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.

In Chapter 7, we showed how a knowledge-based agent could represent the world in which it operates and deduce what actions to take. We used propositional logic as our representation language because it sufficed to illustrate the basic concepts of logic and knowledge-based agents. Unfortunately, propositional logic is too puny a language to represent knowledge of complex environments in a concise way. In this chapter, we examine **first-order logic**,[1] which is sufficiently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages and has been studied intensively for many decades. 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 will discuss the nature of representation languages. Our discussion will motivate the development of first-order logic, a much more expressive language than the propositional logic introduced in Chapter 7. We will look at propositional logic and at other kinds of languages to understand what works and what fails. Our discussion will be cursory, compressing centuries of thought, trial, and error into a few paragraphs.

Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent, in a direct sense, only computational processes. Data structures within programs can 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]. (Such representations might be considered *ad hoc*; database systems were developed precisely to provide a more general, domain-independent way to

[1] Also called **first-order predicate calculus**, sometimes abbreviated as **FOL** or **FOPC**.

store and retrieve facts.) What programming languages lack is any 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.

A second drawback of data structures in programs (and of databases, for that matter) 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 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 example, "$S_{1,4} \wedge S_{1,2}$" 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. Clearly, noncompositionality makes life much more difficult for the reasoning system.

COMPOSITIONALITY

As we saw in Chapter 7, propositional logic lacks the expressive power to describe an environment with many objects *concisely*. 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 somehow make it possible to describe the environment concisely.

A moment's thought suggests that 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 (including logic, mathematics, and the language of diagrams). There is a long tradition in linguistics and the philosophy of language that views natural language as essentially a declarative knowledge representation language and attempts to pin down its formal semantics. Such a research program, if successful, would be of great value to artificial intelligence because it would allow a natural language (or some derivative) to be used within representation and reasoning systems.

The modern view of natural language is that it serves a somewhat different purpose, namely 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!" encoded 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 are also noncompositional—the meaning of a sentence such as "Then she saw it" can depend on a context constructed by many preceding and succeeding sentences. Finally, natural languages suffer from **ambiguity**, which would cause difficulties for thinking. 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."

Our approach will be to 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 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:

OBJECTS
RELATIONS
FUNCTIONS

- 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 . . .

PROPERTIES

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; Properties: evil, king.

The language of **first-order logic**, whose syntax and semantics we will define in the next section, is built around objects and relations. It has been so 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."

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*.

ONTOLOGICAL COMMITMENT

THE LANGUAGE OF THOUGHT

Philosophers and psychologists have long pondered how it is that humans and other animals represent knowledge. It is clear that the evolution of natural language has played an important role in developing this ability in humans. On the other hand, much psychological evidence suggests that humans do not employ language directly in their internal representations. For example, which of the following two phrases formed the opening of Section 8.1?

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

Wanner (1974) found that subjects made the right choice in such tests 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 process the words to form some kind of nonverbal representation, which we call **memory**.

The exact mechanism by which language enables and shapes the representation of ideas in humans remains a fascinating question. The famous **Sapir–Whorf hypothesis** claims that the language we speak profoundly influences the way in which we think and make decisions, in particular by setting up the category structure by which we divide up the world into different sorts of objects. Whorf (1956) claimed that Eskimos have many words for snow and thus experience snow in a different way from speakers of other languages. Some linguists dispute the factual basis for this claim—Pullum (1991) argues that Inuit, Yupik, and other related languages seem to have about the same number of words for snow-related concepts as English—while others support it (Fortescue, 1984). It seems unarguably true that populations having greater familiarity with some aspects of the world develop much more detailed vocabularies—for example, field entomologists divide what most of us call *beetles* into hundreds of thousands of species and are personally familiar with many of these. (The evolutionary biologist J. B. S. Haldane once complained of "An inordinate fondness for beetles" on the part of the Creator.) Moreover, expert skiers have many terms for snow—powder, chowder, mashed potatoes, crud, corn, cement, crust, sugar, asphalt, corduroy, fluff, glop, and so on—that represent distinctions unfamiliar to the lay person. What is unclear is the direction of causality—do skiers become aware of the distinctions only by learning the words, or do the distinctions emerge from individual experience and become matched with the labels current in the community? This question is especially important in the study of child development. As yet, we have little understanding of the extent to which learning language and learning to think are intertwined. For example, does the knowledge of a name for a concept, such as *bachelor*, make it easier to construct and reason with more complex concepts that include that name, such as *eligible bachelor*?

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.[2] First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. Special-purpose logics make still further ontological commitments; for example,

TEMPORAL LOGIC

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

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.

EPISTEMOLOGICAL COMMITMENTS

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. Systems using **probability theory**, on the other hand, can have any *degree of belief*, ranging from 0 (total disbelief) to 1 (total belief).[3] For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75. The ontological and epistemological commitments of five different logics are summarized in Figure 8.1.

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.

In the next section, we will launch into the details of first-order logic. Just as a student of physics requires some familiarity with mathematics, a student of AI must develop a talent for working with logical notation. On the other hand, it is also important *not* to get too concerned with the *specifics* of logical notation—after all, there are dozens of different versions. The main things to keep hold of are how the language facilitates concise representations and how its semantics leads to sound reasoning procedures.

[2] In contrast, facts in **fuzzy logic** have a **degree of truth** between 0 and 1. For example, the sentence "Vienna is a large city" might be true in our world only to degree 0.6.

[3] 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.

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.

Models for first-order logic

Recall from Chapter 7 that the models of a logical language are the formal structures that constitute the possible worlds under consideration. Models for propositional logic are just sets of truth values for the proposition symbols. Models for first-order logic are more interesting.

DOMAIN

DOMAIN ELEMENTS

First, they have objects in them! The **domain** of a model is the set of objects it contains; these objects are sometimes called **domain elements**. 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.

TUPLES

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

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

(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

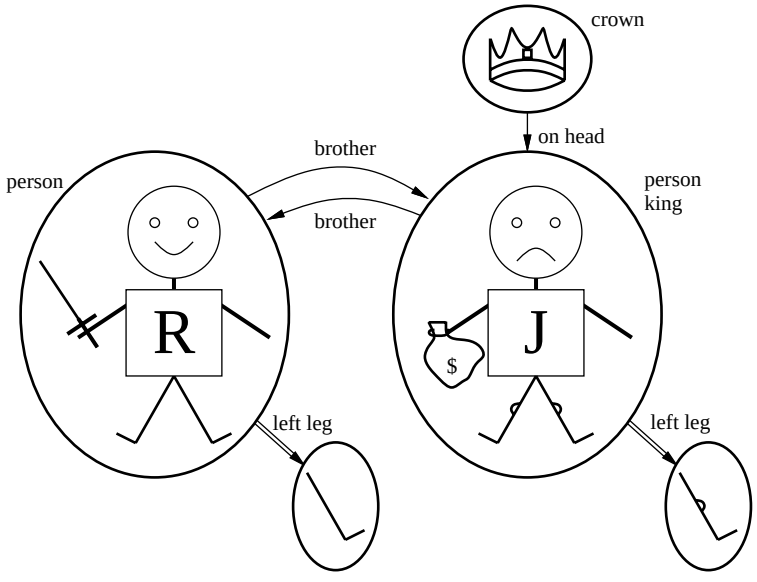


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

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 that includes the following mappings:

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

TOTAL FUNCTIONS

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.

Symbols and interpretations

We turn now to the syntax of the language. The impatient reader can obtain a complete description from the formal grammar of first-order logic 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 SYMBOLS
PREDICATE SYMBOLS
FUNCTION SYMBOLS
ARITY

The semantics must relate sentences to models in order to determine truth. For this to happen, we need 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 we will call the **intended interpretation**—is as follows:

INTERPRETATION
INTENDED INTERPRETATION

- $Richard$ 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); $OnHead$ refers to the "on head" relation that holds between the crown and King John; $Person$, $King$, and $Crown$ refer to the sets of objects that are persons, kings, and crowns.
- $LeftLeg$ refers to the "left leg" function, that is, the mapping given in Equation (8.2).

There are many other possible interpretations relating these symbols to this particular model. 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

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \\
& | & \textbf{(}\ \mathit{Sentence}\ \mathit{Connective}\ \mathit{Sentence}\ \textbf{)} \\
& | & \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
& | & \neg\ \mathit{Sentence} \\
\\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate}(\mathit{Term}, \ldots) \mid\ \mathit{Term} = \mathit{Term} \\
\\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term}, \ldots) \\
& | & \mathit{Constant} \\
& | & \mathit{Variable} \\
\\
\mathit{Connective} & \rightarrow & \Rightarrow \mid\ \wedge \mid\ \vee \mid \Leftrightarrow \\
\mathit{Quantifier} & \rightarrow & \forall \mid\ \exists \\
\mathit{Constant} & \rightarrow & \mathit{A} \mid\ \mathit{X}_1 \mid\ \mathit{John} \mid\ \cdots \\
\mathit{Variable} & \rightarrow & a \mid\ x \mid\ s \mid\ \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{Before} \mid\ \mathit{HasColor} \mid\ \mathit{Raining} \mid\ \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid\ \mathit{LeftLeg} \mid\ \cdots
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form. (See page 984 if you are not familiar with this notation.) The syntax is strict about parentheses; the comments about parentheses and operator precedence on page 205 apply equally to first-order logic.

constant symbols $Richard$ and $John$. 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. 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.

The truth of any sentence is determined by a model and an interpretation for the sentence's symbols. Therefore, entailment, validity, and so on are defined in terms of *all possible models* and *all possible interpretations*. It is important to note that the number of domain elements in each model may be unbounded—for example, the domain elements may be integers or real numbers. Hence, the number of possible models is unbounded, as is the number of interpretations. Checking entailment by the enumeration of all possible models, which works for propositional logic, is not an option for first-order logic. Even if the number of objects is restricted, the number of combinations can be very large. With the symbols in our example, there roughly 10^{25} combinations for a domain with five objects. (See Exercise 8.5.)

Terms

TERM

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, 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 $LeftLeg(John)$. 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 that John must have one) without ever providing a definition of $LeftLeg$. This is something that cannot be done with subroutines in programming languages.[4]

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.

Atomic sentences

Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state facts. An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms:

$$Brother(Richard, John).$$

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.[5] 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).

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

[4] **λ-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. (See Chapter 22.) The `lambda` operator in Lisp 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.

[5] We will usually follow the argument ordering convention that $P(x, y)$ is interpreted as "x is a P of y."

Complex sentences

We can use **logical connectives** to construct more complex sentences, just as in propositional calculus. The semantics of sentences formed with logical connectives is identical to that in the propositional case. Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

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

Quantifiers

QUANTIFIERS

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

$$\forall x \;\; King(x) \Rightarrow Person(x) .$$

$\forall$ 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, $LeftLeg(x)$. A term with no variables is called a **ground term**.

VARIABLE

GROUND TERM

Intuitively, the sentence $\forall x \; P$, where P is any logical expression, says that P is true for every object x. More precisely, $\forall x \; P$ is true in a given model under a given interpretation if P is true in all possible **extended interpretations** constructed from the given interpretation, 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 $\forall x \;\; King(x) \Rightarrow Person(x)$ is true under the original interpretation if the sentence $King(x) \Rightarrow Person(x)$ is true in each of the five extended inter-

pretations. 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), 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 whom the premise is true and saying nothing at all about those individuals for whom the premise is false. Thus, the truth-table entries for $\Rightarrow$ turn 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

$$\forall x \;\; King(x) \wedge Person(x)$$

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 ($\exists$)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe 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 under a given interpretation if P is true in *at least one* extended interpretation that assigns x to a domain element. For our example, this means

that at least one of the following must be 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. Now 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 in any model containing an object for which the premise of the implication is false; hence such sentences really do not say much at all.

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

$$\forall x \;\; \forall y \;\; Brother(x, y) \;\Rightarrow\; Sibling(x, y) \; .$$

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

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

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

$$\forall x \;\; \exists y \;\; Loves(x, y) \; .$$

[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, as we show in Section 8.2.

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

$$\exists y \ \forall x \ \ Loves(x, y) \ .$$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. $\forall x \ (\exists y \ Loves(x, y))$ says that *everyone* has a particular property, namely, the property that somebody loves them. On the other hand, $\exists x \ (\forall y \ Loves(x, y))$ 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

$$\forall x \ \ [Crown(x) \lor (\exists x \ \ Brother(Richard, x))] \ .$$

Here the x in $Brother(Richard, x)$ 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.[7] Another way to think of it is this: $\exists x \ Brother(Richard, x)$ 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 $\exists z \ Brother(Richard, z)$. Because this can be a source of confusion, we will always use different variables.

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:

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

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

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

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:

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

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

[7] It is the potential for interference between quantifiers using the same variable name that motivates the slightly baroque mechanism of extended interpretations in the semantics of quantified sentences. The more intuitively obvious approach of substituting objects for every occurrence of x fails in our example because the x in $Brother(Richard, x)$ would be "captured" by the substitution. Extended interpretations handle this correctly because the inner quantifier's assignment for x overrides the outer quantifier's.

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 make statements to the effect 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 $\neg(x = y)$.

8.3 USING FIRST-ORDER LOGIC

Now that we have defined an expressive logical language, it is time to learn how to use it. The best way to do this is through examples. We have seen some simple sentences illustrating the various aspects of logical syntax; in this section, we will provide more systematic representations of some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

DOMAINS

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

Assertions and queries in first-order logic

ASSERTIONS

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 and that kings are persons:

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

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

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

QUERIES
GOALS

returns $true$. Questions asked using ASK are called **queries** or **goals** (not to be confused with goals as used to describe an agent's desired states). Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two assertions in the preceding paragraph, the query

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

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

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

The answer to this query could be $true$, but this is neither helpful nor amusing. (It is rather like answering "Can you tell me the time?" with "Yes.") A query with existential variables is asking "Is there an x such that . . .," and we solve it by providing such an x. The standard

SUBSTITUTION
BINDING LIST

form for an answer of this sort is a **substitution** or **binding list**, which is a set of variable/term pairs. In this particular case, given just the two assertions, the answer would be $\{x/John\}$. If there is more than one possible answer, a list of substitutions can be returned.

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. We will have two unary predicates, $Male$ and $Female$. Kinship relations—parenthood, brotherhood, marriage, and so on—will be represented by binary predicates: $Parent$, $Sibling$, $Brother$, $Sister$, $Child$, $Daughter$, Son, $Spouse$, $Wife$, $Husband$, $Grandparent$, $Grandchild$, $Cousin$, $Aunt$, and $Uncle$. We will use functions for $Mother$ and $Father$, because every person has exactly one of each of these (at least according to nature's design).

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 female parent:

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

One's husband is one's male spouse:

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

Male and female are disjoint categories:

$$\forall x\ \ Male(x)\ \Leftrightarrow\ \neg Female(x)\ .$$

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 parents:

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

We could go on for several more pages like this, and Exercise 8.11 asks you to do just that.

AXIOM

Each of these sentences can be viewed as an **axiom** of the kinship domain. 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 ($Child$, $Spouse$, and $Female$) in terms of which the others are ultimately defined. This is a very 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 $Parent$, $Spouse$, and $Male$. In some domains, as we will see, there is no clearly identifiable basic set.

DEFINITION

THEOREM

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:

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

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 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. 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 $Male(George)$ and $Spouse(George, Laura)$, one expects to be able to infer $Female(Laura)$; but this does not follow from the axioms given earlier. This is a sign that an axiom is missing. Exercise 8.8 asks you to supply it.

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 will 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:

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

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 will allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so that 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 "de-sugared" to produce an equivalent sentence in ordinary first-order logic.

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.

SETS

The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to build number theory on top of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets by

[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.

adding an element to a set or taking the union or intersection of two sets. We will want to know whether an element is a member of a set and to be able to distinguish sets from objects that are not sets.

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, not necessarily proper, of set s_2). The binary functions are $s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x|s\}$ (the set resulting from adjoining 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 adjoining something to a set:
$$\forall s \;\; Set(s) \;\Leftrightarrow\; (s = \{\}) \vee (\exists x, s_2 \;\; Set(s_2) \wedge s = \{x|s_2\}) \; .$$
2. The empty set has no elements adjoined into it, in other words, there is no way to decompose $EmptySet$ into a smaller set and an element:
$$\neg \exists x, s \;\; \{x|s\} = \{\} \; .$$
3. Adjoining an element already in the set has no effect:
$$\forall x, s \;\; x \in s \;\Leftrightarrow\; s = \{x|s\} \; .$$
4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined with some element y, where either y is the same as x or x is a member of s_2:
$$\forall x, s \;\; x \in s \;\Leftrightarrow\; [\exists y, s_2 \;\; (s = \{y|s_2\} \wedge (x = y \vee x \in s_2))] \; .$$
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

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 $Cons(x, y)$, where y is a nonempty list, is written $[x|y]$. The term $Cons(x, Nil)$, (i.e., the list containing the element x), 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 8.14 asks you to write out the axioms for lists.

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 very 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 will 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:

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

To determine which is best, the agent program constructs a query such as

$$\exists a \;\; BestAction(a, 5) .$$

ASK should solve this query and return a binding list such as $\{a/Grab\}$. The agent program can then return $Grab$ as the action to take, but first it must TELL its own knowledge base that it is performing a $Grab$.

The raw percept data implies certain facts about the current state. For example:

$$\forall t, s, g, m, c \;\; Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t) ,$$
$$\forall t, s, b, m, c \;\; Percept([s, b, Glitter, m, c], t) \Rightarrow Glitter(t) ,$$

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 24. 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. Notice the correspondence between this rule and the direct percept–action connection in the circuit-based agent in Figure 7.20; the circuit connection *implicitly* quantifies over time.

SYNCHRONIC

So far in this section, the sentences dealing with time have been **synchronic** ("same time") sentences, that is, they relate properties of a world state to other properties of the same world state. Sentences that allow reasoning "across time" are called **diachronic**; for example, the agent needs to know how to combine information about its previous location with information about the action just taken in order to determine its current location. We will defer discussion of diachronic sentences until Chapter 10; for now, just assume that the required inferences have been made for location and other time-dependent predicates.

DIACHRONIC

We have represented the percepts and actions; 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

$$\forall x, y, a, b \;\; Adjacent([x, y], [a, b]) \;\Leftrightarrow\; [a, b] \in \{[x + 1, y], [x - 1, y], [x, y + 1], [x, y - 1]\} \; .$$

We could also name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among the pits.[9] It is much 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 wumpus lives in exactly one square, so it is a good idea to use a function such as $Home(Wumpus)$ to name that square. This completely avoids the cumbersome set of sentences required in propositional logic to say that exactly one square contains a wumpus. (It would be even worse for propositional logic with two wumpuses.)

The agent's location changes over time, so we will write $At(Agent, s, t)$ to mean that the agent is at square s at time t. 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). There are two kinds of synchronic rules that could allow such deductions:

DIAGNOSTIC RULES

♢ **Diagnostic rules**:
Diagnostic rules lead from observed effects to hidden causes. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit, or

$$\forall s \;\; Breezy(s) \;\Rightarrow\; \exists r \;\; Adjacent(r, s) \wedge Pit(r) \; ,$$

and that if a square is not breezy, no adjacent square contains a pit: [10]

$$\forall s \;\; \neg Breezy(s) \;\Rightarrow\; \neg\exists r \;\; Adjacent(r, s) \wedge Pit(r) \; .$$

Combining these two, we obtain the biconditional sentence

$$\forall s \;\; Breezy(s) \;\Leftrightarrow\; \exists r \;\; Adjacent(r, s) \wedge Pit(r) \; . \tag{8.3}$$

[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.

[10] There is a natural human tendency to forget to write down negative information such as this. In conversation, this tendency is entirely normal—it would be strange to say "There are two cups on the table *and there are not three or more*," even though "There are two cups on the table" is, strictly speaking, still true when there are three. We will return to this topic in Chapter 10.

CAUSAL RULES

♢ **Causal rules**:
Causal rules reflect the assumed direction of causality in the world: some hidden property of the world causes certain percepts to be generated. For example, a pit causes all adjacent squares to be breezy:

$$\forall r \;\; Pit(r) \;\Rightarrow\; [\forall s \;\; Adjacent(r, s) \;\Rightarrow\; Breezy(s)]$$

and if all squares adjacent to a given square are pitless, the square will not be breezy:

$$\forall s \;\; [\forall r \;\; Adjacent(r, s) \;\Rightarrow\; \neg Pit(r)] \;\Rightarrow\; \neg Breezy(s) \; .$$

With some work, it is possible to show that these two sentences together are logically equivalent to the biconditional sentence in Equation (8.3). The biconditional itself can also be thought of as causal, because it states how the truth value of $Breezy$ is generated from the world state.

MODEL-BASED REASONING

Systems that reason with causal rules are called **model-based reasoning** systems, because the causal rules form a model of how the environment operates. The distinction between model-based and diagnostic reasoning is important in many areas of AI. Medical diagnosis in particular has been an active area of research, in which approaches based on direct associations between symptoms and diseases (a diagnostic approach) have gradually been replaced by approaches using an explicit model of the disease process and how it manifests itself in symptoms. The issues come up again in Chapter 13.

Whichever kind of representation the agent uses, *if the axioms correctly and completely describe the way the world works and the way that percepts are produced, then any complete logical inference procedure will infer the strongest possible description of the world state, given the available percepts.* Thus, the agent designer can concentrate on getting the knowledge right, without worrying too much about the processes of deduction. Furthermore, we have seen that first-order logic can represent the wumpus world no less concisely than the original English-language description given in Chapter 7.

8.4 KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

KNOWLEDGE ENGINEERING

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 will illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar, so that we can concentrate on the representational issues involved. The approach we will take is suitable for developing *special-purpose* knowledge bases whose domain is carefully circumscribed and whose range of queries is known in advance. *General-purpose* knowledge bases, which are intended to support queries across the full range of human knowledge, are discussed in Chapter 10.

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 task.* 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 to answer questions only 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 specific problem instance.* If the ontology is well thought out, this step will be easy. It will involve 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 supplied with additional sentences in the same way that traditional programs are supplied with 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.
7. *Debug 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 identified easily by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes one of the diagnostic axioms for pits,

   $$\forall s \;\; Breezy(s) \;\Rightarrow\; \exists r \;\; Adjacent(r, s) \wedge Pit(r) \;,$$

   but not the other, then the agent will never be able to prove the *absence* of pits. 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, more important, 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 this statement is correct without looking at the rest of the program to see whether, for example, `offset` is used to refer to the current position, or to one beyond the current position, or whether the value of `position` is changed by another statement and so `offset` should also be changed again.

To understand this seven-step process better, we now apply it to an extended example—the domain of electronic circuits.

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 8.4. We follow the seven-step process for knowledge engineering.

Identify the task

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 8.4 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.

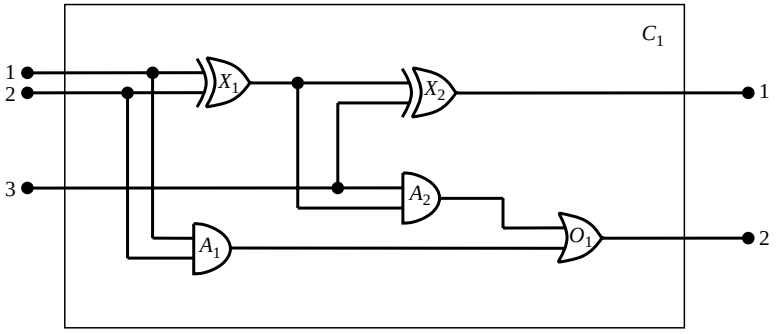


Figure 8.4 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates and one OR gate.

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 the wires take, or the junctions where two wires 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 mention the wire that actually connects them. There are many other factors of the domain that are irrelevant to our analysis, such as the size, shape, color, or cost of the various components.

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. We will start from individual gates and move up to circuits.

First, we need to be able to distinguish a gate from other gates. This is handled by naming gates with constants: X_1, X_2, and so on. Although each gate is connected into the circuit in its own individual way, its *behavior*—the way it transforms input signals into output

signals—depends only on its *type*. We can use a function to refer to the type of the gate.[11] For example, we can write $Type(X_1) = XOR$. This introduces the constant XOR for a particular type of gate; the other constants will be called OR, AND, and NOT. The $Type$ function is not the only way to encode the ontological distinction. We could have used a binary predicate, $Type(X_1, XOR)$, or several individual type predicates, such as $XOR(X_1)$. Either of these choices would work fine, but by choosing the function $Type$, we avoid the need for an axiom which says that each individual gate can have only one type. The semantics of functions already guarantees this.

Next we consider terminals. A gate or circuit can have one or more input terminals and one or more output terminals. We could simply name each one with a constant, just as we named gates. Thus, gate X_1 could have terminals named X_1In_1, X_1In_2, and X_1Out_1. The tendency to generate long compound names should be avoided, however. Calling something X_1In_1 does not make it the first input of X_1; we would still need to say this using an explicit assertion. It is probably better to name the gate using a function, just as we named King John's left leg $LeftLeg(John)$. Thus, let $In(1, X_1)$ denote the first input terminal for gate X_1. A similar function Out is used for output terminals.

The connectivity between gates can be represented by the 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, 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 C1?" We will therefore introduce as objects two "signal values" 1 and 0, and a function $Signal$ that takes a terminal as argument and denotes the signal value for that terminal.

Encode general knowledge of the domain

One sign that we have a good ontology is that there are very few general rules which need to be specified. A sign that we have a good vocabulary is that each rule can be stated clearly and concisely. With our example, we need only seven simple rules to describe everything we need to know about circuits:

1. If two terminals are connected, then they have the same signal:
   $\forall t_1, t_2 \;\; Connected(t_1, t_2) \;\Rightarrow\; Signal(t_1) = Signal(t_2)$
2. The signal at every terminal is either 1 or 0 (but not both):
   $\forall t \;\; Signal(t) = 1 \lor Signal(t) = 0$
   $1 \neq 0$
3. Connected is a commutative predicate:
   $\forall t_1, t_2 \;\; Connected(t_1, t_2) \;\Leftrightarrow\; Connected(t_2, t_1)$
4. An OR gate's output is 1 if and only if any of its inputs is 1:
   $\forall g \;\; Type(g) = OR \;\Rightarrow$
   $Signal(Out(1, g)) = 1 \;\Leftrightarrow\; \exists n \;\; Signal(In(n, g)) = 1$

[11] Note that we have used names beginning with appropriate letters—A_1, X_1, and so on—purely to make the example easier to read. The knowledge base must still contain type information for the gates.

5. An AND gate's output is 0 if and only if any of its inputs is 0:
$$\forall g \;\; Type(g) = AND \;\Rightarrow\; Signal(Out(1,g)) = 0 \;\Leftrightarrow\; \exists n \;\; Signal(In(n,g)) = 0$$
6. An XOR gate's output is 1 if and only if its inputs are different:
$$\forall g \;\; Type(g) = XOR \;\Rightarrow\; Signal(Out(1,g)) = 1 \;\Leftrightarrow\; Signal(In(1,g)) \neq Signal(In(2,g))$$
7. A NOT gate's output is different from its input:
$$\forall g \;\; (Type(g) = NOT) \;\Rightarrow\; Signal(Out(1,g)) \neq Signal(In(1,g))$$

Encode the specific problem instance

The circuit shown in Figure 8.4 is encoded as circuit C_1 with the following description. First, we categorize the gates:

$$Type(X_1) = XOR \quad Type(X_2) = XOR$$
$$Type(A_1) = AND \quad Type(A_2) = AND$$
$$Type(O_1) = OR$$

Then, we show the connections between them:

$$Connected(Out(1,X_1), In(1,X_2)) \quad Connected(In(1,C_1), In(1,X_1))$$
$$Connected(Out(1,X_1), In(2,A_2)) \quad Connected(In(1,C_1), In(1,A_1))$$
$$Connected(Out(1,A_2), In(1,O_1)) \quad Connected(In(2,C_1), In(2,X_1))$$
$$Connected(Out(1,A_1), In(2,O_1)) \quad Connected(In(2,C_1), In(2,A_1))$$
$$Connected(Out(1,X_2), Out(1,C_1)) \quad Connected(In(3,C_1), In(2,X_2))$$
$$Connected(Out(1,O_1), Out(2,C_1)) \quad Connected(In(3,C_1), In(1,A_2)) \;.$$

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?

$$\exists i_1, i_2, i_3 \;\; Signal(In(1,C_1)) = i_1 \wedge Signal(In(2,C_1)) = i_2 \wedge Signal(In(3,C_1)) = i_3 \wedge Signal(Out(1,C_1)) = 0 \wedge Signal(Out(2,C_1)) = 1 \;.$$

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. There are 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?

$$\exists i_1, i_2, i_3, o_1, o_2 \;\; Signal(In(1,C_1)) = i_1 \wedge Signal(In(2,C_1)) = i_2 \wedge Signal(In(3,C_1)) = i_3 \wedge Signal(Out(1,C_1)) = o_1 \wedge Signal(Out(2,C_1)) = o_2 \;.$$

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 8.17.) 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 omit the assertion that $1 \neq 0$.[12] 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

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

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 $1 \neq 0$.

8.5 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.
- A **possible world**, or **model**, for first-order logic is defined by a set of objects, the relations among them, and the functions that can be applied to them.
- **Constant symbols** name objects, **predicate symbols** name relations, and **function symbols** name functions. An **interpretation** specifies a mapping from symbols to the model. **Complex terms** apply function symbols to terms to name an object. Given an interpretation and a model, the truth of a sentence is determined.
- An **atomic sentence** consists of a predicate applied to one or more terms; it is true just when the relation named by the predicate holds between the objects named by the terms. **Complex sentences** use connectives just like propositional logic, and **quantified sentences** allow the expression of general rules.
- 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.

[12] This kind of omission is quite common because humans typically assume that different names refer to different things. Logic programming systems, described in Chapter 9, also make this assumption.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although even Aristotle's logic deals with generalizations over objects, true first-order logic dates from the introduction of quantifiers in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation"). Frege's ability to nest quantifiers was a big step forward, but he used an awkward notation. (An example appears on the front cover of this book.) 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).

A major barrier to the development of first-order logic had been the concentration on one-place predicates to the exclusion of many-place relational predicates. This fixation on one-place predicates had been nearly universal in logical systems from Aristotle up to and including Boole. 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 (1870), who also developed first-order logic independently of Frege, although slightly later (Peirce, 1883).

Leopold Löwenheim (1915) gave a systematic treatment of model theory for first-order logic in 1915. This paper also treated the equality symbol as an integral part of logic. 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.

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 described in Chapter 9. The logicist approach took root at Stanford. 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.

There are a number of good modern introductory texts on first-order logic. Quine (1982) 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. 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) provides both a solid introduction to logic and the first systematic treatment of logical agents with percepts and actions.

EXERCISES

8.1 A logical knowledge base represents the world using a set of sentences with no explicit structure. An **analogical** representation, on the other hand, has physical structure that corresponds directly to the structure of the thing represented. Consider a road map of your country as an analogical representation of facts about the country. The two-dimensional structure of the map corresponds to the two-dimensional surface of the area.

a. Give five examples of *symbols* in the map language.

b. An *explicit* sentence is a sentence that the creator of the representation actually writes down. An *implicit* sentence is a sentence that results from explicit sentences because of properties of the analogical representation. Give three examples each of *implicit* and *explicit* sentences in the map language.

c. Give three examples of facts about the physical structure of your country that cannot be represented in the map language.

d. Give two examples of facts that are much easier to express in the map language than in first-order logic.

e. Give two other examples of useful analogical representations. What are the advantages and disadvantages of each of these languages?

8.2 Consider a knowledge base containing just two sentences: $P(a)$ and $P(b)$. Does this knowledge base entail $\forall\, x\ P(x)$? Explain your answer in terms of models.

8.3 Is the sentence $\exists\, x, y\ \ x = y$ valid? Explain.

8.4 Write down a logical sentence such that every world in which it is true contains exactly one object.

8.5 Consider a symbol vocabulary that contains c constant symbols, p_k predicate symbols of each arity k, and f_k function symbols of each arity k, where $1 \leq k \leq A$. Let the domain size be fixed at D. For any given interpretation–model combination, each predicate or function symbol is mapped onto a relation or function, respectively, of the same arity. You may assume that the functions in the model allow some input tuples to have no value for the function (i.e., the value is the invisible object). Derive a formula for the number of possible interpretation–model combinations for a domain with D elements. Don't worry about eliminating redundant combinations.

8.6 Represent the following sentences in first-order logic, using a consistent vocabulary (which you must define):

a. Some students took French in spring 2001.

b. Every student who takes French passes it.

c. Only one student took Greek in spring 2001.

d. The best score in Greek is always higher than the best score in French.

e. Every person who buys a policy is smart.

f. No person buys an expensive policy.

g. There is an agent who sells policies only to people who are not insured.

h. There is a barber who shaves all men in town who do not shave themselves.

i. A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth.

j. A person born outside the UK, one of whose parents is a UK citizen by birth, is a UK citizen by descent.

k. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.

8.7 Represent the sentence "All Germans speak the same languages" in predicate calculus. Use $Speaks(x, l)$, meaning that person x speaks language l.

8.8 What axiom is needed to infer the fact $Female(Laura)$ given the facts $Male(Jim)$ and $Spouse(Jim, Laura)$?

8.9 Write a general set of facts and axioms to represent the assertion "Wellington heard about Napoleon's death" and to correctly answer the question "Did Napoleon hear about Wellington's death?"

8.10 Rewrite the propositional wumpus world facts from Section 7.5 into first-order logic. How much more compact is this version?

8.11 Write axioms describing the predicates $GrandChild$, $GreatGrandparent$, $Brother$, $Sister$, $Daughter$, Son, $Aunt$, $Uncle$, $BrotherInLaw$, $SisterInLaw$, and $FirstCousin$. Find out the proper definition of mth cousin n times removed, and write the definition in first-order logic.

Now write down the basic facts depicted in the family tree in Figure 8.5. Using a suitable logical reasoning system, TELL it all the sentences you have written down, and ASK it who are Elizabeth's grandchildren, Diana's brothers-in-law, and Zara's great-grandparents.

8.12 Write down a sentence asserting that + is a commutative function. Does your sentence follow from the Peano axioms? If so, explain why; if not, give a model in which the axioms are true and your sentence is false.

8.13 Explain what is wrong with the following proposed definition of the set membership predicate $\in$:

$$\forall x, s \;\; x \in \{x|s\}$$
$$\forall x, s \;\; x \in s \;\Rightarrow\; \forall y \;\; x \in \{y|s\} \; .$$

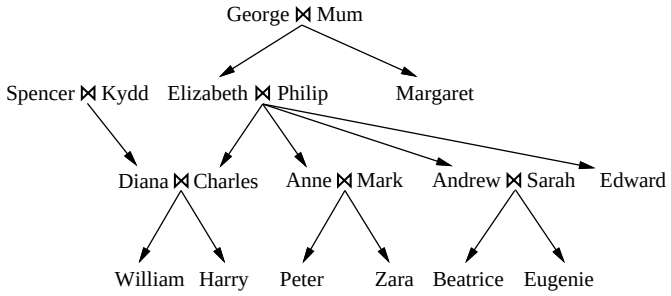


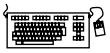
Figure 8.5 A typical family tree. The symbol "=" connects spouses and arrows point to children.

8.14 Using the set axioms as examples, write axioms for the list domain, including all the constants, functions, and predicates mentioned in the chapter.

8.15 Explain what is wrong with the following proposed definition of adjacent squares in the wumpus world:

$$\forall x, y \;\; Adjacent([x, y], [x + 1, y]) \wedge Adjacent([x, y], [x, y + 1]) \; .$$

8.16 Write out the axioms required for reasoning about the wumpus's location, using a constant symbol $Wumpus$ and a binary predicate $In(Wumpus, Location)$. Remember that there is only one wumpus.



8.17 Extend the vocabulary from Section 8.4 to define addition for n-bit binary numbers. Then encode the description of the four-bit adder in Figure 8.6, and pose the queries needed to verify that it is in fact correct.

8.18 The circuit representation in the chapter is more detailed than necessary if we care only about circuit functionality. A simpler formulation describes any m-input, n-output gate or circuit using a predicate with $m+n$ arguments, such that the predicate is true exactly when

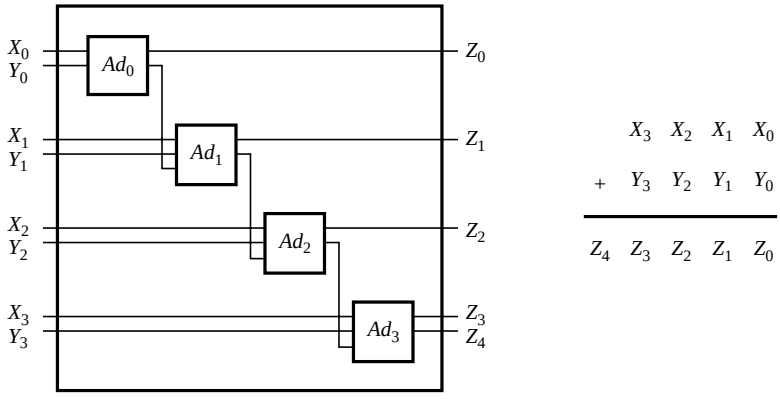


Figure 8.6 A four-bit adder.

the inputs and outputs are consistent. For example, NOT-gates are described by the binary predicate $NOT(i, o)$, for which $NOT(0, 1)$ and $NOT(1, 0)$ are known. Compositions of gates are defined by conjunctions of gate predicates in which shared variables indicate direct connections. For example, a NAND circuit can be composed from ANDs and NOTs:

$$\forall\, i_1, i_2, o_a, o \;\; NAND(i_1, i_2, o) \;\Leftrightarrow\; AND(i_1, i_2, o_a) \wedge NOT(o_a, o)\ .$$

Using this representation, define the one-bit adder in Figure 8.4 and the four-bit adder in Figure 8.6, and explain what queries you would use to verify the designs. What kinds of queries are *not* supported by this representation that *are* supported by the representation in Section 8.4?

8.19 Obtain a passport application for your country, identify the rules determining eligibility for a passport, and translate them into first-order logic, following the steps outlined in Section 8.4.