.

Universal quantifier

Variable

Ground term

Intuitively, the sentence $\forall x\, P$, where P is any logical sentence, says that P is true for every object x. More precisely, $\forall x\, P$ is true in a given model if P is true in all possible **extended interpretations** constructed from the interpretation given in the model, where each extended interpretation specifies a domain element to which x refers.

Extended interpretation

This sounds complicated, but it is really just a careful way of stating the intuitive meaning of universal quantification. Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

$x \rightarrow$ Richard the Lionheart,
$x \rightarrow$ King John,
$x \rightarrow$ Richard's left leg,
$x \rightarrow$ John's left leg,
$x \rightarrow$ the crown.

The universally quantified sentence $\forall x \ \mathit{King}(x) \Rightarrow \mathit{Person}(x)$ is true in the original model if the sentence $\mathit{King}(x) \Rightarrow \mathit{Person}(x)$ is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king $\Rightarrow$ Richard the Lionheart is a person.
King John is a king $\Rightarrow$ King John is a person.
Richard's left leg is a king $\Rightarrow$ Richard's left leg is a person.
John's left leg is a king $\Rightarrow$ John's left leg is a person.
The crown is a king $\Rightarrow$ the crown is a person.

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of "All kings are persons"? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for $\Rightarrow$ (Figure 7.8 on page 219), we see that the implication is true whenever its premise is false—*regardless* of the truth of the conclusion. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implications, we end up asserting the conclusion of the rule just for those objects for which the premise is true and saying nothing at all about those objects for which the premise is false. Thus, the truth-table definition of $\Rightarrow$ turns out to be perfect for writing general rules with universal quantifiers.

A common mistake, made frequently even by diligent readers who have read this paragraph several times, is to use conjunction instead of implication. The sentence

$$\forall x \ \mathit{King}(x) \wedge \mathit{Person}(x)$$

would be equivalent to asserting

> Richard the Lionheart is a king $\wedge$ Richard the Lionheart is a person,
> King John is a king $\wedge$ King John is a person,
> Richard's left leg is a king $\wedge$ Richard's left leg is a person,

and so on. Obviously, this does not capture what we want.

Existential quantification ($\exists$)

Existential quantifier

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object without naming it, by using an **existential quantifier**. To say, for example, that King John has a crown on his head, we write

$$\exists x \;\; Crown(x) \wedge OnHead(x, John) .$$

$\exists x$ is pronounced "There exists an x such that ..." or "For some x...".

Intuitively, the sentence $\exists x\, P$ says that P is true for at least one object x. More precisely, $\exists x\, P$ is true in a given model if P is true in *at least one* extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

> Richard the Lionheart is a crown $\wedge$ Richard the Lionheart is on John's head;
> King John is a crown $\wedge$ King John is on John's head;
> Richard's left leg is a crown $\wedge$ Richard's left leg is on John's head;
> John's left leg is a crown $\wedge$ John's left leg is on John's head;
> The crown is a crown $\wedge$ the crown is on John's head.

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence "King John has a crown on his head." [6]

Just as $\Rightarrow$ appears to be the natural connective to use with $\forall$, $\wedge$ is the natural connective to use with $\exists$. Using $\wedge$ as the main connective with $\forall$ led to an overly strong statement in the example in the previous section; using $\Rightarrow$ with $\exists$ usually leads to a very weak statement, indeed. Consider the following sentence:

$$\exists x \;\; Crown(x) \Rightarrow OnHead(x, John) .$$

On the surface, this might look like a reasonable rendition of our sentence. Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

> Richard the Lionheart is a crown $\Rightarrow$ Richard the Lionheart is on John's head;
> King John is a crown $\Rightarrow$ King John is on John's head;
> Richard's left leg is a crown $\Rightarrow$ Richard's left leg is on John's head;

and so on. An implication is true if both premise and conclusion are true, *or if its premise is false*; so if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true whenever *any* object fails to satisfy the premise; hence such sentences really do not say much at all.

---

[6] There is a variant of the existential quantifier, usually written $\exists^1$ or $\exists!$, that means "There exists exactly one." The same meaning can be expressed using equality statements.

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

$$\forall x\ \forall y\ \mathit{Brother}(x,y) \Rightarrow \mathit{Sibling}(x,y)\,.$$

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

$$\forall x,y\ \mathit{Sibling}(x,y) \Leftrightarrow \mathit{Sibling}(y,x)\,.$$

In other cases we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves:

$$\forall x\ \exists y\ \mathit{Loves}(x,y)\,.$$

On the other hand, to say "There is someone who is loved by everyone," we write

$$\exists y\ \forall x\ \mathit{Loves}(x,y)\,.$$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. $\forall x\ (\exists y\ \mathit{Loves}(x,y))$ says that *everyone* has a particular property, namely, the property that they love someone. On the other hand, $\exists y\ (\forall x\ \mathit{Loves}(x,y))$ says that *someone* in the world has a particular property, namely the property of being loved by everybody.

Some confusion can arise when two quantifiers are used with the same variable name. Consider the sentence

$$\forall x\ (\mathit{Crown}(x) \vee (\exists x\ \mathit{Brother}(\mathit{Richard},x)))\,.$$

Here the x in $\mathit{Brother}(\mathit{Richard},x)$ is *existentially* quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification. Another way to think of it is this: $\exists x\ \mathit{Brother}(\mathit{Richard},x)$ is a sentence about Richard (that he has a brother), not about x; so putting a $\forall x$ outside it has no effect. It could equally well have been written $\exists z\ \mathit{Brother}(\mathit{Richard},z)$. Because this can be a source of confusion, we will always use different variable names with nested quantifiers.

Connections between $\forall$ and $\exists$

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x\ \neg \mathit{Likes}(x,\mathit{Parsnips}) \quad \text{is equivalent to} \quad \neg\exists x\ \mathit{Likes}(x,\mathit{Parsnips})\,.$$

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

$$\forall x\ \mathit{Likes}(x,\mathit{IceCream}) \quad \text{is equivalent to} \quad \neg\exists x\ \neg\mathit{Likes}(x,\mathit{IceCream})\,.$$

Because $\forall$ is really a conjunction over the universe of objects and $\exists$ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{lll}
\neg\exists x\ P \equiv \forall x\ \neg P & \qquad & \neg(P \vee Q) \equiv \neg P \wedge \neg Q \\
\neg\forall x\ P \equiv \exists x\ \neg P & & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\
\forall x\ P \equiv \neg\exists x\ \neg P & & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\
\exists x\ P \equiv \neg\forall x\ \neg P & & P \vee Q \equiv \neg(\neg P \wedge \neg Q)\,.
\end{array}$$

Thus, we do not really need both $\forall$ and $\exists$, just as we do not really need both $\wedge$ and $\vee$. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

8.2.7 Equality

Equality symbol

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to signify that two terms refer to the same object. For example,

$$Father(John) = Henry$$

says that the object referred to by $Father(John)$ and the object referred to by $Henry$ are the same. Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.

The equality symbol can be used to state facts about a given function, as we just did for the *Father* symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard) \wedge \neg(x = y) \,.$$

The sentence

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard)$$

does not have the intended meaning. In particular, it is true in the model of Figure 8.2, where Richard has only one brother. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x = y)$ rules out such models. The notation $x \neq y$ is sometimes used as an abbreviation for $\neg(x = y)$.

8.2.8 Database semantics

Continuing the example from the previous section, suppose that we believe that Richard has two brothers, John and Geoffrey.[7] We could write

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \,, \tag{8.3}$$

but that wouldn't completely capture the state of affairs. First, this assertion is true in a model where Richard has only one brother—we need to add $John \neq Geoffrey$. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:

$$\begin{aligned} &Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \wedge John \neq Geoffrey \\ &\wedge \forall x \;\; Brother(x, Richard) \;\Rightarrow\; (x = John \vee x = Geoffrey) \,. \end{aligned}$$

This logical sentence seems much more cumbersome than the corresponding English sentence. But if we fail to translate the English properly, our logical reasoning system will make mistakes. Can we devise a semantics that allows a more straightforward logical sentence?

Unique-names assumption

Closed-world assumption

Domain closure

One proposal that is very popular in database systems works as follows. First, we insist that every constant symbol refer to a distinct object—the **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false—the **closed-world assumption**. Finally, we invoke **domain closure**, meaning that each model contains no more domain elements than those named by the constant symbols.

[7] Actually he had four, the others being William and Henry.

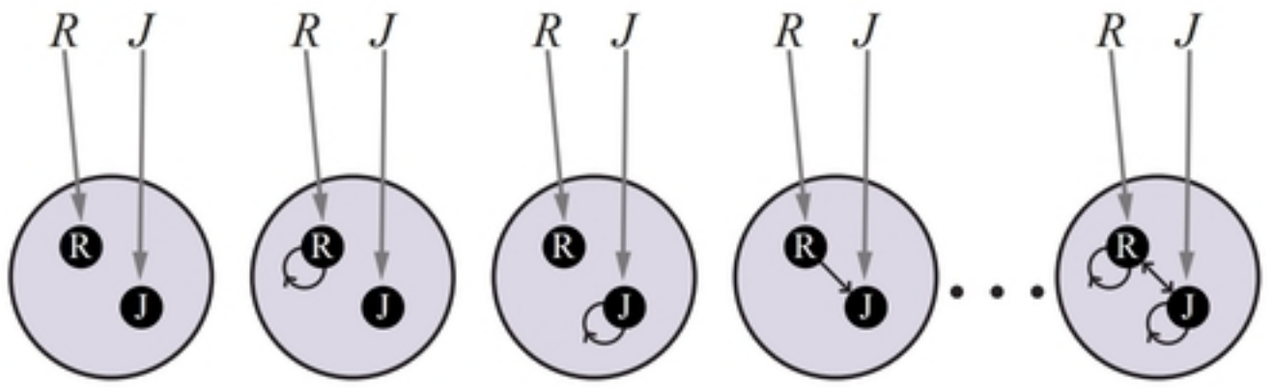

Figure 8.5 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

Under the resulting semantics, Equation (8.3) does indeed state that Richard has exactly two brothers, John and Geoffrey. We call this **database semantics** to distinguish it from the standard semantics of first-order logic. Database semantics is also used in logic programming systems, as explained in Section 9.4.4.

Database semantics

It is instructive to consider the set of all possible models under database semantics for the same case as shown in Figure 8.4 (page 259). Figure 8.5 shows some of the models, ranging from the model with no tuples satisfying the relation to the model with all tuples satisfying the relation. With two objects, there are four possible two-element tuples, so there are $2^4 = 16$ different subsets of tuples that can satisfy the relation. Thus, there are 16 possible models in all—a lot fewer than the infinitely many models for the standard first-order semantics. On the other hand, the database semantics requires definite knowledge of what the world contains.

This example brings up an important point: there is no one "correct" semantics for logic. The usefulness of any proposed semantics depends on how concise and intuitive it makes the expression of the kinds of knowledge we want to write down, and on how easy and natural it is to develop the corresponding rules of inference. Database semantics is most useful when we are certain about the identity of all the objects described in the knowledge base and when we have all the facts at hand; in other cases, it is quite awkward. For the rest of this chapter, we assume the standard semantics while noting instances in which this choice leads to cumbersome expressions.

8.3 Using First-Order Logic

Now that we have defined an expressive logical language, let's learn how to use it. In this section, we provide example sentences in some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

Domain

We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at the wumpus world. Section 8.4.2 contains a more substantial example (electronic circuits) and Chapter 10 covers everything in the universe.

8.3.1 Assertions and queries in first-order logic

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a

Assertion

person, and all kings are persons:

$$\text{TELL}(KB, King(John)).$$
$$\text{TELL}(KB, Person(Richard)).$$
$$\text{TELL}(KB, \forall x\ King(x) \Rightarrow Person(x)).$$

We can ask questions of the knowledge base using ASK. For example,

$$\text{ASK}(KB, King(John))$$

returns *true*. Questions asked with ASK are called **queries** or **goals**. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the three assertions above, the query

$$\text{ASK}(KB, Person(John))$$

should also return *true*. We can ask quantified queries, such as

$$\text{ASK}(KB, \exists x\ Person(x)).$$

The answer is *true*, but this is perhaps not as helpful as we would like. It is rather like answering "Can you tell me the time?" with "Yes." If we want to know what value of x makes the sentence true, we will need a different function, which we call ASKVARS,

$$\text{ASKVARS}(KB, Person(x))$$

and which yields a stream of answers. In this case there will be two answers: $\{x/John\}$ and $\{x/Richard\}$. Such an answer is called a **substitution** or **binding list**. ASKVARS is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific values. That is not the case with first-order logic; in a *KB* that has been told only that $King(John) \vee King(Richard)$ there is no single binding to $x$that makes the query $\exists x\ King(x)$ true, even though the query is in fact true.

Substitution
Binding list

8.3.2 The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent."

Clearly, the objects in our domain are people. Unary predicates include *Male* and *Female*, among others. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: *Parent*, *Sibling*, *Brother*, *Sister*, *Child*, *Daughter*, *Son*, *Spouse*, *Wife*, *Husband*, *Grandparent*, *Grandchild*, *Cousin*, *Aunt*, and *Uncle*. We use functions for *Mother* and *Father*, because every person has exactly one of each of these, biologically (although we could introduce additional functions for adoptive mothers, surrogate mothers, etc.).

We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's parent who is female:

$$\forall m,c\ \ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m,c).$$

One's husband is one's male spouse:

$$\forall w,h\ \ Husband(h,w) \Leftrightarrow Male(h) \wedge Spouse(h,w).$$

Parent and child are inverse relations:

$$\forall p,c\ \ Parent(p,c) \Leftrightarrow Child(c,p).$$

A grandparent is a parent of one's parent:

$$\forall g,c \ \ Grandparent(g,c) \Leftrightarrow \exists p \ \ Parent(g,p) \wedge Parent(p,c) .$$

A sibling is another child of one's parent:

$$\forall x,y \ \ Sibling(x,y) \Leftrightarrow x \neq y \wedge \exists p \ \ Parent(p,x) \wedge Parent(p,y) .$$

We could go on for several more pages like this, and Exercise 8.KINS asks you to do just that.

Each of these sentences can be viewed as an **axiom** of the kinship domain, as explained in Section 7.1. Axioms are commonly associated with purely mathematical domains—we will see some axioms for numbers shortly—but they are needed in all domains. They provide the basic factual information from which useful conclusions can be derived. Our kinship axioms are also **definitions**; they have the form $\forall x,y \ \ P(x,y) \Leftrightarrow \ldots$. The axioms define the *Mother* function and the *Husband*, *Male*, *Parent*, *Grandparent*, and *Sibling* predicates in terms of other predicates. Our definitions "bottom out" at a basic set of predicates (*Child*, *Female*, etc.) in terms of which the others are ultimately defined. Definition

This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions. Notice that there is not necessarily a unique set of primitive predicates; we could equally well have used *Parent* instead of *Child*. In some domains, as we show, there is no clearly identifiable basic set.

Not all logical sentences about a domain are axioms. Some are **theorems**—that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric: Theorem

$$\forall x,y \ \ Sibling(x,y) \Leftrightarrow Sibling(y,x) .$$

Is this an axiom or a theorem? In fact, it is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return *true*.

From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time, rather like a physicist having to rederive the rules of calculus for every new problem.

Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully. For example, there is no obvious definitive way to complete the sentence

$$\forall x \ \ Person(x) \Leftrightarrow \ldots$$

Fortunately, first-order logic allows us to make use of the *Person* predicate without completely defining it. Instead, we can write partial specifications of properties that every person has and properties that make something a person:

$$\forall x \ \ Person(x) \Rightarrow \ldots$$
$$\forall x \ \ldots \Rightarrow Person(x) .$$

Axioms can also be "just plain facts," such as $Male(Jim)$ and $Spouse(Jim, Laura)$. Such facts form the descriptions of specific problem instances, enabling specific questions to be

answered. If all goes well, the answers to these questions will then be theorems that follow from the axioms.

Often, one finds that the expected answers are not forthcoming—for example, from $Spouse(Jim, Laura)$ one expects (under the laws of many countries) to be able to infer that $\neg Spouse(George, Laura)$; but this does not follow from the axioms given earlier—even after we add $Jim \neq George$ as suggested in Section 8.2.8. This is a sign that an axiom is missing. Exercise 8.HILL asks the reader to supply it.

8.3.3 Numbers, sets, and lists

Natural numbers

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of **natural numbers** or nonnegative integers. We need a predicate $NatNum$ that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor). The **Peano axioms** define natural numbers and addition.[8] Natural numbers are defined recursively:

Peano axioms

$$NatNum(0).$$
$$\forall n \; NatNum(n) \Rightarrow NatNum(S(n)).$$

That is, 0 is a natural number, and for every object n, if n is a natural number, then $S(n)$ is a natural number. So the natural numbers are 0, $S(0)$, $S(S(0))$, and so on. We also need axioms to constrain the successor function:

$$\forall n \; 0 \neq S(n).$$
$$\forall m, n \; m \neq n \Rightarrow S(m) \neq S(n).$$

Now we can define addition in terms of the successor function:

$$\forall m \; NatNum(m) \Rightarrow +(0, m) = m.$$
$$\forall m, n \; NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m), n) = S(+(m, n)).$$

The first of these axioms says that adding 0 to any natural number m gives m itself. Notice the use of the binary function symbol "+" in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using **infix** notation. (The notation we have used for first-order logic is called **prefix**.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so the second axiom becomes

Infix

Prefix

$$\forall m, n \; NatNum(m) \wedge NatNum(n) \Rightarrow (m + 1) + n = (m + n) + 1.$$

This axiom reduces addition to repeated application of the successor function.

Syntactic sugar

The use of infix notation is an example of **syntactic sugar**, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be "desugared" to produce an equivalent sentence in ordinary first-order logic. Another example is using square brackets rather than parentheses to make it easier to see what left bracket matches with what right bracket. Yet another example is collapsing quantifiers: replacing $\forall x \; \forall y \; P(x, y)$ with $\forall x, y \; P(x, y)$.

Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

---

[8] The Peano axioms also include the principle of induction, which is a sentence of second-order logic rather than of first-order logic. The importance of this distinction is explained in Chapter 9.

The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to define number theory in terms of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets from elements or from operations on other sets. We will want to know whether an element is a member of a set and we will want to distinguish sets from objects that are not sets. Set

We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{\}$. There is one unary predicate, *Set*, which is true of sets. The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset of s_2, possibly equal to s_2). The binary functions are $s_1 \cap s_2$ (intersection), $s_1 \cup s_2$ (union), and $Add(x,s)$ (the set resulting from adding element x to set s). One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adding something to a set:
$$\forall s \ Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \ Set(s_2) \wedge s = Add(x, s_2)) .$$
2. The empty set has no elements added into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element:
$$\neg \exists x, s \ Add(x,s) = \{\} .$$
3. Adding an element already in the set has no effect:
$$\forall x, s \ x \in s \Leftrightarrow s = Add(x,s) .$$
4. The only members of a set are the elements that were added into it. We express this recursively, saying that x is a member of s if and only if s is equal to some element y added to some set s_2, where either y is the same as x or x is a member of s_2:
$$\forall x, s \ x \in s \Leftrightarrow \exists y, s_2 \ (s = Add(y, s_2) \wedge (x = y \vee x \in s_2)) .$$
5. A set is a subset of another set if and only if all of the first set's members are members of the second set:
$$\forall s_1, s_2 \ s_1 \subseteq s_2 \Leftrightarrow (\forall x \ x \in s_1 \Rightarrow x \in s_2) .$$
6. Two sets are equal if and only if each is a subset of the other:
$$\forall s_1, s_2 \ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1) .$$
7. An object is in the intersection of two sets if and only if it is a member of both sets:
$$\forall x, s_1, s_2 \ x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2) .$$
8. An object is in the union of two sets if and only if it is a member of either set:
$$\forall x, s_1, s_2 \ x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2) .$$

Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list. We can use the vocabulary of Lisp for lists: *Nil* is the constant list with no elements; *Cons*, *Append*, *First*, and *Rest* are functions; and *Find* is the predicate that does for lists what *Member* does for sets. *List* is a predicate that is true only of lists. As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is $[]$. The term $Cons(x, Nil)$ (i.e., the list containing the element x followed by nothing) is written as $[x]$. A list of several elements, such as $[A, B, C]$, corresponds to the nested term $Cons(A, Cons(B, Cons(C, Nil)))$. Exercise 8.LIST asks you to write out the axioms for lists. List

8.3.4 The wumpus world

Some propositional logic axioms for the wumpus world were given in Chapter 7. The first-order axioms in this section are much more concise, capturing in a natural way exactly what we want to say.

Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be

$$Percept([Stench, Breeze, Glitter, None, None], 5) .$$

Here, *Percept* is a binary predicate, and *Stench* and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

$$Turn(Right),\ Turn(Left),\ Forward,\ Shoot,\ Grab,\ Climb .$$

To determine which is best, the agent program executes the query

$$\text{ASKVARS}(KB, BestAction(a, 5)) ,$$

which returns a binding list such as $\{a/Grab\}$. The agent program can then return *Grab* as the action to take. The raw percept data implies certain facts about the current state. For example:

$$\begin{aligned}
&\forall t,s,g,w,c\ \ Percept([s,Breeze,g,w,c],t) \Rightarrow Breeze(t)\\
&\forall t,s,g,w,c\ \ Percept([s,None,g,w,c],t) \Rightarrow \neg Breeze(t)\\
&\forall t,s,b,w,c\ \ Percept([s,b,Glitter,w,c],t) \Rightarrow Glitter(t)\\
&\forall t,s,b,w,c\ \ Percept([s,b,None,w,c],t) \Rightarrow \neg Glitter(t)
\end{aligned}$$

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 25. Notice the quantification over time t. In propositional logic, we would need copies of each sentence for each time step.

Simple "reflex" behavior can also be implemented by quantified implication sentences. For example, we have

$$\forall t\ \ Glitter(t) \Rightarrow BestAction(Grab, t) .$$

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion $BestAction(Grab, 5)$—that is, *Grab* is the right thing to do.

We have represented the agent's inputs and outputs; now it is time to represent the environment itself. Let us begin with objects. Obvious candidates are squares, pits, and the wumpus. We could name each square—$Square_{1,2}$ and so on—but then the fact that $Square_{1,2}$ and $Square_{1,3}$ are adjacent would have to be an "extra" fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term $[1,2]$. Adjacency of any two squares can be defined as

$$\begin{aligned}
&\forall x,y,a,b\ \ Adjacent([x,y],[a,b]) \Leftrightarrow\\
&\qquad (x = a \wedge (y = b-1 \vee y = b+1)) \vee (y = b \wedge (x = a-1 \vee x = a+1)) .
\end{aligned}$$

We could name each pit, but this would be inappropriate for a different reason: there is no

reason to distinguish among pits.[9] It is simpler to use a unary predicate *Pit* that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant *Wumpus* is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint).

The agent's location changes over time, so we write $At(Agent, s, t)$ to mean that the agent is at square s at time t. We can fix the wumpus to a specific location forever with $\forall t\, At(Wumpus, [1,3], t)$. We can then say that objects can be at only one location at a time:

$$\forall x, s_1, s_2, t\ \ At(x, s_1, t) \wedge At(x, s_2, t) \Rightarrow s_1 = s_2\,.$$

Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$$\forall s, t\ \ At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s)\,.$$

It is useful to know that a *square* is breezy because we know that the pits cannot move about. Notice that *Breezy* has no time argument.

Having discovered which places are breezy (or smelly) and, very importantly, *not* breezy (or *not* smelly), the agent can deduce where the pits are (and where the wumpus is). Whereas propositional logic necessitates a separate axiom for each square (see R_2 and R_3 on page 220) and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

$$\forall s\ \ Breezy(s) \Leftrightarrow \exists r\ \ Adjacent(r, s) \wedge Pit(r)\,. \tag{8.4}$$

Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow (Equation (7.2) on page 240) becomes

$$\forall t\ \ HaveArrow(t+1) \Leftrightarrow (HaveArrow(t) \wedge \neg Action(Shoot, t))\,.$$

From these two example sentences, we can see that the first-order logic formulation is no less concise than the original English-language description given in Chapter 7. The reader is invited to construct analogous axioms for the agent's location and orientation; in these cases, the axioms quantify over both space and time. As in the case of propositional state estimation, an agent can use logical inference with axioms of this kind to keep track of aspects of the world that are not directly observed. Chapter 11 goes into more depth on the subject of first-order successor-state axioms and their uses for constructing plans.

8.4 Knowledge Engineering in First-Order Logic

The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. This section describes the general process of knowledge-base construction—a process called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We illustrate the knowledge engineering process in an electronic circuit domain. The approach we take is suitable for developing *special-purpose* knowledge bases whose domain is carefully circumscribed and

Knowledge engineering

[9] Similarly, most of us do not name each bird that flies overhead as it migrates to warmer regions in winter. An ornithologist wishing to study migration patterns, survival rates, and so on *does* name each bird, by means of a ring on its leg, because individual birds must be tracked.

whose range of queries is known in advance. *General-purpose* knowledge bases, which cover a broad range of human knowledge and are intended to support tasks such as natural language understanding, are discussed in Chapter 10.

8.4.1 The knowledge engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. *Identify the questions.* The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions, or is it required only to answer questions about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents in Chapter 2.
2. *Assemble the relevant knowledge.* The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

   Knowledge acquisition

   For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.) For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.
3. *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering *style*. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

   Ontology
4. *Encode general knowledge about the domain.* The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. *Encode a description of the problem instance.* If the ontology is well thought out, this step is easy. It involves writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is given sentences in the same way that traditional programs are given input data.

6. *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.
7. *Debug and evaluate the knowledge base.* Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes a diagnostic rule (see Exercise 8.WUMD) for finding the wumpus,

$$\forall s \; Smelly(s) \Rightarrow Adjacent(Home(Wumpus), s)\,,$$

instead of the biconditional, then the agent will never be able to prove the *absence* of wumpuses. Incorrect axioms can be identified because they are false statements about the world. For example, the sentence

$$\forall x \; NumOfLegs(x, 4) \Rightarrow Mammal(x)$$

is false for reptiles, amphibians, and tables. *The falsehood of this sentence can be determined independently of the rest of the knowledge base.* In contrast, a typical error in a program looks like this:

```
offset = position + 1.
```

It is impossible to tell whether `offset` should be `position` or `position + 1` without understanding the surrounding context.

When you get to the point where there are no obvious errors in your knowledge base, it is tempting to declare success. But unless there are obviously no errors, it is better to formally evaluate your system by running it on a test suite of queries and measuring how many you get right. Without objective measurement, it is too easy to convince yourself that the job is done. To understand this seven-step process better, we now apply it to an extended example—the domain of electronic circuits.

8.4.2 The electronic circuits domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 8.6. We follow the seven-step process for knowledge engineering.

Identify the questions

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 8.6 actually add properly? If all the inputs are high, what is the output of gate A2? Questions about the circuit's structure are also interesting. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section. There are more detailed levels of analysis, including those related to timing delays, circuit area, power consumption, production cost, and so on. Each of these levels would require additional knowledge.

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.

To reason about functionality and connectivity, we do not need to talk about the wires themselves, the paths they take, or the junctions where they come together. All that matters is the connections between terminals—we can say that one output terminal is connected to another input terminal without having to say what actually connects them. Other factors such as the size, shape, color, or cost of the various components are irrelevant to our analysis.

If our purpose were something other than verifying designs at the gate level, the ontology would be different. For example, if we were interested in debugging faulty circuits, then it would probably be a good idea to include the wires in the ontology, because a faulty wire can corrupt the signal flowing along it. For resolving timing faults, we would need to include gate delays. If we were interested in designing a product that would be profitable, then the cost of the circuit and its speed relative to other products on the market would be important.

Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. First, we need to be able to distinguish gates from each other and from other objects. Each gate is represented as an object named by a constant, about which we assert that it is a gate with, say, $Gate(X_1)$. The behavior of each gate is determined by its type: one of the constants AND, OR, XOR, or NOT. Because a gate has exactly one type, a function is appropriate: $Type(X_1) = XOR$. Circuits, like gates, are identified by a predicate: $Circuit(C_1)$.

Next we consider terminals, which are identified by the predicate $Terminal(x)$. A circuit can have one or more input terminals and one or more output terminals. We use the function

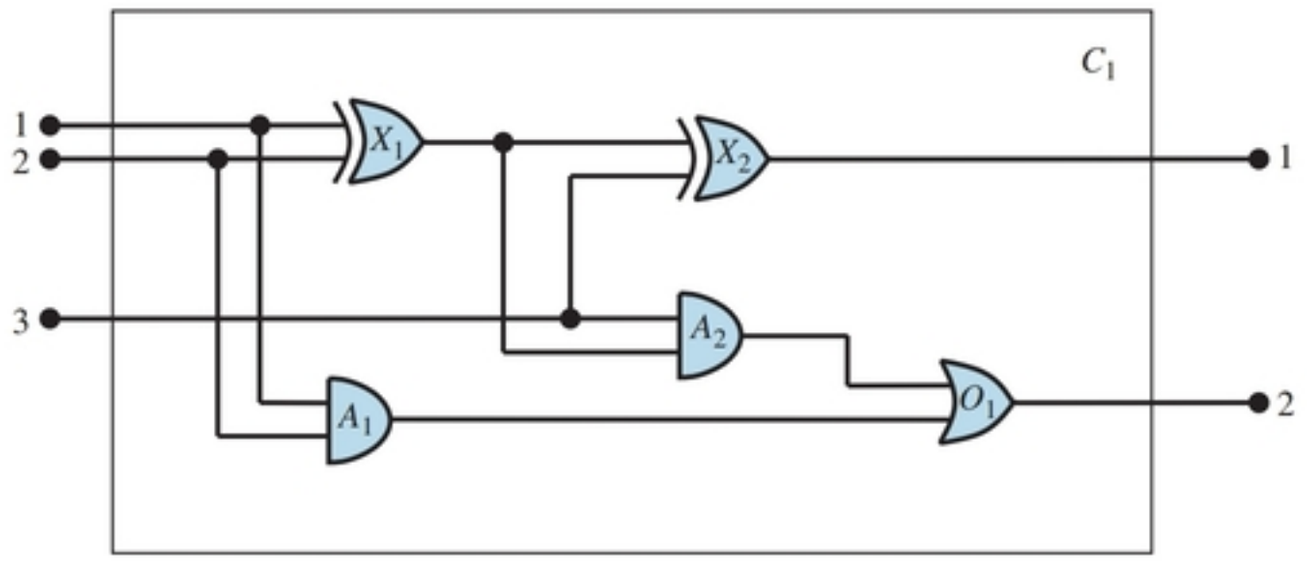

Figure 8.6 A digital circuit C_1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

$In(1,X_1)$ to denote the first input terminal for circuit X_1. A similar function $Out(n,c)$ is used for output terminals. The function $Arity(c,i,j)$ says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a predicate, *Connected*, which takes two terminals as arguments, as in $Connected(Out(1,X_1),In(1,X_2))$.

Finally, we need to know whether a signal is on or off. One possibility is to use a unary predicate, $On(t)$, which is true when the signal at a terminal is on. This makes it a little difficult, however, to pose questions such as "What are all the possible values of the signals at the output terminals of circuit C_1 ?" We therefore introduce as objects two signal values, 1 and 0, representing "on" and "off" respectively, and a function $Signal(t)$ that denotes the signal value for the terminal t.

Encode general knowledge of the domain

One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need:

1. If two terminals are connected, then they have the same signal:
$$\forall t_1,t_2 \;\; Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1,t_2) \Rightarrow Signal(t_1) = Signal(t_2) .$$
2. The signal at every terminal is either 1 or 0:
$$\forall t \;\; Terminal(t) \Rightarrow Signal(t) = 1 \vee Signal(t) = 0 \;\; .$$
3. *Connected* is commutative:
$$\forall t_1,t_2 \;\; Connected(t_1,t_2) \Leftrightarrow Connected(t_2,t_1) .$$
4. There are four types of gates:
$$\forall g \;\; Gate(g) \wedge k = Type(g) \Rightarrow k = AND \vee k = OR \vee k = XOR \vee k = NOT .$$
5. An AND gate's output is 0 if and only if any of its inputs is 0:
$$\forall g \;\; Gate(g) \wedge Type(g) = AND \Rightarrow Signal(Out(1,g)) = 0 \Leftrightarrow \exists n \;\; Signal(In(n,g)) = 0 .$$
6. An OR gate's output is 1 if and only if any of its inputs is 1:
$$\forall g \;\; Gate(g) \wedge Type(g) = OR \Rightarrow Signal(Out(1,g)) = 1 \Leftrightarrow \exists n \;\; Signal(In(n,g)) = 1 .$$
7. An XOR gate's output is 1 if and only if its inputs are different:
$$\forall g \;\; Gate(g) \wedge Type(g) = XOR \Rightarrow Signal(Out(1,g)) = 1 \Leftrightarrow Signal(In(1,g)) \neq Signal(In(2,g)) .$$
8. A NOT gate's output is different from its input:
$$\forall g \;\; Gate(g) \wedge Type(g) = NOT \Rightarrow Signal(Out(1,g)) \neq Signal(In(1,g)) .$$
9. The gates (except for NOT) have two inputs and one output.
$$\forall g \;\; Gate(g) \wedge Type(g) = NOT \Rightarrow Arity(g,1,1) \;\; .$$
$$\forall g \;\; Gate(g) \wedge k = Type(g) \wedge (k = AND \vee k = OR \vee k = XOR) \Rightarrow Arity(g,2,1)$$
10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:
$$\forall c,i,j \;\; Circuit(c) \wedge Arity(c,i,j) \Rightarrow \forall n \;\; (n \leq i \Rightarrow Terminal(In(n,c))) \wedge (n > i \Rightarrow In(n,c) = Nothing) \wedge$$
$$\forall n \;\; (n \leq j \Rightarrow Terminal(Out(n,c))) \wedge (n > j \Rightarrow Out(n,c) = Nothing)$$

11. Gates, terminals, and signals are all distinct.
$$\forall g,t,s \;\; Gate(g) \wedge Terminal(t) \wedge Signal(s) \Rightarrow g \neq t \wedge g \neq s \wedge t \neq s\,.$$
12. Gates are circuits.
$$\forall g \;\; Gate(g) \Rightarrow Circuit(g)$$

Encode the specific problem instance

The circuit shown in Figure 8.6 is encoded as circuit C_1 with the following description. First we categorize the circuit and its component gates:

$$\begin{aligned}
&Circuit(C_1) \wedge Arity(C_1,3,2)\\
&Gate(X_1) \wedge Type(X_1) = XOR\\
&Gate(X_2) \wedge Type(X_2) = XOR\\
&Gate(A_1) \wedge Type(A_1) = AND\\
&Gate(A_2) \wedge Type(A_2) = AND\\
&Gate(O_1) \wedge Type(O_1) = OR\,.
\end{aligned}$$

Then we show the connections between them:

$$\begin{aligned}
&Connected(Out(1,X_1),In(1,X_2)) \quad Connected(In(1,C_1),In(1,X_1))\\
&Connected(Out(1,X_1),In(2,A_2)) \quad Connected(In(1,C_1),In(1,A_1))\\
&Connected(Out(1,A_2),In(1,O_1)) \quad Connected(In(2,C_1),In(2,X_1))\\
&Connected(Out(1,A_1),In(2,O_1)) \quad Connected(In(2,C_1),In(2,A_1))\\
&Connected(Out(1,X_2),Out(1,C_1)) \quad Connected(In(3,C_1),In(2,X_2))\\
&Connected(Out(1,O_1),Out(2,C_1)) \quad Connected(In(3,C_1),In(1,A_2))\,.
\end{aligned}$$

Pose queries to the inference procedure

What combinations of inputs would cause the first output of C_1 (the sum bit) to be 0 and the second output of C_1 (the carry bit) to be 1?

$$\begin{aligned}
\exists i_1,i_2,i_3 \;\; &Signal(In(1,C_1)) = i_1 \wedge Signal(In(2,C_1)) = i_2 \wedge Signal(In(3,C_1)) = i_3\\
&\wedge Signal(Out(1,C_1)) = 0 \wedge Signal(Out(2,C_1)) = 1\,.
\end{aligned}$$

The answers are substitutions for the variables i_1, i_2, and i_3 such that the resulting sentence is entailed by the knowledge base. ASKVARS will give us three such substitutions:

$$\{i_1/1, i_2/1, i_3/0\} \quad \{i_1/1, i_2/0, i_3/1\} \quad \{i_1/0, i_2/1, i_3/1\}\,.$$

What are the possible sets of values of all the terminals for the adder circuit?

$$\begin{aligned}
\exists i_1,i_2,i_3,o_1,o_2 \;\; &Signal(In(1,C_1)) = i_1 \wedge Signal(In(2,C_1)) = i_2\\
&\wedge Signal(In(3,C_1)) = i_3 \wedge Signal(Out(1,C_1)) = o_1 \wedge Signal(Out(2,C_1)) = o_2\,.
\end{aligned}$$

This final query will return a complete input–output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification**. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out. (See Exercise 8.ADDR.) Many domains are amenable to the same kind of structured knowledge-base development, in which more complex concepts are defined on top of simpler concepts.

Circuit verification

Debug the knowledge base

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge. For example, suppose we fail to read Section 8.2.8 and hence forget to assert that $1 \neq 0$. Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. We can pinpoint the problem by asking for the outputs of each gate. For example, we can ask

$$\exists i_1, i_2, o \;\; Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge Signal(Out(1, X_1)) = o \,,$$

which reveals that no outputs are known at X_1 for the input cases 10 and 01. Then, we look at the axiom for XOR gates, as applied to X_1:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; Signal(In(1, X_1)) \neq Signal(In(2, X_1)) \,.$$

If the inputs are known to be, say, 1 and 0, then this reduces to

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; 1 \neq 0 \,.$$

Now the problem is apparent: the system is unable to infer that $Signal(Out(1, X_1)) = 1$, so we need to tell it that $1 \neq 0$.

Summary

This chapter has introduced **first-order logic**, a representation language that is far more powerful than propositional logic. The important points are as follows:

- Knowledge representation languages should be declarative, compositional, expressive, context independent, and unambiguous.
- Logics differ in their **ontological commitments** and **epistemological commitments**. While propositional logic commits only to the existence of facts, first-order logic commits to the existence of objects and relations and thereby gains expressive power, appropriate for domains such as the wumpus world and electronic circuits.
- Both propositional logic and first-order logic share a difficulty in representing vague propositions. This difficulty limits their applicability in domains that require personal judgments, like politics or cuisine.
- The syntax of first-order logic builds on that of propositional logic. It adds terms to represent objects, and has universal and existential quantifiers to construct assertions about all or some of the possible values of the quantified variables.
- A **possible world**, or **model**, for first-order logic includes a set of objects and an **interpretation** that maps constant symbols to objects, predicate symbols to relations among objects, and function symbols to functions on objects.
- An atomic sentence is true only when the relation named by the predicate holds between the objects named by the terms. **Extended interpretations**, which map quantifier variables to objects in the model, define the truth of quantified sentences.
- Developing a knowledge base in first-order logic requires a careful process of analyzing the domain, choosing a vocabulary, and encoding the axioms required to support the desired inferences.

Bibliographical and Historical Notes

Although Aristotle's logic dealt with generalizations over objects, it fell far short of the expressive power of first-order logic. A major barrier to its further development was its concentration on one-place predicates to the exclusion of many-place relational predicates. The first systematic treatment of relations was given by Augustus De Morgan (1864), who cited the following example to show the sorts of inferences that Aristotle's logic could not handle: "All horses are animals; therefore, the head of a horse is the head of an animal." This inference is inaccessible to Aristotle because any valid rule that can support this inference must first analyze the sentence using the two-place predicate "x is the head of y." The logic of relations was studied in depth by Charles Sanders Peirce (Peirce, 1870; Misak, 2004).

True first-order logic dates from the introduction of quantifiers in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation"). Peirce (1883) also developed first-order logic independently of Frege, although slightly later. Frege's ability to nest quantifiers was a big step forward, but he used an awkward notation. The present notation for first-order logic is due substantially to Giuseppe Peano (1889), but the semantics is virtually identical to Frege's. Oddly enough, Peano's axioms were due in large measure to Grassmann (1861) and Dedekind (1888).

Leopold Löwenheim (1915) gave a systematic treatment of model theory for first-order logic, including the first proper treatment of the equality symbol. Löwenheim's results were further extended by Thoralf Skolem (1920). Alfred Tarski (1935, 1956) gave an explicit definition of truth and model-theoretic satisfaction in first-order logic, using set theory.

John McCarthy (1958) was primarily responsible for the introduction of first-order logic as a tool for building AI systems. The prospects for logic-based AI were advanced significantly by Robinson's (1965) development of resolution, a complete procedure for first-order inference. The logicist approach took root at Stanford University. Cordell Green (1969a, 1969b) developed a first-order reasoning system, QA3, leading to the first attempts to build a logical robot at SRI (Fikes and Nilsson, 1971). First-order logic was applied by Zohar Manna and Richard Waldinger (1971) for reasoning about programs and later by Michael Genesereth (1984) for reasoning about circuits. In Europe, logic programming (a restricted form of first-order reasoning) was developed for linguistic analysis (Colmerauer *et al.*, 1973) and for general declarative systems (Kowalski, 1974). Computational logic was also well entrenched at Edinburgh through the LCF (Logic for Computable Functions) project (Gordon *et al.*, 1979). These developments are chronicled further in Chapters 9 and 10.

Practical applications built with first-order logic include a system for evaluating the manufacturing requirements for electronic products (Mannion, 2002), a system for reasoning about policies for file access and digital rights management (Halpern and Weissman, 2008), and a system for the automated composition of Web services (McIlraith and Zeng, 2001).

Reactions to the Whorf hypothesis (Whorf, 1956) and the problem of language and thought in general, appear in multiple books (Pullum, 1991; Pinker, 2003) including the seemingly opposing titles *Why the World Looks Different in Other Languages* (Deutscher, 2010) and *Why The World Looks the Same in Any Language* (McWhorter, 2014) (although both authors agree that there are differences and the differences are small). The "theory" theory (Gopnik and Glymour, 2002; Tenenbaum *et al.*, 2007) views children's learning about the world as analogous to the construction of scientific theories. Just as the predictions of a ma-

chine learning algorithm depend strongly on the vocabulary supplied to it, so will the child's formulation of theories depend on the linguistic environment in which learning occurs.

There are a number of good introductory texts on first-order logic, including some by leading figures in the history of logic: Alfred Tarski (1941), Alonzo Church (1956), and W.V. Quine (1982) (which is one of the most readable). Enderton (1972) gives a more mathematically oriented perspective. A highly formal treatment of first-order logic, along with many more advanced topics in logic, is provided by Bell and Machover (1977). Manna and Waldinger (1985) give a readable introduction to logic from a computer science perspective, as do Huth and Ryan (2004), who concentrate on program verification. Barwise and Etchemendy (2002) take an approach similar to the one used here. Smullyan (1995) presents results concisely, using the tableau format. Gallier (1986) provides an extremely rigorous mathematical exposition of first-order logic, along with a great deal of material on its use in automated reasoning. *Logical Foundations of Artificial Intelligence* (Genesereth and Nilsson, 1987) is both a solid introduction to logic and the first systematic treatment of logical agents with percepts and actions, and there are two good handbooks: van Bentham and ter Meulen (1997) and Robinson and Voronkov (2001). The journal of record for the field of pure mathematical logic is the *Journal of Symbolic Logic*, whereas the *Journal of Applied Logic* deals with concerns closer to those of artificial intelligence.