MODULE 4

FIRST-ORDER LOGIC

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.

8.1 REPRESENTATION REVISITED

In this section, we discuss the nature of representation languages. Our discussion motivates the development of first-order logic, a much more expressive language than the propositional logic introduced in Chapter 7. We 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 store and

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

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

COMPOSITIONALITY

As we saw in Chapter 7, however, propositional logic lacks the expressive power to *concisely* describe an environment with many objects. For example, we were forced to write a separate rule about breezes and pits for each square, such as

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

In English, on the other hand, it seems easy enough to say, once and for all, "Squares adjacent to pits are breezy." The syntax and semantics of English somehow make it possible to describe the environment concisely.

8.1.1 The language of thought

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

The modern view of natural language is that it serves a as a medium for **communication** rather than pure representation. When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence "Look!" represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the **context** in which the sentence was spoken. Clearly, one could not store a sentence such as "Look!" in a knowledge base and expect to

recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented. Natural languages also suffer from **ambiguity**, a problem for a representation language. As Pinker (1995) puts it: "When people think about *spring*, surely they are not confused as to whether they are thinking about a season or something that goes *boing*—and if one word can correspond to two thoughts, thoughts can't be words."

AMBIGUITY

The famous **Sapir–Whorf hypothesis** claims that our understanding of the world *is* strongly influenced by the language we speak. Whorf (1956) wrote "We cut nature up, organize it into concepts, and ascribe significances as we do, largely because we are parties to an agreement to organize it this way—an agreement that holds throughout our speech community and is codified in the patterns of our language." It is certainly true that different speech communities divide up the world differently. The French have two words "chaise" and "fauteuil," for a concept that English speakers cover with one: "chair." But English speakers can easily recognize the category fauteuil and give it a name—roughly "open-arm chair"—so does language really make a difference? Whorf relied mainly on intuition and speculation, but in the intervening years we actually have real data from anthropological, psychological and neurological studies.

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

"In this section, we discuss the nature of representation languages . . ."

"This section covers the topic of knowledge representation languages . . ."

Wanner (1974) did a similar experiment and found that subjects made the right choice at chance level—about 50% of the time—but remembered the content of what they read with better than 90% accuracy. This suggests that people process the words to form some kind of *nonverbal* representation.

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

Language also seems to influence thought through seemingly arbitrary grammatical features such as the gender of nouns. For example, "bridge" is masculine in Spanish and feminine in German. Boroditsky (2003) asked subjects to choose English adjectives to describe a photograph of a particular bridge. Spanish speakers chose *big*, *dangerous*, *strong*, and *towering*, whereas German speakers chose *beautiful*, *elegant*, *fragile*, and *slender*. Words can serve as anchor points that affect how we perceive the world. Loftus and Palmer (1974) showed experimental subjects a movie of an auto accident. Subjects who were asked "How fast were the cars going when they contacted each other?" reported an average of 32 mph, while subjects who were asked the question with the word "smashed" instead of "contacted" reported 41mph for the same cars in the same movie.

In a first-order logic reasoning system that uses CNF, we can see that the linguistic form "$\neg(A \vee B)$" and "$\neg A \wedge \neg B$" are the same because we can look inside the system and see that the two sentences are stored as the same canonical CNF form. Can we do that with the human brain? Until recently the answer was "no," but now it is "maybe." Mitchell *et al.* (2008) put subjects in an fMRI (functional magnetic resonance imaging) machine, showed them words such as "celery," and imaged their brains. The researchers were then able to train a computer program to predict, from a brain image, what word the subject had been presented with. Given two choices (e.g., "celery" or "airplane"), the system predicts correctly 77% of the time. The system can even predict at above-chance levels for words it has never seen an fMRI image of before (by considering the images of related words) and for people it has never seen before (proving that fMRI reveals some level of common representation across people). This type of work is still in its infancy, but fMRI (and other imaging technology such as intracranial electrophysiology (Sahin *et al.*, 2009)) promises to give us much more concrete ideas of what human knowledge representations are like.

From the viewpoint of formal logic, representing the same knowledge in two different ways makes absolutely no difference; the same facts will be derivable from either representation. In practice, however, one representation might require fewer steps to derive a conclusion, meaning that a reasoner with limited resources could get to the conclusion using one representation but not the other. For *nondeductive* tasks such as learning from experience, outcomes are *necessarily* dependent on the form of the representations used. We show in Chapter 18 that when a learning program considers two possible theories of the world, both of which are consistent with all the data, the most common way of breaking the tie is to choose the most succinct theory—and that depends on the language used to represent theories. Thus, the influence of language on thought is unavoidable for any agent that does learning.

8.1.2 Combining the best of formal and natural languages

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

OBJECT
RELATION
FUNCTION

- Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...

PROPERTY

- 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 ...

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 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."

ONTOLOGICAL COMMITMENT

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of *reality*. Mathematically, this commitment is expressed through the nature of the formal **models** with respect to which the truth of sentences is defined. For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states: true or false, and each model assigns $true$ or $false$ to each proposition symbol (see Section 7.4.2).[2] First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. The formal models are correspondingly more complicated than those for propositional logic. Special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) "first class" status within the logic, rather than simply defining them within the knowledge base.

TEMPORAL LOGIC

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 COMMITMENT

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,

[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 in fuzzy logic.

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.

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.

8.2.1 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. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic are much more interesting. First, they have objects in them! The **domain** of a model is the set of objects or **domain elements** it contains. The domain is required to be *nonempty*—every possible world must contain at least one object. (See Exercise 8.7 for a discussion of empty worlds.) Mathematically speaking, it doesn't matter *what* these objects are—all that matters is *how many* there are in each particular model—but for pedagogical purposes we'll use a concrete example. Figure 8.2 shows a model with five

DOMAIN

DOMAIN ELEMENTS

[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.

objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

TUPLE

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

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

(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 just one tuple, $\langle$the crown, King John$\rangle$. 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:

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

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"

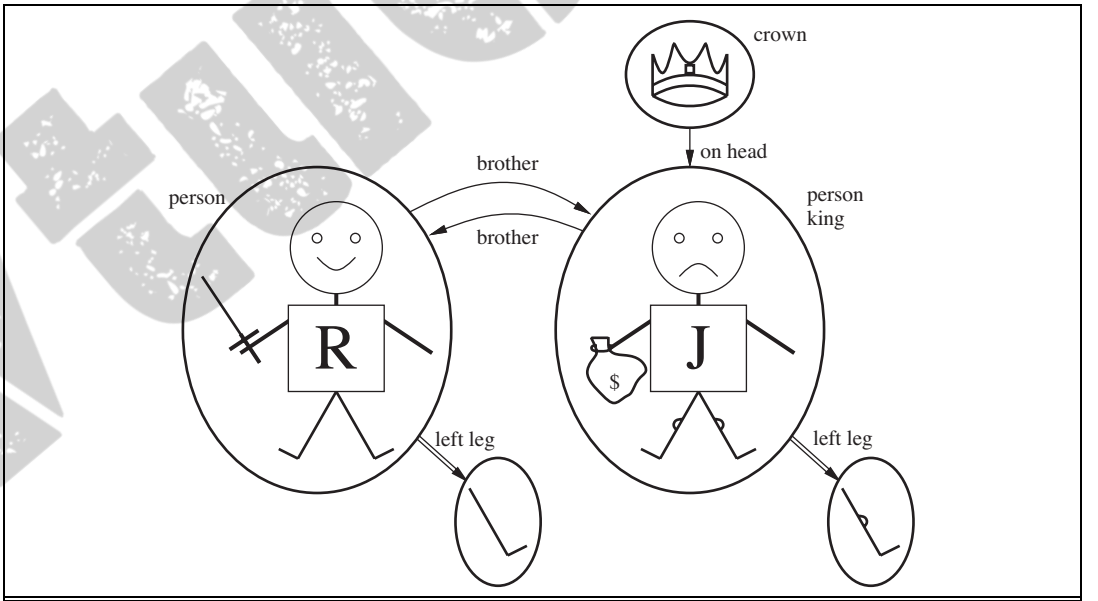


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

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.

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

8.2.2 Symbols and interpretations

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

CONSTANT SYMBOL
PREDICATE SYMBOL
FUNCTION SYMBOL

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.

ARITY

INTERPRETATION

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

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, of course. For example, one interpretation maps $Richard$ to the crown and $John$ to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols $Richard$ and $John$. 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.[4] 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.

[4] Later, in Section 8.2.8, we examine a semantics in which every object has exactly one name.

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

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

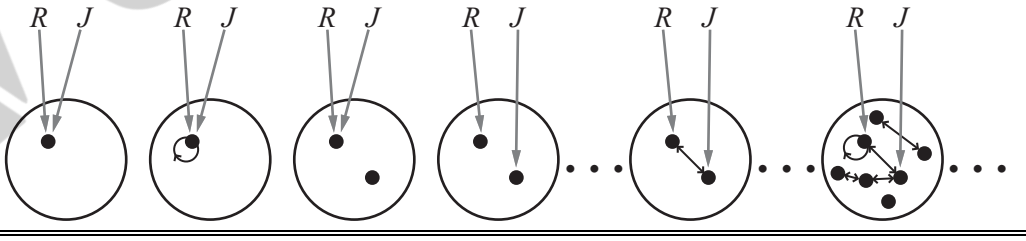


Figure 8.4 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects. Just as with propositional logic, entailment, validity, and so on are defined in terms of *all possible models*. To get an idea of what the set of all possible models looks like, see Figure 8.4. It shows that models vary in how many objects they contain—from one up to infinity—and in the way the constant symbols map to objects. If there are two constant symbols and one object, then both symbols must refer to the same object; but this can still happen even with more objects. When there are more objects than constant symbols, some of the objects will have no names. Because the number of possible models is unbounded, checking entailment by the enumeration of all possible models is not feasible for first-order logic (unlike propositional logic). Even if the number of objects is restricted, the number of combinations can be very large. (See Exercise 8.5.) For the example in Figure 8.4, there are 137,506,194,466 models with six or fewer objects.

8.2.3 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.[5]

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

8.2.4 Atomic sentences

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**

[5] **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.

ATOMIC SENTENCE
ATOM

sentence (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as

$$Brother(Richard, John).$$

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.[6] 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 if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

8.2.5 Complex sentences

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

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

8.2.6 Quantifiers

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*.

QUANTIFIER

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 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 if P is true in all possible **extended interpretations** constructed from the interpretation given in the model, where each

EXTENDED INTERPRETATION

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

extended interpretation specifies a domain element to which x refers.

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

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

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of "All kings are persons"? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for $\Rightarrow$ (Figure 7.8 on page 246), 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 definition of $\Rightarrow$ turns out to be perfect for writing general rules with universal quantifiers.

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

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

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 if P is true in *at least one* extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

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

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence "King John has a crown on his head." [7]

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 whenever *any* object fails to satisfy the premise; 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

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

[7] 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.

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:

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

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 they love someone. On the other hand, $\exists y \; (\forall x \; Loves(x, y))$ 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

$$\forall x \;\; (Crown(x) \vee (\exists x \;\; Brother(Richard, x))) \; .$$

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. 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 variable names with nested quantifiers.

Connections between $\forall$ and $\exists$

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

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

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:

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

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

8.2.7 Equality

EQUALITY SYMBOL

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

$$Father(John) = Henry$$

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

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

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

The sentence

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

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

8.2.8 An alternative semantics?

Continuing the example from the previous section, suppose that we believe that Richard has two brothers, John and Geoffrey.[8] Can we capture this state of affairs by asserting

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

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

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \wedge John \neq Geoffrey$$
$$\wedge \forall x \;\; Brother(x, Richard) \;\Rightarrow\; (x = John \vee x = Geoffrey)\ .$$

For many purposes, this seems much more cumbersome than the corresponding natural-language expression. As a consequence, humans may make mistakes in translating their knowledge into first-order logic, resulting in unintuitive behaviors from logical reasoning systems that use the knowledge. Can we devise a semantics that allows a more straightforward logical expression?

One proposal that is very popular in database systems works as follows. First, we insist that every constant symbol refer to a distinct object—the so-called **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false—the **closed-world assumption**. Finally, we invoke **domain closure**, meaning that each model

UNIQUE-NAMES ASSUMPTION
CLOSED-WORLD ASSUMPTION
DOMAIN CLOSURE

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

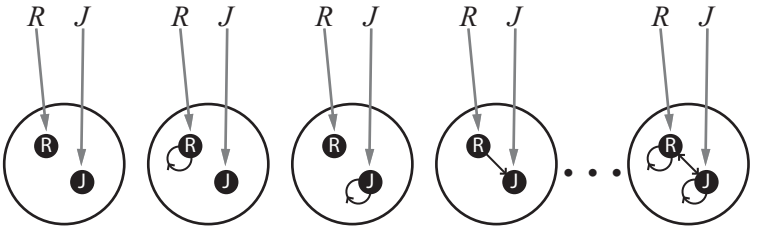


Figure 8.5 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

DATABASE SEMANTICS

contains no more domain elements than those named by the constant symbols. Under the resulting semantics, which we call **database semantics** to distinguish it from the standard semantics of first-order logic, the sentence Equation (8.3) does indeed state that Richard's two brothers are John and Geoffrey. Database semantics is also used in logic programming systems, as explained in Section 9.4.5.

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

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

8.3 Using First-Order Logic

Now that we have defined an expressive logical language, 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 provide more systematic representations

DOMAIN

of some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at

the wumpus world. The next section contains a more substantial example (electronic circuits) and Chapter 12 covers everything in the universe.

8.3.1 Assertions and queries in first-order logic

ASSERTION

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

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

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

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

QUERY
GOAL

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

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

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

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

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

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

SUBSTITUTION
BINDING LIST

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

8.3.2 The kinship domain

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

Clearly, the objects in our domain are people. We have two unary predicates, $Male$ and $Female$. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: $Parent$, $Sibling$, $Brother$, $Sister$, $Child$, $Daughter$, Son, $Spouse$, $Wife$, $Husband$, $Grandparent$, $Grandchild$, $Cousin$, $Aunt$, and $Uncle$. We use functions for $Mother$ and $Father$, because every person has exactly one of each of these (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 8.14 asks you to do just that.

Each of these sentences can be viewed as an **axiom** of the kinship domain, as explained in Section 7.1. Axioms are commonly associated with purely mathematical domains—we will see some axioms for numbers shortly—but they are needed in all domains. They provide the basic factual information from which useful conclusions can be derived. Our kinship axioms are also **definitions**; they have the form $\forall x, y \;\; P(x, y) \;\Leftrightarrow\; \ldots$. The axioms define the $Mother$ function and the $Husband$, $Male$, $Parent$, $Grandparent$, and $Sibling$ predicates in terms of other predicates. Our definitions "bottom out" at a basic set of predicates ($Child$, $Spouse$, and $Female$) in terms of which the others are ultimately defined. This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions. Notice that there is not necessarily a unique set of primitive predicates; we could equally well have used $Parent$, $Spouse$, and $Male$. In some domains, as we show, there is no clearly identifiable basic set.

DEFINITION

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

THEOREM

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

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

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

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

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

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

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

Axioms can also be "just plain facts," such as $Male(Jim)$ and $Spouse(Jim, Laura)$. Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the axioms. Often, one finds that the expected answers are not forthcoming—for example, from $Spouse(Jim, Laura)$ one expects (under the laws of many countries) to be able to infer $\neg Spouse(George, Laura)$; but this does not follow from the axioms given earlier—even after we add $Jim \neq George$ as suggested in Section 8.2.8. This is a sign that an axiom is missing. Exercise 8.8 asks the reader to supply it.

8.3.3 Numbers, sets, and lists

NATURAL NUMBERS

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of **natural numbers** or non-negative 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.[9] 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. (After reading Section 8.2.8, you will notice that these axioms allow for other natural numbers besides the usual ones; see Exercise 8.12.) We also need axioms to constrain the successor function:

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

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

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

The first of these axioms says that adding 0 to any natural number m gives m itself. Notice the use of the binary function symbol "+" in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using **infix** notation. (The notation we have used for first-order

INFIX

[9] 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.

PREFIX

logic is called **prefix**.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n+1$, so the second axiom becomes

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

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

SYNTACTIC SUGAR

The use of infix notation is an example of **syntactic sugar**, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be "desugared" to produce an equivalent sentence in ordinary first-order logic.

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.

SET

The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to define number theory in terms of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets by 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 we will want 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 $\{\}$ 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:

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cup s_2) \;\Leftrightarrow\; (x \in s_1 \vee x \in s_2) \; .$$

LIST

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 8.16 asks you to write out the axioms for lists.

8.3.4 The wumpus world

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

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

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

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

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

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

$$\text{AskVars}(\exists a \;\; BestAction(a, 5)) \; ,$$

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

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

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.

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

$$\forall x, y, a, b \;\; Adjacent([x, y], [a, b]) \;\Leftrightarrow \\ (x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)) \, .$$

We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits.[10] It is simpler to use a unary predicate Pit that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant $Wumpus$ is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint).

The agent's location changes over time, so we write $At(Agent, s, t)$ to mean that the agent is at square s at time t. We can fix the wumpus's location with $\forall t \; At(Wumpus, [2, 2], t)$. We can then say that objects can only be at one location at a time:

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

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

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

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

Having discovered which places are breezy (or smelly) and, very important, *not* breezy (or *not* smelly), the agent can deduce where the pits are (and where the wumpus is). Whereas propositional logic necessitates a separate axiom for each square (see R_2 and R_3 on page 247) and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

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

Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow (Equation (7.2) on page 267) becomes

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

From these two example sentences, we can see that the first-order logic formulation is no less concise than the original English-language description given in Chapter 7. The reader

[10] 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.

is invited to construct analogous axioms for the agent's location and orientation; in these cases, the axioms quantify over both space and time. As in the case of propositional state estimation, an agent can use logical inference with axioms of this kind to keep track of aspects of the world that are not directly observed. Chapter 10 goes into more depth on the subject of first-order successor-state axioms and their uses for constructing plans.

8.4 KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

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 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 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 cover a broad range of human knowledge and are intended to support tasks such as natural language understanding, are discussed in Chapter 12.

8.4.1 The knowledge-engineering process

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

1. *Identify the 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. Thus, we avoid the need for writing an application-specific solution algorithm.
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 easily identified by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes a diagnostic rule (see Exercise 8.13) for finding the wumpus,

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

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

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

is false for reptiles, amphibians, and, more importantly, 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.

8.4.2 The electronic circuits domain

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

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

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a

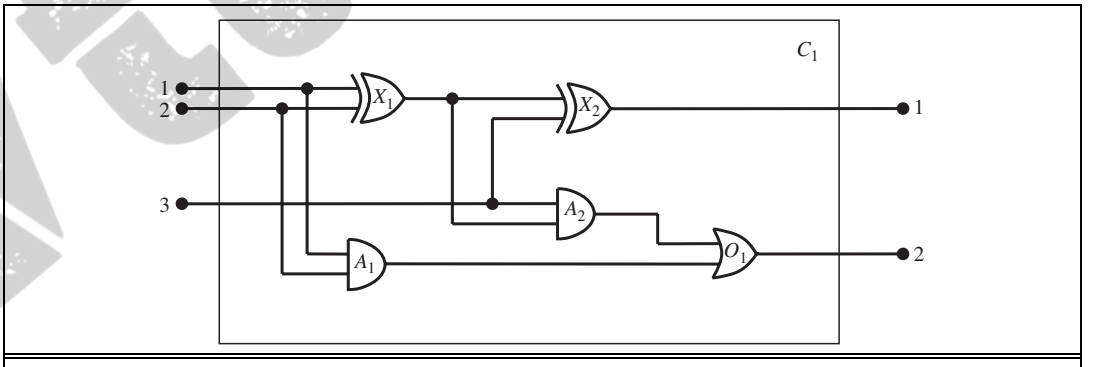


Figure 8.6 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.

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

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

Decide on a vocabulary

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

Next we consider terminals, which are identified by the predicate $Terminal(x)$. A gate or circuit can have one or more input terminals and one or more output terminals. We use the function $In(1, X_1)$ to denote the first input terminal for gate X_1. A similar function Out is used for output terminals. The function $Arity(c, i, j)$ says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a predicate, $Connected$, which takes two terminals as arguments, as in $Connected(Out(1, X_1), In(1, X_2))$.

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

Encode general knowledge of the domain

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

1. If two terminals are connected, then they have the same signal:
$$\forall\, t_1, t_2 \;\; Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \;\Rightarrow\; Signal(t_1) = Signal(t_2)\ .$$

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

Encode the specific problem instance

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

$$Circuit(C_1) \wedge Arity(C_1, 3, 2)$$
$$Gate(X_1) \wedge Type(X_1) = XOR$$
$$Gate(X_2) \wedge Type(X_2) = XOR$$
$$Gate(A_1) \wedge Type(A_1) = AND$$
$$Gate(A_2) \wedge Type(A_2) = AND$$
$$Gate(O_1) \wedge Type(O_1) = OR \; .$$

Then, we show the connections between them:

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

Pose queries to the inference procedure

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

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

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

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

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

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

This final query will return a complete input–output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification**. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out. (See Exercise 8.26.) Many domains are amenable to the same kind of structured knowledge-base development, in which more complex concepts are defined on top of simpler concepts.

CIRCUIT VERIFICATION

Debug the knowledge base

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

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

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 $1 \neq 0$.

9 INFERENCE IN FIRST-ORDER LOGIC

In which we define effective procedures for answering questions posed in first-order logic.

Chapter 7 showed how sound and complete inference can be achieved for propositional logic. In this chapter, we extend those results to obtain algorithms that can answer any answerable question stated in first-order logic. Section 9.1 introduces inference rules for quantifiers and shows how to reduce first-order inference to propositional inference, albeit at potentially great expense. Section 9.2 describes the idea of **unification**, showing how it can be used to construct inference rules that work directly with first-order sentences. We then discuss three major families of first-order inference algorithms. **Forward chaining** and its applications to **deductive databases** and **production systems** are covered in Section 9.3; **backward chaining** and **logic programming** systems are developed in Section 9.4. Forward and backward chaining can be very efficient, but are applicable only to knowledge bases that can be expressed as sets of Horn clauses. General first-order sentences require resolution-based **theorem proving**, which is described in Section 9.5.

9.1 PROPOSITIONAL VS. FIRST-ORDER INFERENCE

This section and the next introduce the ideas underlying modern logical inference systems. We begin with some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that *first-order* inference can be done by converting the knowledge base to *propositional* logic and using *propositional* inference, which we already know how to do. The next section points out an obvious shortcut, leading to inference methods that manipulate first-order sentences directly.

9.1.1 Inference rules for quantifiers

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x \;\; King(x) \wedge Greedy(x) \;\Rightarrow\; Evil(x) \; .$$

Then it seems quite permissible to infer any of the following sentences:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$
$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John)) .$$
$$\vdots$$

UNIVERSAL INSTANTIATION
GROUND TERM

The rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.[1] To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

EXISTENTIAL INSTANTIATION

In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)} .$$

For example, from the sentence

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

we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object. Of course, that name must not already belong to another object. Mathematics provides a nice example: suppose we discover that there is a number that is a little bigger than 2.71828 and that satisfies the equation $d(x^y)/dy = x^y$ for x. We can give this number a name, such as e, but it would be a mistake to give it the name of an existing object, such as π. In logic, the new name is called a **Skolem constant**. Existential Instantiation is a special case of a more general process called **skolemization**, which we cover in Section 9.5.

SKOLEM CONSTANT

Whereas Universal Instantiation can be applied many times to produce many different consequences, Existential Instantiation can be applied once, and then the existentially quantified sentence can be discarded. For example, we no longer need $\exists x \; Kill(x, Victim)$ once we have added the sentence $Kill(Murderer, Victim)$. Strictly speaking, the new knowledge base is not logically equivalent to the old, but it can be shown to be **inferentially equivalent** in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

INFERENTIAL EQUIVALENCE

[1] Do not confuse these substitutions with the extended interpretations used to define the semantics of quantifiers. The substitution replaces a variable with a term (a piece of syntax) to produce a new sentence, whereas an interpretation maps a variable to an object in the domain.

9.1.2 Reduction to propositional inference

Once we have rules for inferring nonquantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. In this section we give the main ideas; the details are given in Section 9.5.

The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations. For example, suppose our knowledge base contains just the sentences

$$\begin{array}{l} \forall x \;\; King(x) \wedge Greedy(x) \;\Rightarrow\; Evil(x) \\ King(John) \\ Greedy(John) \\ Brother(Richard, John) \; . \end{array} \tag{9.1}$$

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, $\{x/John\}$ and $\{x/Richard\}$. We obtain

$$\begin{array}{l} King(John) \wedge Greedy(John) \;\Rightarrow\; Evil(John) \\ King(Richard) \wedge Greedy(Richard) \;\Rightarrow\; Evil(Richard) \; , \end{array}$$

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences—$King(John)$, $Greedy(John)$, and so on—as proposition symbols. Therefore, we can apply any of the complete propositional algorithms in Chapter 7 to obtain conclusions such as $Evil(John)$.

This technique of **propositionalization** can be made completely general, as we show in Section 9.5; that is, every first-order knowledge base and query can be propositionalized in such a way that entailment is preserved. Thus, we have a complete decision procedure for entailment . . . or perhaps not. There is a problem: when the knowledge base includes a function symbol, the set of possible ground-term substitutions is infinite! For example, if the knowledge base mentions the $Father$ symbol, then infinitely many nested terms such as $Father(Father(Father(John)))$ can be constructed. Our propositional algorithms will have difficulty with an infinitely large set of sentences.

Fortunately, there is a famous theorem due to Jacques Herbrand (1930) to the effect that if a sentence is entailed by the original, first-order knowledge base, then there is a proof involving just a *finite* subset of the propositionalized knowledge base. Since any such subset has a maximum depth of nesting among its ground terms, we can find the subset by first generating all the instantiations with constant symbols ($Richard$ and $John$), then all terms of depth 1 ($Father(Richard)$ and $Father(John)$), then all terms of depth 2, and so on, until we are able to construct a propositional proof of the entailed sentence.

We have sketched an approach to first-order inference via propositionalization that is **complete**—that is, any entailed sentence can be proved. This is a major achievement, given that the space of possible models is infinite. On the other hand, we do not know until the proof is done that the sentence *is* entailed! What happens when the sentence is *not* entailed? Can we tell? Well, for first-order logic, it turns out that we cannot. Our proof procedure can go on and on, generating more and more deeply nested terms, but we will not know whether it is stuck in a hopeless loop or whether the proof is just about to pop out. This is very much

like the halting problem for Turing machines. Alan Turing (1936) and Alonzo Church (1936) both proved, in rather different ways, the inevitability of this state of affairs. *The question of entailment for first-order logic is* ***semidecidable****—that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every nonentailed sentence.*

9.2 Unification and Lifting

The preceding section described the understanding of first-order inference that existed up to the early 1960s. The sharp-eyed reader (and certainly the computational logicians of the early 1960s) will have noticed that the propositionalization approach is rather inefficient. For example, given the query $Evil(x)$ and the knowledge base in Equation (9.1), it seems perverse to generate sentences such as $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$. Indeed, the inference of $Evil(John)$ from the sentences

$$\forall x \;\; King(x) \wedge Greedy(x) \;\Rightarrow\; Evil(x)$$
$$King(John)$$
$$Greedy(John)$$

seems completely obvious to a human being. We now show how to make it completely obvious to a computer.

9.2.1 A first-order inference rule

The inference that John is evil—that is, that $\{x/John\}$ solves the query $Evil(x)$—works like this: to use the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitution θ that makes each of the conjuncts of the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution $\theta = \{x/John\}$ achieves that aim.

We can actually make the inference step do even more work. Suppose that instead of knowing $Greedy(John)$, we know that *everyone* is greedy:

$$\forall y \;\; Greedy(y) \; . \tag{9.2}$$

Then we would still like to be able to conclude that $Evil(John)$, because we know that John is a king (given) and John is greedy (because everyone is greedy). What we need for this to work is to find a substitution both for the variables in the implication sentence and for the variables in the sentences that are in the knowledge base. In this case, applying the substitution $\{x/John, y/John\}$ to the implication premises $King(x)$ and $Greedy(x)$ and the knowledge-base sentences $King(John)$ and $Greedy(y)$ will make them identical. Thus, we can infer the conclusion of the implication.

GENERALIZED MODUS PONENS

This inference process can be captured as a single inference rule that we call **Generalized Modus Ponens**:[2] For atomic sentences p_i, p_i', and q, where there is a substitution θ

such that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i,

$$\frac{p_1',\ p_2',\ \ldots,\ p_n',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)} .$$

There are $n+1$ premises to this rule: the n atomic sentences p_i' and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

p_1' is $King(John)$	p_1 is $King(x)$
p_2' is $Greedy(y)$	p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$	q is $Evil(x)$
$\text{SUBST}(\theta, q)$ is $Evil(John)$.	

It is easy to show that Generalized Modus Ponens is a sound inference rule. First, we observe that, for any sentence p (whose variables are assumed to be universally quantified) and for any substitution θ,

$$p \models \text{SUBST}(\theta, p)$$

holds by Universal Instantiation. It holds in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1', \ldots, p_n'$ we can infer

$$\text{SUBST}(\theta, p_1') \wedge \ldots \wedge \text{SUBST}(\theta, p_n')$$

and from the implication $p_1 \wedge \ldots \wedge p_n \Rightarrow q$ we can infer

$$\text{SUBST}(\theta, p_1) \wedge \ldots \wedge \text{SUBST}(\theta, p_n) \ \Rightarrow\ \text{SUBST}(\theta, q) .$$

Now, θ in Generalized Modus Ponens is defined so that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i; therefore the first of these two sentences matches the premise of the second exactly. Hence, $\text{SUBST}(\theta, q)$ follows by Modus Ponens.

LIFTING

Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic. We will see in the rest of this chapter that we can develop lifted versions of the forward chaining, backward chaining, and resolution algorithms introduced in Chapter 7. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions that are required to allow particular inferences to proceed.

9.2.2 Unification

UNIFICATION
UNIFIER

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) .$$

Let us look at some examples of how UNIFY should behave. Suppose we have a query $AskVars(Knows(John, x))$: whom does John know? Answers to this query can be found

[2] Generalized Modus Ponens is more general than Modus Ponens (page 249) in the sense that the known facts and the premise of the implication need match only up to a substitution, rather than exactly. On the other hand, Modus Ponens allows any sentence α as the premise, rather than just a conjunction of atomic sentences.

by finding all sentences in the knowledge base that unify with $Knows(John, x)$. Here are the results of unification with four different sentences that might be in the knowledge base:

$$\begin{aligned}
&\text{UNIFY}(Knows(John, x),\ Knows(John, Jane)) = \{x/Jane\}\\
&\text{UNIFY}(Knows(John, x),\ Knows(y, Bill)) = \{x/Bill, y/John\}\\
&\text{UNIFY}(Knows(John, x),\ Knows(y, Mother(y))) = \{y/John, x/Mother(John)\}\\
&\text{UNIFY}(Knows(John, x),\ Knows(x, Elizabeth)) = fail\ .
\end{aligned}$$

The last unification fails because x cannot take on the values $John$ and $Elizabeth$ at the same time. Now, remember that $Knows(x, Elizabeth)$ means "Everyone knows Elizabeth," so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $Knows(x, Elizabeth)$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

STANDARDIZING APART

$$\text{UNIFY}(Knows(John, x),\ Knows(x_{17}, Elizabeth)) = \{x/Elizabeth, x_{17}/John\}\ .$$

Exercise 9.12 delves further into the need for standardizing apart.

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $\text{UNIFY}(Knows(John, x), Knows(y, z))$ could return $\{y/John, x/z\}$ or $\{y/John, x/John, z/John\}$. The first unifier gives $Knows(John, z)$ as the result of unification, whereas the second gives $Knows(John, John)$. The second result could be obtained from the first by an additional substitution $\{z/John\}$; we say that the first unifier is *more general* than the second, because it places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single **most general unifier** (or MGU) that is unique up to renaming and substitution of variables. (For example, $\{x/John\}$ and $\{y/John\}$ are considered equivalent, as are $\{x/John, y/John\}$ and $\{x/John, y/x\}$.) In this case it is $\{y/John, x/z\}$.

MOST GENERAL UNIFIER

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, $S(x)$ can't unify with $S(S(x))$. This so-called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

OCCUR CHECK

9.2.3 Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. $\text{STORE}(s)$ stores a sentence s into the knowledge base and $\text{FETCH}(q)$ returns all unifiers such that the query q unifies with some

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

sentence in the knowledge base. The problem we used to illustrate unification—finding all facts that unify with $Knows(John, x)$—is an instance of FETCHing.

The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works, and it's all you need to understand the rest of the chapter. The remainder of this section outlines ways to make retrieval more efficient; it can be skipped on first reading.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. For example, there is no point in trying to unify $Knows(John, x)$ with $Brother(Richard, John)$. We can avoid such unifications by **indexing** the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the $Knows$ facts in one bucket and all the $Brother$ facts in another. The buckets can be stored in a hash table for efficient access.

INDEXING

PREDICATE INDEXING

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. Sometimes, however, a predicate has many clauses. For example, suppose that the tax authorities want to keep track of who employs whom, using a predicate $Employs(x, y)$. This would be a very large bucket with perhaps millions of employers

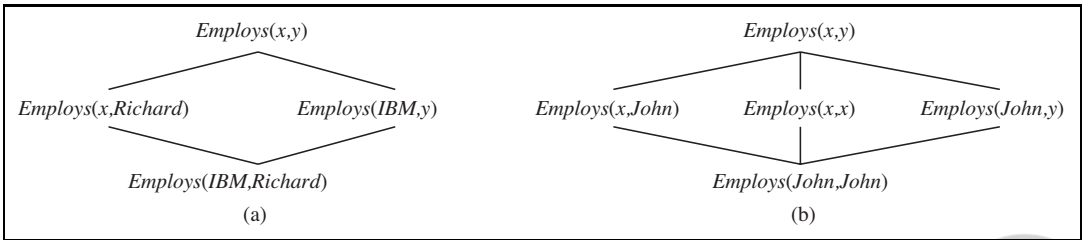


Figure 9.2 (a) The subsumption lattice whose lowest node is $Employs(IBM, Richard)$. (b) The subsumption lattice for the sentence $Employs(John, John)$.

and tens of millions of employees. Answering a query such as $Employs(x, Richard)$ with predicate indexing would require scanning the entire bucket.

For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query. For other queries, such as $Employs(IBM, y)$, we would need to have indexed the facts by combining the predicate with the first argument. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with.

Given a sentence to be stored, it is possible to construct indices for *all possible* queries that unify with it. For the fact $Employs(IBM, Richard)$, the queries are

$Employs(IBM, Richard)$	Does IBM employ Richard?
$Employs(x, Richard)$	Who employs Richard?
$Employs(IBM, y)$	Whom does IBM employ?
$Employs(x, y)$	Who employs whom?

SUBSUMPTION LATTICE

These queries form a **subsumption lattice**, as shown in Figure 9.2(a). The lattice has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the "highest" common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically (Exercise 9.5). A sentence with repeated constants has a slightly different lattice, as shown in Figure 9.2(b). Function symbols and variables in the sentences to be stored introduce still more interesting lattice structures.

The scheme we have described works very well whenever the lattice contains a small number of nodes. For a predicate with n arguments, however, the lattice contains $O(2^n)$ nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices. At some point, the benefits of indexing are outweighed by the costs of storing and maintaining all the indices. We can respond by adopting a fixed policy, such as maintaining indices only on keys composed of a predicate plus each argument, or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked. For most AI systems, the number of facts to be stored is small enough that efficient indexing is considered a solved problem. For commercial databases, where facts number in the billions, the problem has been the subject of intensive study and technology development..

9.3 FORWARD CHAINING

A forward-chaining algorithm for propositional definite clauses was given in Section 7.5. The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses. Definite clauses such as $Situation \Rightarrow Response$ are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and forward chaining can be implemented very efficiently.

9.3.1 First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 256): they are disjunctions of literals of which *exactly one is positive*. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Not every knowledge base can be converted into a set of definite clauses because of the single-positive-literal restriction, but many can. Consider the following problem:

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

"... it is a crime for an American to sell weapons to hostile nations":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) . \quad (9.3)$$

"Nono ... has some missiles." The sentence $\exists x\ Owns(Nono, x) \wedge Missile(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M_1:

$$Owns(Nono, M_1) \quad (9.4)$$
$$Missile(M_1) \quad (9.5)$$

"All of its missiles were sold to it by Colonel West":

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) . \quad (9.6)$$

We will also need to know that missiles are weapons:

$$Missile(x) \Rightarrow Weapon(x) \quad (9.7)$$

and we must know that an enemy of America counts as "hostile":

$$Enemy(x, America) \Rightarrow Hostile(x) . \tag{9.8}$$

"West, who is American ...":

$$American(West) . \tag{9.9}$$

"The country Nono, an enemy of America ...":

$$Enemy(Nono, America) . \tag{9.10}$$

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases. We will see that the absence of function symbols makes inference much easier.

DATALOG

9.3.2 A simple forward-chaining algorithm

The first forward-chaining algorithm we consider is a simple one, shown in Figure 9.3. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. Notice that a fact is not "new" if it is just a **renaming** of a known fact. One sentence is a renaming of another if they are identical except for the names of the variables. For example, $Likes(x, IceCream)$ and $Likes(y, IceCream)$ are renamings of each other because they differ only in the choice of x or y; their meanings are identical: everyone likes ice cream.

RENAMING

We use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (9.3), (9.6), (9.7), and (9.8). Two iterations are required:

- On the first iteration, rule (9.3) has unsatisfied premises.
  Rule (9.6) is satisfied with $\{x/M_1\}$, and $Sells(West, M_1, Nono)$ is added.
  Rule (9.7) is satisfied with $\{x/M_1\}$, and $Weapon(M_1)$ is added.
  Rule (9.8) is satisfied with $\{x/Nono\}$, and $Hostile(Nono)$ is added.
- On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and $Criminal(West)$ is added.

Figure 9.4 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar to those for propositional forward chaining (page 258); the principal difference is that a first-order fixed point can include universally quantified atomic sentences.

FOL-FC-ASK is easy to analyze. First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses. For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each rule in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new then
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

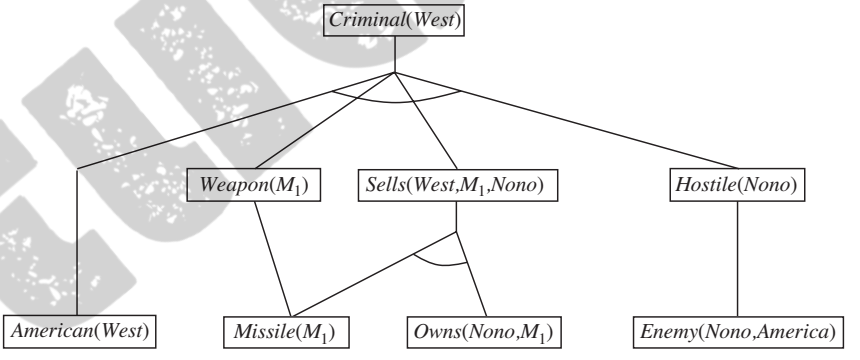


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

possible facts that can be added, which determines the maximum number of iterations. Let k be the maximum **arity** (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols. Clearly, there can be no more than pn^k distinct ground facts, so after this many iterations the algorithm must have reached a fixed point. Then we can make an argument very similar to the proof of completeness for propositional forward

chaining. (See page 258.) The details of how to make the transition from propositional to first-order completeness are given for the resolution algorithm in Section 9.5.

For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence q is entailed, we must appeal to Herbrand's theorem to establish that the algorithm will find a proof. (See Section 9.5 for the resolution case.) If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms

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

then forward chaining adds $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$, and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semidecidable.

9.3.3 Efficient forward chaining

The forward-chaining algorithm in Figure 9.3 is designed for ease of understanding rather than for efficiency of operation. There are three possible sources of inefficiency. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration. Finally, the algorithm might generate many facts that are irrelevant to the goal. We address each of these issues in turn.

PATTERN MATCHING

Matching rules against known facts

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

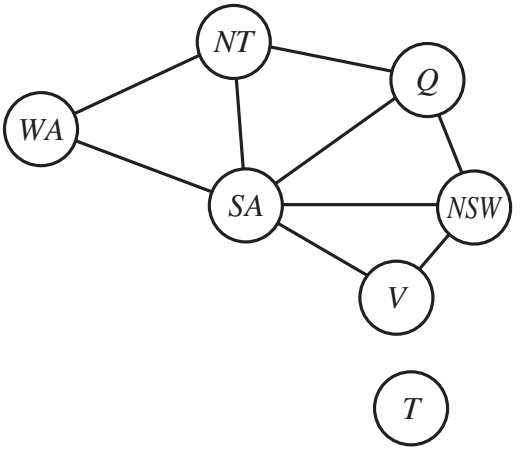
$$Missile(x) \Rightarrow Weapon(x) \;.$$

Then we need to find all the facts that unify with $Missile(x)$; in a suitably indexed knowledge base, this can be done in constant time per fact. Now consider a rule such as

$$Missile(x) \wedge Owns(Nono, x) \;\Rightarrow\; Sells(West, x, Nono) \;.$$

Again, we can find all the objects owned by Nono in constant time per object; then, for each object, we could check whether it is a missile. If the knowledge base contains many objects owned by Nono and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nono. This is the **conjunct ordering** problem: find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized. It turns out that finding the optimal ordering is NP-hard, but good heuristics are available. For example, the **minimum-remaining-values** (MRV) heuristic used for CSPs in Chapter 6 would suggest ordering the conjuncts to look for missiles first if fewer missiles than objects are owned by Nono.

CONJUNCT ORDERING



(a)

$$Diff(wa, nt) \wedge Diff(wa, sa) \wedge$$
$$Diff(nt, q) \wedge Diff(nt, sa) \wedge$$
$$Diff(q, nsw) \wedge Diff(q, sa) \wedge$$
$$Diff(nsw, v) \wedge Diff(nsw, sa) \wedge$$
$$Diff(v, sa) \Rightarrow Colorable()$$
$$Diff(Red, Blue) \quad Diff(Red, Green)$$
$$Diff(Green, Red) \quad Diff(Green, Blue)$$
$$Diff(Blue, Red) \quad Diff(Blue, Green)$$

(b)

Figure 9.5 (a) Constraint graph for coloring the map of Australia. (b) The map-coloring CSP expressed as a single definite clause. Each map region is represented as a variable whose value can be one of the constants Red, $Green$ or $Blue$.

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains—for example, $Missile(x)$ is a unary constraint on x. Extending this idea, *we can express every finite-domain CSP as a single definite clause together with some associated ground facts.* Consider the map-coloring problem from Figure 6.1, shown again in Figure 9.5(a). An equivalent formulation as a single definite clause is given in Figure 9.5(b). Clearly, the conclusion $Colorable()$ can be inferred only if the CSP has a solution. Because CSPs in general include 3-SAT problems as special cases, we can conclude that *matching a definite clause against a set of facts is NP-hard.*

It might seem rather depressing that forward chaining has an NP-hard matching problem in its inner loop. There are three ways to cheer ourselves up:

DATA COMPLEXITY

- We can remind ourselves that most rules in real-world knowledge bases are small and simple (like the rules in our crime example) rather than large and complex (like the CSP formulation in Figure 9.5). It is common in the database world to assume that both the sizes of rules and the arities of predicates are bounded by a constant and to worry only about **data complexity**—that is, the complexity of inference as a function of the number of ground facts in the knowledge base. It is easy to show that the data complexity of forward chaining is polynomial.
- We can consider subclasses of rules for which matching is efficient. Essentially every Datalog clause can be viewed as defining a CSP, so matching will be tractable just when the corresponding CSP is tractable. Chapter 6 describes several tractable families of CSPs. For example, if the constraint graph (the graph whose nodes are variables and whose links are constraints) forms a tree, then the CSP can be solved in linear time. Exactly the same result holds for rule matching. For instance, if we remove South

Australia from the map in Figure 9.5, the resulting clause is

$$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \Rightarrow Colorable()$$

which corresponds to the reduced CSP shown in Figure 6.12 on page 224. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.

- We can try to to eliminate redundant rule-matching attempts in the forward-chaining algorithm, as described next.

Incremental forward chaining

When we showed how forward chaining works on the crime example, we cheated; in particular, we omitted some of the rule matching done by the algorithm shown in Figure 9.3. For example, on the second iteration, the rule

$$Missile(x) \Rightarrow Weapon(x)$$

matches against $Missile(M_1)$ (again), and of course the conclusion $Weapon(M_1)$ is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation: *Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration $t-1$.* This is true because any inference that does not require a new fact from iteration $t-1$ could have been done at iteration $t-1$ already.

This observation leads naturally to an incremental forward-chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p_i that unifies with a fact p'_i newly inferred at iteration $t-1$. The rule-matching step then fixes p_i to match with p'_i, but allows the other conjuncts of the rule to match with facts from any previous iteration. This algorithm generates exactly the same facts at each iteration as the algorithm in Figure 9.3, but is much more efficient.

With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact, and indeed many real systems operate in an "update" mode wherein forward chaining occurs in response to each new fact that is TELLed to the system. Inferences cascade through the set of rules until the fixed point is reached, and then the process begins again for the next new fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of redundant work is done in repeatedly constructing partial matches that have some unsatisfied premises. Our crime example is rather too small to show this effectively, but notice that a partial match is constructed on the first iteration between the rule

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

and the fact $American(West)$. This partial match is then discarded and rebuilt on the second iteration (when the rule succeeds). It would be better to retain and gradually complete the partial matches as new facts arrive, rather than discarding them.

RETE

The **rete** algorithm[3] was the first to address this problem. The algorithm preprocesses the set of rules in the knowledge base to construct a sort of dataflow network in which each

[3] Rete is Latin for net. The English pronunciation rhymes with treaty.

node is a literal from a rule premise. Variable bindings flow through the network and are filtered out when they fail to match a literal. If two literals in a rule share a variable—for example, $Sells(x, y, z) \wedge Hostile(z)$ in the crime example—then the bindings from each literal are filtered through an equality node. A variable binding reaching a node for an n-ary literal such as $Sells(x, y, z)$ might have to wait for bindings for the other variables to be established before the process can continue. At any given point, the state of a rete network captures all the partial matches of the rules, avoiding a great deal of recomputation.

PRODUCTION SYSTEM

Rete networks, and various improvements thereon, have been a key component of so-called **production systems**, which were among the earliest forward-chaining systems in widespread use.[4] The XCON system (originally called R1; McDermott, 1982) was built with a production-system architecture. XCON contained several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built with the same underlying technology, which has been implemented in the general-purpose language OPS-5.

COGNITIVE ARCHITECTURES

Production systems are also popular in **cognitive architectures**—that is, models of human reasoning—such as ACT (Anderson, 1983) and SOAR (Laird *et al.*, 1987). In such systems, the "working memory" of the system models human short-term memory, and the productions are part of long-term memory. On each cycle of operation, productions are matched against the working memory of facts. A production whose conditions are satisfied can add or delete facts in working memory. In contrast to the typical situation in databases, production systems often have many rules and relatively few facts. With suitably optimized matching technology, some modern systems can operate in real time with tens of millions of rules.

Irrelevant facts

The final source of inefficiency in forward chaining appears to be intrinsic to the approach and also arises in the propositional context. Forward chaining makes all allowable inferences based on the known facts, *even if they are irrelevant to the goal at hand*. In our crime example, there were no rules capable of drawing irrelevant conclusions, so the lack of directedness was not a problem. In other cases (e.g., if many rules describe the eating habits of Americans and the prices of missiles), FOL-FC-ASK will generate many irrelevant conclusions.

One way to avoid drawing irrelevant conclusions is to use backward chaining, as described in Section 9.4. Another solution is to restrict forward chaining to a selected subset of rules, as in PL-FC-ENTAILS? (page 258). A third approach has emerged in the field of **deductive databases**, which are large-scale databases, like relational databases, but which use forward chaining as the standard inference tool rather than SQL queries. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings—those belonging to a so-called **magic set**—are considered during forward inference. For example, if the goal is $Criminal(West)$, the rule that concludes $Criminal(x)$ will be rewritten to include an extra conjunct that constrains the value of x:

DEDUCTIVE DATABASES

MAGIC SET

$$Magic(x) \wedge American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) .$$

[4] The word **production** in **production systems** denotes a condition–action rule.

The fact $Magic(West)$ is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process. The complete process for defining magic sets and rewriting the knowledge base is too complex to go into here, but the basic idea is to perform a sort of "generic" backward inference from the goal in order to work out which variable bindings need to be constrained. The magic sets approach can therefore be thought of as a kind of hybrid between forward inference and backward preprocessing.