7 LOGICAL AGENTS

In which we design agents that can form representations of a complex world, use a process of inference to derive new representations about the world, and use these new representations to deduce what to do.

REASONING
REPRESENTATION
KNOWLEDGE-BASED AGENTS

Humans, it seems, know things; and what they know helps them do things. These are not empty statements. They make strong claims about how the intelligence of humans is achieved—not by purely reflex mechanisms but by processes of **reasoning** that operate on internal **representations** of knowledge. In AI, this approach to intelligence is embodied in **knowledge-based agents**.

The problem-solving agents of Chapters 3 and 4 know things, but only in a very limited, inflexible sense. For example, the transition model for the 8-puzzle—knowledge of what the actions do—is hidden inside the domain-specific code of the RESULT function. It can be used to predict the outcome of actions but not to deduce that two tiles cannot occupy the same space or that states with odd parity cannot be reached from states with even parity. The atomic representations used by problem-solving agents are also very limiting. In a partially observable environment, an agent's only choice for representing what it knows about the current state is to list all possible concrete states—a hopeless prospect in large environments.

Chapter 6 introduced the idea of representing states as assignments of values to variables; this is a step in the right direction, enabling some parts of the agent to work in a domain-independent way and allowing for more efficient algorithms. In this chapter and those that follow, we take this step to its logical conclusion, so to speak—we develop **logic** as a general class of representations to support knowledge-based agents. Such agents can combine and recombine information to suit myriad purposes. Often, this process can be quite far removed from the needs of the moment—as when a mathematician proves a theorem or an astronomer calculates the earth's life expectancy. Knowledge-based agents can accept new tasks in the form of explicitly described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge.

LOGIC

We begin in Section 7.1 with the overall agent design. Section 7.2 introduces a simple new environment, the wumpus world, and illustrates the operation of a knowledge-based agent without going into any technical detail. Then we explain the general principles of **logic**

in Section 7.3 and the specifics of **propositional logic** in Section 7.4. While less expressive than **first-order logic** (Chapter 8), propositional logic illustrates all the basic concepts of logic; it also comes with well-developed inference technologies, which we describe in sections 7.5 and 7.6. Finally, Section 7.7 combines the concept of knowledge-based agents with the technology of propositional logic to build some simple agents for the wumpus world.

7.1 KNOWLEDGE-BASED AGENTS

KNOWLEDGE BASE
SENTENCE
KNOWLEDGE REPRESENTATION LANGUAGE
AXIOM

The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively.

INFERENCE

Both operations may involve **inference**—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously. Later in this chapter, we will be more precise about the crucial word "follow." For now, take it to mean that the inference process should not make things up as it goes along.

Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, KB,

BACKGROUND KNOWLEDGE

which may initially contain some **background knowledge**.

Each time the agent program is called, it does three things. First, it TELLs the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action.

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other. MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK. Later sections will reveal these details.

The agent in Figure 7.1 appears quite similar to the agents with internal state described in Chapter 2. Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

KNOWLEDGE LEVEL

the **knowledge level**, where we need specify only what the agent knows and what its goals are, in order to fix its behavior. For example, an automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge *because it knows that that will achieve its goal*. Notice that this analysis is independent of how the taxi works at the **implementation level**. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

IMPLEMENTATION LEVEL

A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the **declarative** approach to system building. In contrast, the **procedural** approach encodes desired behaviors directly as program code. In the 1970s and 1980s, advocates of the two approaches engaged in heated debates. We now understand that a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.

DECLARATIVE

We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself. These mechanisms, which are discussed in Chapter 18, create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

7.2 THE WUMPUS WORLD

In this section we describe an environment in which knowledge-based agents can show their worth. The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain

WUMPUS WORLD

bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

A sample wumpus world is shown in Figure 7.2. The precise definition of the task environment is given, as suggested in Section 2.3, by the PEAS description:

- **Performance measure**: +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment**: A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- **Actuators**: The agent can move *Forward*, *TurnLeft* by $90°$, or *TurnRight* by $90°$. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].
- **Sensors**: The agent has five sensors, each of which gives a single bit of information:
  - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
  - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
  - In the square where the gold is, the agent will perceive a *Glitter*.
  - When an agent walks into a wall, it will perceive a *Bump*.
  - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

  The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get $[Stench, Breeze, None, None, None]$.

We can characterize the wumpus environment along the various dimensions given in Chapter 2. Clearly, it is discrete, static, and single-agent. (The wumpus doesn't move, fortunately.) It is sequential, because rewards may come only after many actions are taken. It is partially observable, because some aspects of the state are not directly perceivable: the agent's location, the wumpus's state of health, and the availability of an arrow. As for the locations of the pits and the wumpus: we could treat them as unobserved parts of the state that happen to be immutable—in which case, the transition model for the environment is completely

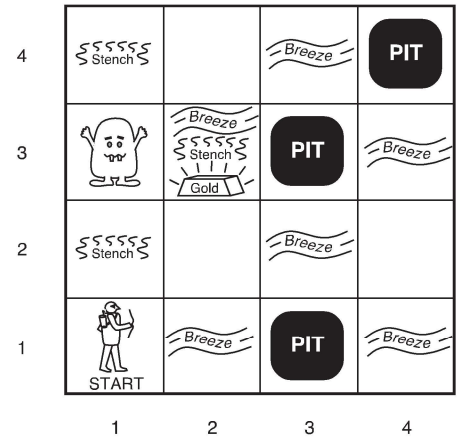


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

known; or we could say that the transition model itself is unknown because the agent doesn't know which *Forward* actions are fatal—in which case, discovering the locations of pits and wumpus completes the agent's knowledge of the transition model.

For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment; overcoming this ignorance seems to require logical reasoning. In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure 7.2. We use an informal knowledge representation language consisting of writing down symbols in a grid (as in Figures 7.3 and 7.4).

The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively, in square [1,1].

The first percept is $[None, None, None, None, None]$, from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure 7.3(a) shows the agent's state of knowledge at this point.

A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation "P?" in Figure 7.3(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the

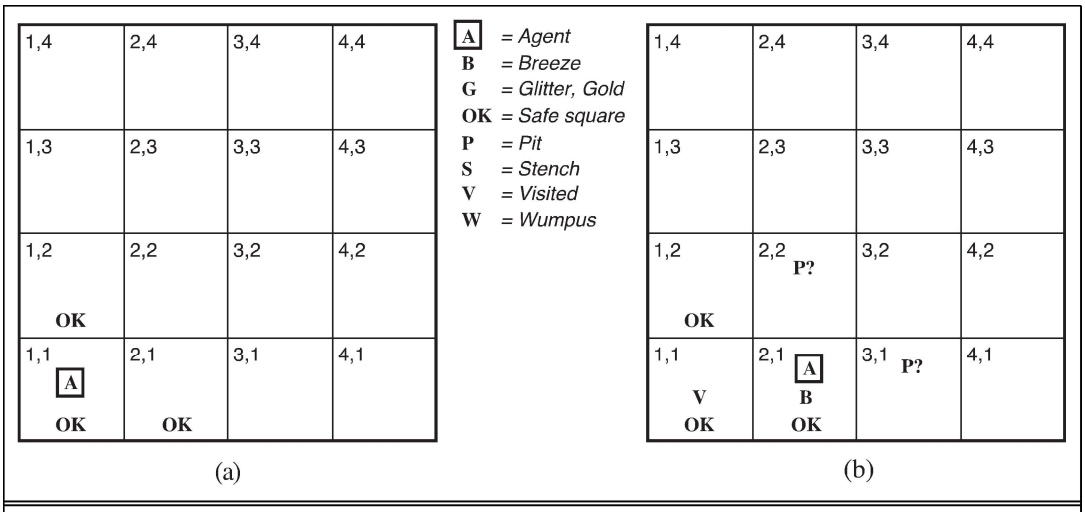


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept $[None, None, None, None, None]$. (b) After one move, with percept $[None, Breeze, None, None, None]$.

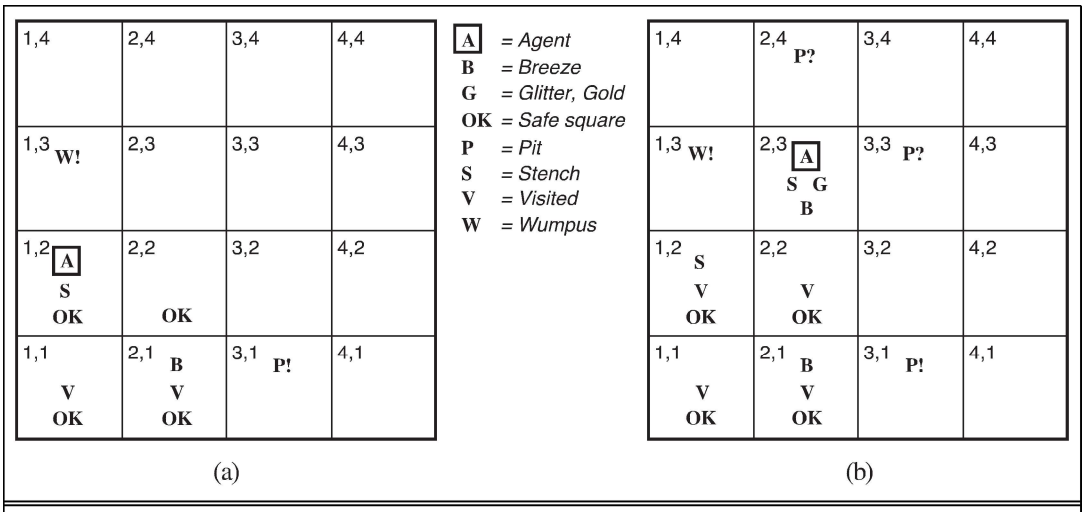


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept $[Stench, None, None, None, None]$. (b) After the fifth move, with percept $[Stench, Breeze, Glitter, None, None]$.

wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 7.4(b). In [2,3], the agent detects a glitter, so it should grab the gold and then return home.

Note that in each case for which the agent draws a conclusion from the available information, that conclusion is *guaranteed* to be correct if the available information is correct. This is a fundamental property of logical reasoning. In the rest of this chapter, we describe how to build logical agents that can represent information and draw conclusions such as those described in the preceding paragraphs.

7.3 LOGIC

This section summarizes the fundamental concepts of logical representation and reasoning. These beautiful ideas are independent of any of logic's particular forms. We therefore postpone the technical details of those forms until the next section, using instead the familiar example of ordinary arithmetic.

SYNTAX

In Section 7.1, we said that knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: "$x + y = 4$" is a well-formed sentence, whereas "$x4y+ =$" is not.

SEMANTICS

TRUTH

POSSIBLE WORLD

A logic must also define the **semantics** or meaning of sentences. The semantics defines the **truth** of each sentence with respect to each **possible world**. For example, the semantics for arithmetic specifies that the sentence "$x + y = 4$" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no "in between."[1]

MODEL

When we need to be precise, we use the term **model** in place of "possible world." Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which simply fixes the truth or falsehood of every relevant sentence. Informally, we may think of a possible world as, for example, having x men and y women sitting at a table playing bridge, and the sentence $x + y = 4$ is true when there are four people in total. Formally, the possible models are just all possible assignments of real numbers to the variables x and y. Each such assignment fixes the truth of any sentence of arithmetic whose variables are x and y. If a sentence α is true in model m, we say that m **satisfies** α or sometimes m **is a model of** α. We use the notation $M(\alpha)$ to mean the set of all models of α.

SATISFACTION

Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write

ENTAILMENT

$$\alpha \models \beta$$

[1] **Fuzzy logic**, discussed in Chapter 14, allows for degrees of truth.

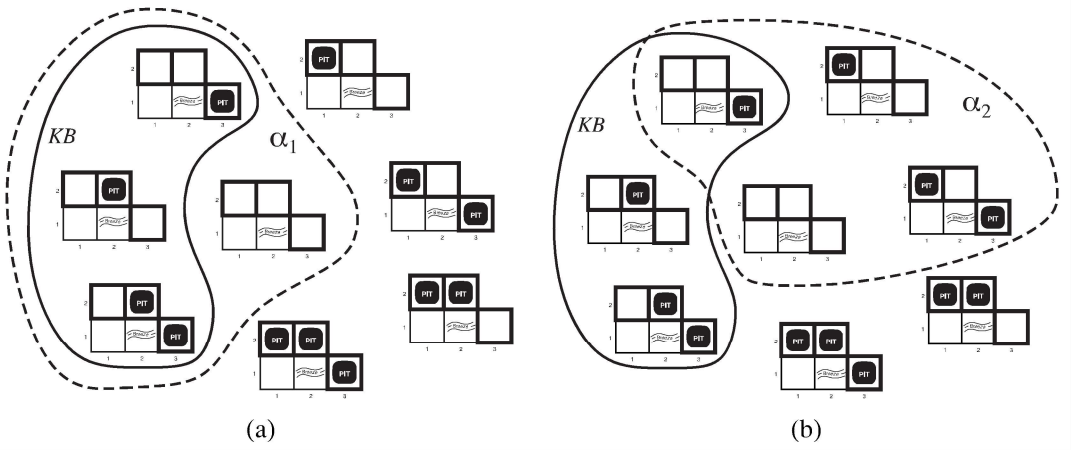


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

to mean that the sentence α entails the sentence β. The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta) \ .$$

(Note the direction of the $\subseteq$ here: if $\alpha \models \beta$, then α is a *stronger* assertion than β: it rules out *more* possible worlds.) The relation of entailment is familiar from arithmetic; we are happy with the idea that the sentence $x = 0$ entails the sentence $xy = 0$. Obviously, in any model where x is zero, it is the case that xy is zero (regardless of the value of y).

We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section. Consider the situation in Figure 7.3(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are $2^3 = 8$ possible models. These eight models are shown in Figure 7.5.[2]

The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows—for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are

[2] Although the figure shows the models as partial wumpus worlds, they are really nothing more than assignments of *true* and *false* to the sentences "there is a pit in [1,2]" etc. Models, in the mathematical sense, do not need to have 'orrible 'airy wumpuses in them.

shown surrounded by a solid line in Figure 7.5. Now let us consider two possible conclusions:

α_1 = "There is no pit in [1,2]."
α_2 = "There is no pit in [2,2]."

We have surrounded the models of α_1 and α_2 with dotted lines in Figures 7.5(a) and 7.5(b), respectively. By inspection, we see the following:

in every model in which KB is true, α_1 is also true.

Hence, $KB \models \alpha_1$: there is no pit in [1,2]. We can also see that

in some models in which KB is true, α_2 is false.

Hence, $KB \not\models \alpha_2$: the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)[3]

The preceding example not only illustrates entailment but also shows how the definition of entailment can be applied to derive conclusions—that is, to carry out **logical inference**. The inference algorithm illustrated in Figure 7.5 is called **model checking**, because it enumerates all possible models to check that α is true in all models in which KB is true, that is, that $M(KB) \subseteq M(\alpha)$.

LOGICAL INFERENCE
MODEL CHECKING

In understanding entailment and inference, it might help to think of the set of all consequences of KB as a haystack and of α as a needle. Entailment is like the needle being in the haystack; inference is like finding it. This distinction is embodied in some formal notation: if an inference algorithm i can derive α from KB, we write

$$KB \vdash_i \alpha \ ,$$

which is pronounced "α is derived from KB by i" or "i derives α from KB."

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles. It is easy to see that model checking, when it is applicable,[4] is a sound procedure.

SOUND
TRUTH-PRESERVING

The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue.[5] Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

COMPLETENESS

We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, *if KB is true in the* real *world, then any sentence α derived from KB by a sound inference procedure is also true in the real world.* So, while an inference process operates on "syntax"—internal physical configurations such as bits in registers or patterns of electrical blips in brains—the process *corresponds*

[3] The agent can calculate the *probability* that there is a pit in [2,2]; Chapter 13 shows how.

[4] Model checking works if the space of models is finite—for example, in wumpus worlds of fixed size. For arithmetic, on the other hand, the space of models is infinite: even if we restrict ourselves to the integers, there are infinitely many pairs of values for x and y in the sentence $x + y = 4$.

[5] Compare with the case of infinite search spaces in Chapter 3, where depth-first search is not complete.

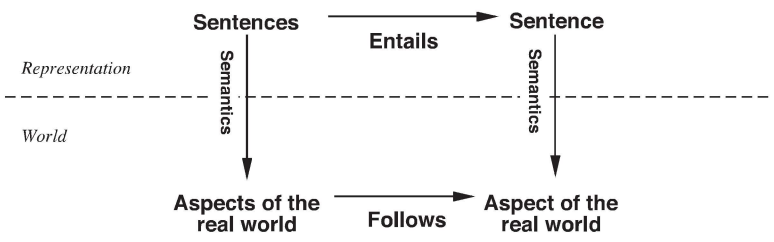


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

to the real-world relationship whereby some aspect of the real world is the case[6] by virtue of other aspects of the real world being the case. This correspondence between world and representation is illustrated in Figure 7.6.

GROUNDING

The final issue to consider is **grounding**—the connection between logical reasoning processes and the real environment in which the agent exists. In particular, *how do we know that KB is true in the real world?* (After all, KB is just "syntax" inside the agent's head.) This is a philosophical question about which many, many books have been written. (See Chapter 26.) A simple answer is that the agent's sensors create the connection. For example, our wumpus-world agent has a smell sensor. The agent program creates a suitable sentence whenever there is a smell. Then, whenever that sentence is in the knowledge base, it is true in the real world. Thus, the meaning and truth of percept sentences are defined by the processes of sensing and sentence construction that produce them. What about the rest of the agent's knowledge, such as its belief that wumpuses cause smells in adjacent squares? This is not a direct representation of a single percept, but a general rule—derived, perhaps, from perceptual experience but not identical to a statement of that experience. General rules like this are produced by a sentence construction process called **learning**, which is the subject of Part V. Learning is fallible. It could be the case that wumpuses cause smells *except on February 29 in leap years*, which is when they take their baths. Thus, KB may not be true in the real world, but with good learning procedures, there is reason for optimism.

7.4 Propositional Logic: A Very Simple Logic

PROPOSITIONAL LOGIC

We now present a simple but powerful logic called **propositional logic**. We cover the syntax of propositional logic and its semantics—the way in which the truth of sentences is determined. Then we look at **entailment**—the relation between a sentence and another sentence that follows from it—and see how this leads to a simple algorithm for logical inference. Everything takes place, of course, in the wumpus world.

[6] As Wittgenstein (1922) put it in his famous *Tractatus*: "The world is everything that is the case."

7.4.1 Syntax

ATOMIC SENTENCES
PROPOSITION SYMBOL

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, $W_{1,3}$ and $North$. The names are arbitrary but are often chosen to have some mnemonic value—we use $W_{1,3}$ to stand for the proposition that the wumpus is in [1,3]. (Remember that symbols such as $W_{1,3}$ are *atomic*, i.e., W, 1, and 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed meanings: $True$ is the always-true proposition and $False$ is the always-false proposition.

COMPLEX SENTENCES
LOGICAL CONNECTIVES

Complex sentences are constructed from simpler sentences, using parentheses and **logical connectives**. There are five connectives in common use:

NEGATION
LITERAL

- $\neg$ (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).

CONJUNCTION

- $\wedge$ (and). A sentence whose main connective is $\wedge$, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The $\wedge$ looks like an "A" for "And.")

DISJUNCTION

- $\vee$ (or). A sentence using $\vee$, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction** of the **disjuncts** $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$. (Historically, the $\vee$ comes from the Latin "vel," which means "or." For most people, it is easier to remember $\vee$ as an upside-down $\wedge$.)

IMPLICATION
PREMISE
CONCLUSION
RULES

- $\Rightarrow$ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \wedge P_{3,1})$, and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as $\supset$ or $\rightarrow$.

BICONDITIONAL

- $\Leftrightarrow$ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**. Some other books write this as $\equiv$.

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
& \mid & \neg\ \mathit{Sentence} \\
& \mid & \mathit{Sentence} \wedge \mathit{Sentence} \\
& \mid & \mathit{Sentence} \vee \mathit{Sentence} \\
& \mid & \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
& \mid & \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
\text{OPERATOR PRECEDENCE} & : & \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 7.7 gives a formal grammar of propositional logic; see page 1060 if you are not familiar with the BNF notation. The BNF grammar by itself is ambiguous; a sentence with several operators can be parsed by the grammar in multiple ways. To eliminate the ambiguity we define a precedence for each operator. The "not" operator ($\neg$) has the highest precedence, which means that in the sentence $\neg A \wedge B$ the $\neg$ binds most tightly, giving us the equivalent of $(\neg A) \wedge B$ rather than $\neg(A \wedge B)$. (The notation for ordinary arithmetic is the same: $-2+4$ is 2, not –6.) When in doubt, use parentheses to make sure of the right interpretation. Square brackets mean the same thing as parentheses; the choice of square brackets or parentheses is solely to make it easier for a human to read a sentence.

7.4.2 Semantics

Having specified the syntax of propositional logic, we now specify its semantics. The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the **truth value**—$true$ or $false$—for every proposition symbol. For example, if the sentences in the knowledge base make use of the proposition symbols $P_{1,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is

TRUTH VALUE

$$m_1 = \{P_{1,2} = false,\ P_{2,2} = false,\ P_{3,1} = true\}\ .$$

With three proposition symbols, there are $2^3 = 8$ possible models—exactly those depicted in Figure 7.5. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. $P_{1,2}$ is just a symbol; it might mean "there is a pit in [1,2]" or "I'm in Paris today and tomorrow."

The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives. Atomic sentences are easy:

- $True$ is true in every model and $False$ is false in every model.
- The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m_1 given earlier, $P_{1,2}$ is false.

For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here "iff" means "if and only if"):

- $\neg P$ is true iff P is false in m.
- $P \wedge Q$ is true iff both P and Q are true in m.
- $P \vee Q$ is true iff either P or Q is true in m.
- $P \Rightarrow Q$ is true unless P is true and Q is false in m.
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m.

TRUTH TABLE

The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be computed with respect to any model m by a simple recursive evaluation. For example,

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

the sentence $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, evaluated in m_1, gives $true \wedge (false \vee true) = true \wedge true = true$. Exercise 7.3 asks you to write the algorithm PL-TRUE?(s, m), which computes the truth value of a propositional logic sentence s in a model m.

The truth tables for "and," "or," and "not" are in close accord with our intuitions about the English words. The main point of possible confusion is that $P \vee Q$ is true when P is true or Q is true *or both*. A different connective, called "exclusive or" ("xor" for short), yields false when both disjuncts are true.[7] There is no consensus on the symbol for exclusive or; some choices are $\dot{\vee}$ or $\neq$ or $\oplus$.

The truth table for $\Rightarrow$ may not quite fit one's intuitive understanding of "P implies Q" or "if P then Q." For one thing, propositional logic does not require any relation of *causation* or *relevance* between P and Q. The sentence "5 is odd implies Tokyo is the capital of Japan" is a true sentence of propositional logic (under the normal interpretation), even though it is a decidedly odd sentence of English. Another point of confusion is that any implication is true whenever its antecedent is false. For example, "5 is even implies Sam is smart" is true, regardless of whether Sam is smart. This seems bizarre, but it makes sense if you think of "$P \Rightarrow Q$" as saying, "If P is true, then I am claiming that Q is true. Otherwise I am making no claim." The only way for this sentence to be *false* is if P is true but Q is false.

The biconditional, $P \Leftrightarrow Q$, is true whenever both $P \Rightarrow Q$ and $Q \Rightarrow P$ are true. In English, this is often written as "P if and only if Q." Many of the rules of the wumpus world are best written using $\Leftrightarrow$. For example, a square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need a biconditional,

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

where $B_{1,1}$ means that there is a breeze in [1,1].

7.4.3 A simple knowledge base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We focus first on the *immutable* aspects of the wumpus world, leaving the mutable aspects for a later section. For now, we need the following symbols for each $[x, y]$ location:

[7] Latin has a separate word, *aut*, for exclusive or.

$P_{x,y}$ is true if there is a pit in $[x, y]$.
$W_{x,y}$ is true if there is a wumpus in $[x, y]$, dead or alive.
$B_{x,y}$ is true if the agent perceives a breeze in $[x, y]$.
$S_{x,y}$ is true if the agent perceives a stench in $[x, y]$.

The sentences we write will suffice to derive $\neg P_{1,2}$ (there is no pit in [1,2]), as was done informally in Section 7.3. We label each sentence R_i so that we can refer to them:

- There is no pit in [1,1]:
$$R_1 : \quad \neg P_{1,1} .$$
- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:
$$R_2 : \quad B_{1,1} \quad \Leftrightarrow \quad (P_{1,2} \vee P_{2,1}) .$$
$$R_3 : \quad B_{2,1} \quad \Leftrightarrow \quad (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$
- The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3(b).
$$R_4 : \quad \neg B_{1,1} .$$
$$R_5 : \quad B_{2,1} .$$

7.4.4 A simple inference procedure

Our goal now is to decide whether $KB \models \alpha$ for some sentence α. For example, is $\neg P_{1,2}$ entailed by our KB? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of $true$ or $false$ to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$, and $P_{3,1}$. With seven symbols, there are $2^7 = 128$ possible models; in three of these, KB is true (Figure 7.9). In those three models, $\neg P_{1,2}$ is true, hence there is no pit in [1,2]. On the other hand, $P_{2,2}$ is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2].

Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10. Like the BACKTRACKING-SEARCH algorithm on page 215, TT-ENTAILS? performs a recursive enumeration of a finite space of assignments to symbols. The algorithm is **sound** because it implements directly the definition of entailment, and **complete** because it works for any KB and α and always terminates—there are only finitely many models to examine.

Of course, "finitely many" is not always the same as "few." If KB and α contain n symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$. (The space complexity is only $O(n)$ because the enumeration is depth-first.) Later in this chapter we show algorithms that are much more efficient in many cases. Unfortunately, propositional entailment is co-NP-complete (i.e., probably no easier than NP-complete—see Appendix A), so *every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input.*

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. *KB* is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

function TT-ENTAILS?(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, *symbols*, { })

---

function TT-CHECK-ALL(KB, α, *symbols*, *model*) **returns** *true* or *false*
  if EMPTY?(*symbols*) **then**
    if PL-TRUE?(KB, *model*) **then return** PL-TRUE?(α, *model*)
    else return *true* // *when KB is false, always return true*
  else do
    $P \leftarrow$ FIRST(*symbols*)
    rest ← REST(*symbols*)
    return (TT-CHECK-ALL(KB, α, *rest*, *model* $\cup \{P = true\}$)
      and
      TT-CHECK-ALL(KB, α, *rest*, *model* $\cup \{P = false\}$))

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

7.5 Propositional Theorem Proving

So far, we have shown how to determine entailment by *model checking*: enumerating models and showing that the sentence must hold in all models. In this section, we show how entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.

THEOREM PROVING

Before we plunge into the details of theorem-proving algorithms, we will need some additional concepts related to entailment. The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 7.11. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:

LOGICAL EQUIVALENCE

$$\alpha \equiv \beta \quad \text{if and only if} \quad \alpha \models \beta \text{ and } \beta \models \alpha \ .$$

The second concept we will need is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true. Because the sentence $True$ is true in all models, every valid sentence is logically equivalent to $True$. What good are valid sentences? From our definition of entailment, we can derive the **deduction theorem**, which was known to the ancient Greeks:

VALIDITY

TAUTOLOGY

DEDUCTION THEOREM

For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

(Exercise 7.5 asks for a proof.) Hence, we can decide if $\alpha \models \beta$ by checking that $(\alpha \Rightarrow \beta)$ is true in every model—which is essentially what the inference algorithm in Figure 7.10 does—

or by proving that $(\alpha \Rightarrow \beta)$ is equivalent to $True$. Conversely, the deduction theorem states that every valid implication sentence describes a legitimate inference.

SATISFIABILITY

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, *some* model. For example, the knowledge base given earlier, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 6 ask whether the constraints are satisfiable by some assignment.

SAT

Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. We also have the following useful result:

$\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

REDUCTIO AD ABSURDUM
REFUTATION
CONTRADICTION

Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ corresponds exactly to the standard mathematical proof technique of *reductio ad absurdum* (literally, "reduction to an absurd thing"). It is also called proof by **refutation** or proof by **contradiction**. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α. This contradiction is exactly what is meant by saying that the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

7.5.1 Inference and proofs

INFERENCE RULES
PROOF
MODUS PONENS

This section covers **inference rules** that can be applied to derive a **proof**—a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens** (Latin for *mode that affirms*) and is written

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}\ .$$

The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred. For example, if $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \wedge WumpusAlive)$ are given, then $Shoot$ can be inferred.

AND-ELIMINATION

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}\ .$$

For example, from $(WumpusAhead \wedge WumpusAlive)$, $WumpusAlive$ can be inferred.

By considering the possible truth values of α and β, one can show easily that Modus Ponens and And-Elimination are sound once and for all. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.

All of the logical equivalences in Figure 7.11 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}\ .$$

Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain $\alpha \Rightarrow \beta$ and α from β.

Let us see how these inference rules and equivalences can be used in the wumpus world. We start with the knowledge base containing R_1 through R_5 and show how to prove $\neg P_{1,2}$, that is, there is no pit in [1,2]. First, we apply biconditional elimination to R_2 to obtain

$$R_6 : \quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Then we apply And-Elimination to R_6 to obtain

$$R_7 : \quad ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Logical equivalence for contrapositives gives

$$R_8 : \quad (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})) .$$

Now we can apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

$$R_9 : \quad \neg(P_{1,2} \vee P_{2,1}) .$$

Finally, we apply De Morgan's rule, giving the conclusion

$$R_{10} : \quad \neg P_{1,2} \wedge \neg P_{2,1} .$$

That is, neither [1,2] nor [2,1] contains a pit.

We found this proof by hand, but we can apply any of the search algorithms in Chapter 3 to find a sequence of steps that constitutes a proof. We just need to define a proof problem as follows:

- INITIAL STATE: the initial knowledge base.
- ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.

Thus, searching for proofs is an alternative to enumerating models. In many practical cases *finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are.* For example, the proof given earlier leading to $\neg P_{1,2} \wedge \neg P_{2,1}$ does not mention the propositions $B_{2,1}$, $P_{1,1}$, $P_{2,2}$, or $P_{3,1}$. They can be ignored because the goal proposition, $P_{1,2}$, appears only in sentence R_2; the other propositions in R_2 appear only in R_4 and R_2; so R_1, R_3, and R_5 have no bearing on the proof. The same would hold even if we added a million more sentences to the knowledge base; the simple truth-table algorithm, on the other hand, would be overwhelmed by the exponential explosion of models.

MONOTONICITY

One final property of logical systems is **monotonicity**, which says that the set of entailed sentences can only *increase* as information is added to the knowledge base.[8] For any sentences α and β,

$$\text{if} \quad KB \models \alpha \quad \text{then} \quad KB \wedge \beta \models \alpha .$$

[8] **Nonmonotonic** logics, which violate the monotonicity property, capture a common property of human reasoning: changing one's mind. They are discussed in Section 12.6.

For example, suppose the knowledge base contains the additional assertion β stating that there are exactly eight pits in the world. This knowledge might help the agent draw *additional* conclusions, but it cannot invalidate any conclusion α already inferred—such as the conclusion that there is no pit in [1,2]. Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base—the conclusion of the rule must follow *regardless of what else is in the knowledge base*.

7.5.2 Proof by resolution

We have argued that the inference rules covered so far are *sound*, but we have not discussed the question of *completeness* for the inference algorithms that use them. Search algorithms such as iterative deepening search (page 89) are complete in the sense that they will find any reachable goal, but if the available inference rules are inadequate, then the goal is not reachable—no proof exists that uses only those inference rules. For example, if we removed the biconditional elimination rule, the proof in the preceding section would not go through. The current section introduces a single inference rule, **resolution**, that yields a complete inference algorithm when coupled with any complete search algorithm.

We begin by using a simple version of the resolution rule in the wumpus world. Let us consider the steps leading up to Figure 7.4(a): the agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the knowledge base:

$$R_{11} : \quad \neg B_{1,2} \ .$$
$$R_{12} : \quad B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3}) \ .$$

By the same process that led to R_{10} earlier, we can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pitless):

$$R_{13} : \quad \neg P_{2,2} \ .$$
$$R_{14} : \quad \neg P_{1,3} \ .$$

We can also apply biconditional elimination to R_3, followed by Modus Ponens with R_5, to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

$$R_{15} : \quad P_{1,1} \vee P_{2,2} \vee P_{3,1} \ .$$

Now comes the first application of the resolution rule: the literal $\neg P_{2,2}$ in R_{13} *resolves with* the literal $P_{2,2}$ in R_{15} to give the **resolvent**

RESOLVENT

$$R_{16} : \quad P_{1,1} \vee P_{3,1} \ .$$

In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]. Similarly, the literal $\neg P_{1,1}$ in R_1 resolves with the literal $P_{1,1}$ in R_{16} to give

$$R_{17} : \quad P_{3,1} \ .$$

In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last two inference steps are examples of the **unit resolution** inference rule,

UNIT RESOLUTION

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k} \ ,$$

where each ℓ is a literal and ℓ_i and m are **complementary literals** (i.e., one is the negation

COMPLEMENTARY LITERALS

CLAUSE

of the other). Thus, the unit resolution rule takes a **clause**—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as a **unit clause**.

UNIT CLAUSE

RESOLUTION

The unit resolution rule can be generalized to the full **resolution** rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n} ,$$

where ℓ_i and m_j are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals. For example, we have

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}} .$$

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.[9] The removal of multiple copies of literals is called **factoring**. For example, if we resolve $(A \vee B)$ with $(A \vee \neg B)$, we obtain $(A \vee A)$, which is reduced to just A.

FACTORING

The *soundness* of the resolution rule can be seen easily by considering the literal ℓ_i that is complementary to literal m_j in the other clause. If ℓ_i is true, then m_j is false, and hence $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$ must be true, because $m_1 \vee \cdots \vee m_n$ is given. If ℓ_i is false, then $\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k$ must be true because $\ell_1 \vee \cdots \vee \ell_k$ is given. Now ℓ_i is either true or false, so one or other of these conclusions holds—exactly as the resolution rule states.

What is more surprising about the resolution rule is that it forms the basis for a family of *complete* inference procedures. *A resolution-based theorem prover can, for any sentences α and β in propositional logic, decide whether $\alpha \models \beta$.* The next two subsections explain how resolution accomplishes this.

Conjunctive normal form

The resolution rule applies only to clauses (that is, disjunctions of literals), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that *every sentence of propositional logic is logically equivalent to a conjunction of clauses.* A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** (see Figure 7.14). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ into CNF. The steps are as follows:

CONJUNCTIVE NORMAL FORM

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$
2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) .$$

[9] If a clause is viewed as a *set* of literals, then this restriction is automatically respected. Using set notation for clauses makes the resolution rule much cleaner, at the cost of introducing additional notation.

3. CNF requires $\neg$ to appear only in literals, so we "move $\neg$ inwards" by repeated application of the following equivalences from Figure 7.11:

   $\neg(\neg\alpha) \equiv \alpha$ (double-negation elimination)
   $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ (De Morgan)
   $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ (De Morgan)

   In the example, we require just one application of the last rule:

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

4. Now we have a sentence containing nested $\wedge$ and $\vee$ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing $\vee$ over $\wedge$ wherever possible.

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

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction introduced on page 250. That is, to show that $KB \models \alpha$, we show that $(KB \wedge \neg\alpha)$ is unsatisfiable. We do this by proving a contradiction.

A resolution algorithm is shown in Figure 7.12. First, $(KB \wedge \neg\alpha)$ is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

- there are no new clauses that can be added, in which case KB does not entail α; or,
- two clauses resolve to yield the *empty* clause, in which case KB entails α.

The empty clause—a disjunction of no disjuncts—is equivalent to $False$ because a disjunction is true only if at least one of its disjuncts is true. Another way to see that an empty clause represents a contradiction is to observe that it arises only from resolving two complementary unit clauses such as P and $\neg P$.

We can apply the resolution procedure to a very simple inference in the wumpus world. When the agent is in [1,1], there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

and we wish to prove α which is, say, $\neg P_{1,2}$. When we convert $(KB \wedge \neg\alpha)$ into CNF, we obtain the clauses shown at the top of Figure 7.13. The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square. Inspection of Figure 7.13 reveals that many resolution steps are pointless. For example, the clause $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ is equivalent to $True \vee P_{1,2}$ which is equivalent to $True$. Deducing that $True$ is true is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded.

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each pair of clauses C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

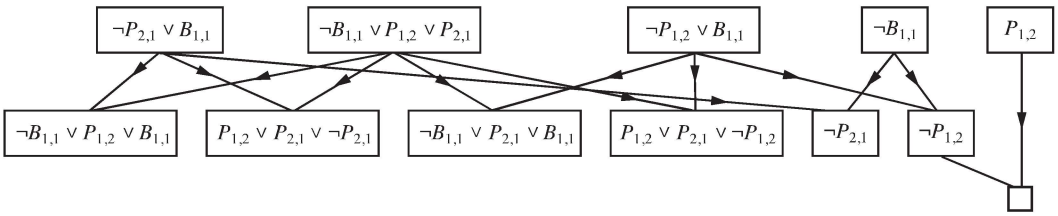


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

Completeness of resolution

RESOLUTION CLOSURE

To conclude our discussion of resolution, we now show why PL-RESOLUTION is complete. To do this, we introduce the **resolution closure** $RC(S)$ of a set of clauses S, which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable *clauses*. It is easy to see that $RC(S)$ must be finite, because there are only finitely many distinct clauses that can be constructed out of the symbols $P_1, \ldots, P_k$ that appear in S. (Notice that this would not be true without the factoring step that removes multiple copies of literals.) Hence, PL-RESOLUTION always terminates.

GROUND RESOLUTION THEOREM

The completeness theorem for resolution in propositional logic is called the **ground resolution theorem**:

> If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

This theorem is proved by demonstrating its contrapositive: if the closure $RC(S)$ does *not*

contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for $P_1, \ldots, P_k$. The construction procedure is as follows:

For i from 1 to k,

- If a clause in $RC(S)$ contains the literal $\neg P_i$ and all its other literals are false under the assignment chosen for $P_1, \ldots, P_{i-1}$, then assign $false$ to P_i.
- Otherwise, assign $true$ to P_i.

This assignment to $P_1, \ldots, P_k$ is a model of S. To see this, assume the opposite—that, at some stage i in the sequence, assigning symbol P_i causes some clause C to become false. For this to happen, it must be the case that all the *other* literals in C must already have been falsified by assignments to $P_1, \ldots, P_{i-1}$. Thus, C must now look like either $(false \vee false \vee \cdots false \vee P_i)$ or like $(false \vee false \vee \cdots false \vee \neg P_i)$. If just one of these two is in $RC(S)$, then the algorithm will assign the appropriate truth value to P_i to make C true, so C can only be falsified if *both* of these clauses are in $RC(S)$. Now, since $RC(S)$ is closed under resolution, it will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to $P_1, \ldots, P_{i-1}$. This contradicts our assumption that the first falsified clause appears at stage i. Hence, we have proved that the construction never falsifies a clause in $RC(S)$; that is, it produces a model of $RC(S)$ and thus a model of S itself (since S is contained in $RC(S)$).

7.5.3 Horn clauses and definite clauses

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.

DEFINITE CLAUSE

One such restricted form is the **definite clause**, which is a disjunction of literals of which *exactly one is positive*. For example, the clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ is a definite clause, whereas $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ is not.

HORN CLAUSE

Slightly more general is the **Horn clause**, which is a disjunction of literals of which *at most one is positive*. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause.

GOAL CLAUSES

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.13.) For example, the definite clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ can be written as the implication $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy. In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as $L_{1,1}$, is called a **fact**. It too can be written in implication form as $True \Rightarrow L_{1,1}$, but it is simpler to write just $L_{1,1}$.

BODY
HEAD
FACT

$$
\begin{aligned}
\mathit{CNFSentence} &\rightarrow \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} &\rightarrow \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Literal} &\rightarrow \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} &\rightarrow P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} &\rightarrow \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{aligned}
$$

Figure 7.14 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as $A \wedge B \Rightarrow C$ is still a definite clause when it is written as $\neg A \vee \neg B \vee C$, but only the former is considered the canonical form for definite clauses. One more class is the *k*-CNF sentence, which is a CNF sentence where each clause has at most *k* literals.

FORWARD-CHAINING
BACKWARD-CHAINING

2. Inference with Horn clauses can be done through the **forward-chaining** and **backward-chaining** algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for **logic programming**, which is discussed in Chapter 9.
3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base—a pleasant surprise.

7.5.4 Forward and backward chaining

The forward-chaining algorithm PL-FC-ENTAILS?(KB, q) determines if a single proposition symbol q—the query—is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if $L_{1,1}$ and Breeze are known and $(L_{1,1} \wedge \mathit{Breeze}) \Rightarrow B_{1,1}$ is in the knowledge base, then $B_{1,1}$ can be added. This process continues until the query q is added or until no further inferences can be made. The detailed algorithm is shown in Figure 7.15; the main point to remember is that it runs in linear time.

The best way to understand the algorithm is through an example and a picture. Figure 7.16(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7.16(b) shows the same knowledge base drawn as an **AND–OR graph** (see Chapter 4). In AND–OR graphs, multiple links joined by an arc indicate a conjunction—every link must be proved—while multiple links without an arc indicate a disjunction—any link can be proved. It is easy to see how forward chaining works in the graph. The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p ← POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false
```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the *inferred* table (after the algorithm reaches a **fixed point** where no new inferences are possible). The table contains *true* for each symbol inferred during the process, and *false* for all other symbols. We can view the table as a logical model; moreover, *every definite clause in the original KB is true in this model.* To see this, assume the opposite, namely that some clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in the model. Then $a_1 \wedge \ldots \wedge a_k$ must be true in the model and b must be false in the model. But this contradicts our assumption that the algorithm has reached a fixed point! We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

FIXED POINT

Forward chaining is an example of the general concept of **data-driven** reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind. For example, the wumpus agent might TELL its percepts to the knowledge base using

DATA-DRIVEN

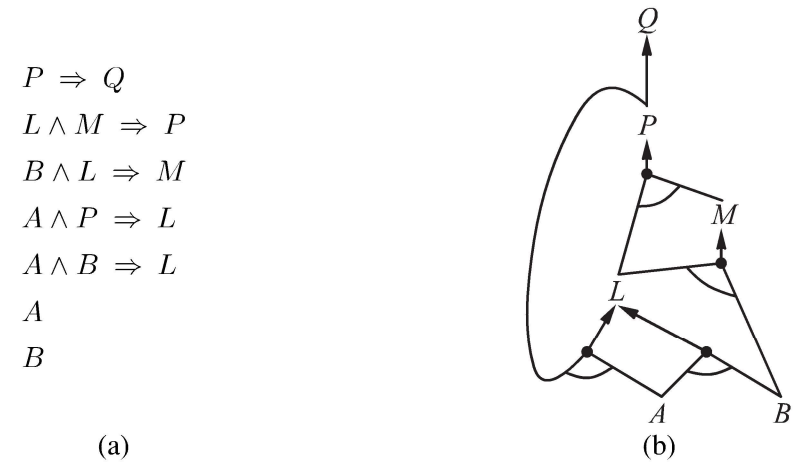


Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

an incremental forward-chaining algorithm in which new facts can be added to the agenda to initiate new inferences. In humans, a certain amount of data-driven reasoning occurs as new information arrives. For example, if I am indoors and hear rain starting to fall, it might occur to me that the picnic will be canceled. Yet it will probably not occur to me that the seventeenth petal on the largest rose in my neighbor's garden will get wet; humans keep forward chaining under careful control, lest they be swamped with irrelevant consequences.

The backward-chaining algorithm, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7.16, it works back down the graph until it reaches a set of known facts, A and B, that forms the basis for a proof. The algorithm is essentially identical to the AND-OR-GRAPH-SEARCH algorithm in Figure 4.11. As with forward chaining, an efficient implementation runs in linear time.

GOAL-DIRECTED REASONING

Backward chaining is a form of **goal-directed reasoning**. It is useful for answering specific questions such as "What shall I do now?" and "Where are my keys?" Often, the cost of backward chaining is *much less* than linear in the size of the knowledge base, because the process touches only relevant facts.

7.6 EFFECTIVE PROPOSITIONAL MODEL CHECKING

In this section, we describe two families of efficient algorithms for general propositional inference based on model checking: One approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the "technology" of propositional logic. This section can be skimmed on a first reading of the chapter.

The algorithms we describe are for checking satisfiability: the SAT problem. (As noted earlier, testing entailment, $\alpha \models \beta$, can be done by testing *un*satisfiability of $\alpha \wedge \neg\beta$.) We have already noted the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of algorithms closely resemble the backtracking algorithms of Section 6.3 and the local search algorithms of Section 6.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

7.6.1 A complete backtracking algorithm

DAVIS–PUTNAM ALGORITHM

The first algorithm we consider is often called the **Davis–Putnam algorithm**, after the seminal paper by Martin Davis and Hilary Putnam (1960). The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962), so we will call it DPLL after the initials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set of clauses. Like BACKTRACKING-SEARCH and TT-ENTAILS?, it is essentially a recursive, depth-first enumeration of possible models. It embodies three improvements over the simple scheme of TT-ENTAILS?:

- *Early termination*: The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete. For example, the sentence $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of the values of B and C. Similarly, a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.

PURE SYMBOL

- *Pure symbol heuristic*: A **pure symbol** is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals $true$, because doing so can never make a clause false. Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains $B = false$, then the clause $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.
- *Unit clause heuristic*: A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned $false$ by the model. For example, if the model contains $B = true$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to $false$. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
         DPLL(clauses, rest, model ∪ {P=false}))
```

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately (Exercise 7.23). Notice also that assigning one unit clause can create another unit clause—for example, when C is set to *false*, $(C \vee A)$ becomes a unit clause, causing *true* to be assigned to A. This "cascade" of forced assignments is called **unit propagation**. It resembles the process of forward chaining with definite clauses, and indeed, if the CNF expression contains only definite clauses then DPLL essentially replicates forward chaining. (See Exercise 7.24.)

UNIT PROPAGATION

The DPLL algorithm is shown in Figure 7.17, which gives the the essential skeleton of the search process.

What Figure 7.17 does not show are the tricks that enable SAT solvers to scale up to large problems. It is interesting that most of these tricks are in fact rather general, and we have seen them before in other guises:

1. **Component analysis** (as seen with Tasmania in CSPs): As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.
2. **Variable and value ordering** (as seen in Section 6.3.1 for CSPs): Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** (see page 216) suggests choosing the variable that appears most frequently over all remaining clauses.

3. **Intelligent backtracking** (as seen in Section 6.3 for CSPs): Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.
4. **Random restarts** (as seen on page 124 for hill-climbing): Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.
5. **Clever indexing** (as seen in many algorithms): The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as "the set of clauses in which variable X_i appears as a positive literal." This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds.

With these enhancements, modern solvers can handle problems with tens of millions of variables. They have revolutionized areas such as hardware verification and security protocol verification, which previously required laborious, hand-guided proofs.

7.6.2 Local search algorithms

We have seen several local search algorithms so far in this book, including HILL-CLIMBING (page 122) and SIMULATED-ANNEALING (page 126). These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact, this is exactly the measure used by the MIN-CONFLICTS algorithm for CSPs (page 221). All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required. In recent years, there has been a great deal of experimentation to find a good balance between greediness and randomness.

One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT (Figure 7.18). On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a "min-conflicts" step that minimizes the number of unsatisfied clauses in the new state and (2) a "random walk" step that picks the symbol randomly.

When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns *failure*, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time. If we set $max\_flips = \infty$ and $p > 0$, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
      if model satisfies clauses then return model
      clause ← a randomly selected clause from clauses that is false in model
      with probability p flip the value in model of a randomly selected symbol from clause
      else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

upon the solution. Alas, if max_flips is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

For this reason, WALKSAT is most useful when we expect a solution to exist—for example, the problems discussed in Chapters 3 and 6 usually have solutions. On the other hand, WALKSAT cannot always detect *unsatisfiability*, which is required for deciding entailment. For example, an agent cannot *reliably* use WALKSAT to prove that a square is safe in the wumpus world. Instead, it can say, "I thought about it for an hour and couldn't come up with a possible world in which the square *isn't* safe." This may be a good empirical indicator that the square is safe, but it's certainly not a proof.

7.6.3 The landscape of random SAT problems

Some SAT problems are harder than others. *Easy* problems can be solved by any old algorithm, but because we know that SAT is NP-complete, at least some problem instances must require exponential run time. In Chapter 6, we saw some surprising discoveries about certain kinds of problems. For example, the n-queens problem—thought to be quite tricky for backtracking search algorithms—turned out to be trivially easy for local search methods, such as min-conflicts. This is because solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby. Thus, n-queens is easy because it is **underconstrained**.

UNDERCONSTRAINED

When we look at satisfiability problems in conjunctive normal form, an underconstrained problem is one with relatively *few* clauses constraining the variables. For example, here is a randomly generated 3-CNF sentence with five symbols and five clauses:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \\ \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C) \ .$$

Sixteen of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model. This is an easy satisfiability problem, as are

most such underconstrained problems. On the other hand, an *overconstrained* problem has many clauses relative to the number of variables and is likely to have no solutions.

To go beyond these basic intuitions, we must define exactly how random sentences are generated. The notation $CNF_k(m, n)$ denotes a k-CNF sentence with m clauses and n symbols, where the clauses are chosen uniformly, independently, and without replacement from among all clauses with k different literals, which are positive or negative at random. (A symbol may not appear twice in a clause, nor may a clause appear twice in a sentence.)

Given a source of random sentences, we can measure the probability of satisfiability. Figure 7.19(a) plots the probability for $CNF_3(m, 50)$, that is, sentences with 50 variables and 3 literals per clause, as a function of the clause/symbol ratio, m/n. As we expect, for small m/n the probability of satisfiability is close to 1, and at large m/n the probability is close to 0. The probability drops fairly sharply around $m/n = 4.3$. Empirically, we find that the "cliff" stays in roughly the same place (for $k = 3$) and gets sharper and sharper as n increases. Theoretically, the **satisfiability threshold conjecture** says that for every $k \geq 3$, there is a threshold ratio r_k such that, as n goes to infinity, the probability that $CNF_k(n, rn)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above. The conjecture remains unproven.

SATISFIABILITY THRESHOLD CONJECTURE

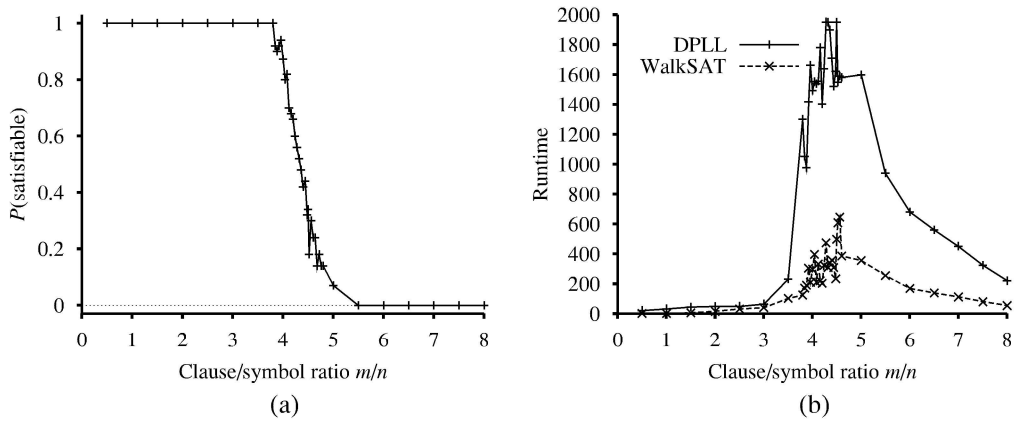


Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median run time (measured in number of recursive calls to DPLL, a good proxy) on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

Now that we have a good idea where the satisfiable and unsatisfiable problems are, the next question is, where are the hard problems? It turns out that they are also often at the threshold value. Figure 7.19(b) shows that 50-symbol problems at the threshold value of 4.3 are about 20 times more difficult to solve than those at a ratio of 3.3. The underconstrained problems are easiest to solve (because it is so easy to guess a solution); the overconstrained problems are not as easy as the underconstrained, but still are much easier than the ones right at the threshold.

7.7 AGENTS BASED ON PROPOSITIONAL LOGIC

In this section, we bring together what we have learned so far in order to construct wumpus world agents that use propositional logic. The first step is to enable the agent to deduce, to the extent possible, the state of the world given its percept history. This requires writing down a complete logical model of the effects of actions. We also show how the agent can keep track of the world efficiently without going back into the percept history for each inference. Finally, we show how the agent can use logical inference to construct plans that are guaranteed to achieve its goals.

7.7.1 The current state of the world

As stated at the beginning of the chapter, a logical agent operates by deducing what to do from a knowledge base of sentences about the world. The knowledge base is composed of axioms—general knowledge about how the world works—and percept sentences obtained from the agent's experience in a particular world. In this section, we focus on the problem of deducing the current state of the wumpus world—where am I, is that square safe, and so on.

We began collecting axioms in Section 7.4.3. The agent knows that the starting square contains no pit ($\neg P_{1,1}$) and no wumpus ($\neg W_{1,1}$). Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus. Thus, we include a large collection of sentences of the following form:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$
$$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$$
$$\ldots$$

The agent also knows that there is exactly one wumpus. This is expressed in two parts. First, we have to say that there is *at least one* wumpus:

$$W_{1,1} \vee W_{1,2} \vee \cdots \vee W_{4,3} \vee W_{4,4} \ .$$

Then, we have to say that there is *at most one* wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free:

$$\neg W_{1,1} \vee \neg W_{1,2}$$
$$\neg W_{1,1} \vee \neg W_{1,3}$$
$$\ldots$$
$$\neg W_{4,3} \vee \neg W_{4,4} \ .$$

So far, so good. Now let's consider the agent's percepts. If there is currently a stench, one might suppose that a proposition $Stench$ should be added to the knowledge base. This is not quite right, however: if there was no stench at the previous time step, then $\neg Stench$ would already be asserted, and the new assertion would simply result in a contradiction. The problem is solved when we realize that a percept asserts something *only about the current time*. Thus, if the time step (as supplied to MAKE-PERCEPT-SENTENCE in Figure 7.1) is 4, then we add

$Stench^4$ to the knowledge base, rather than $Stench$—neatly avoiding any contradiction with $\neg Stench^3$. The same goes for the breeze, bump, glitter, and scream percepts.

The idea of associating propositions with time steps extends to any aspect of the world that changes over time. For example, the initial knowledge base includes $L^0_{1,1}$—the agent is in square $[1,1]$ at time 0—as well as $FacingEast^0$, $HaveArrow^0$, and $WumpusAlive^0$. We use the word **fluent** (from the Latin *fluens*, flowing) to refer an aspect of the world that changes. "Fluent" is a synonym for "state variable," in the sense described in the discussion of factored representations in Section 2.4.7 on page 57. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables**.

FLUENT

ATEMPORAL VARIABLE

We can connect stench and breeze percepts directly to the properties of the squares where they are experienced through the location fluent as follows.[10] For any time step t and any square $[x, y]$, we assert

$$L^t_{x,y} \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L^t_{x,y} \Rightarrow (Stench^t \Leftrightarrow S_{x,y}) .$$

Now, of course, we need axioms that allow the agent to keep track of fluents such as $L^t_{x,y}$. These fluents change as the result of actions taken by the agent, so, in the terminology of Chapter 3, we need to write down the **transition model** of the wumpus world as a set of logical sentences.

First, we need proposition symbols for the occurrences of actions. As with percepts, these symbols are indexed by time; thus, $Forward^0$ means that the agent executes the *Forward* action at time 0. By convention, the percept for a given time step happens first, followed by the action for that time step, followed by a transition to the next time step.

To describe how the world changes, we can try writing **effect axioms** that specify the outcome of an action at the next time step. For example, if the agent is at location $[1,1]$ facing east at time 0 and goes $Forward$, the result is that the agent is in square $[2,1]$ and no longer is in $[1,1]$:

EFFECT AXIOM

$$L^0_{1,1} \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L^1_{2,1} \wedge \neg L^1_{1,1}) . \tag{7.1}$$

We would need one such sentence for each possible time step, for each of the 16 squares, and each of the four orientations. We would also need similar sentences for the other actions: *Grab*, *Shoot*, *Climb*, *TurnLeft*, and *TurnRight*.

Let us suppose that the agent does decide to move *Forward* at time 0 and asserts this fact into its knowledge base. Given the effect axiom in Equation (7.1), combined with the initial assertions about the state at time 0, the agent can now deduce that it is in $[2,1]$. That is, $\text{ASK}(KB, L^1_{2,1}) = true$. So far, so good. Unfortunately, the news elsewhere is less good: if we $\text{ASK}(KB, HaveArrow^1)$, the answer is $false$, that is, the agent cannot prove it still has the arrow; nor can it prove it *doesn't* have it! The information has been lost because the effect axiom fails to state what remains *unchanged* as the result of an action. The need to do this gives rise to the **frame problem**.[11] One possible solution to the frame problem would

FRAME PROBLEM

[10] Section 7.4.3 conveniently glossed over this requirement.

[11] The name "frame problem" comes from "frame of reference" in physics—the assumed stationary background with respect to which motion is measured. It also has an analogy to the frames of a movie, in which normally most of the background stays constant while changes occur in the foreground.

FRAME AXIOM

be to add **frame axioms** explicitly asserting all the propositions that remain the same. For example, for each time t we would have

$$\begin{aligned} &Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1}) \\ &Forward^t \Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1}) \\ &\ldots \end{aligned}$$

where we explicitly mention every proposition that stays unchanged from time t to time $t+1$ under the action $Forward$. Although the agent now knows that it still has the arrow after moving forward and that the wumpus hasn't died or come back to life, the proliferation of frame axioms seems remarkably inefficient. In a world with m different actions and n fluents, the set of frame axioms will be of size $O(mn)$. This specific manifestation of the

REPRESENTATIONAL FRAME PROBLEM

frame problem is sometimes called the **representational frame problem**. Historically, the problem was a significant one for AI researchers; we explore it further in the notes at the end of the chapter.

The representational frame problem is significant because the real world has very many fluents, to put it mildly. Fortunately for us humans, each action typically changes no more

LOCALITY

than some small number k of those fluents—the world exhibits **locality**. Solving the representational frame problem requires defining the transition model with a set of axioms of size

INFERENTIAL FRAME PROBLEM

$O(mk)$ rather than size $O(mn)$. There is also an **inferential frame problem**: the problem of projecting forward the results of a t step plan of action in time $O(kt)$ rather than $O(nt)$.

The solution to the problem involves changing one's focus from writing axioms about *actions* to writing axioms about *fluents*. Thus, for each fluent F, we will have an axiom that defines the truth value of F^{t+1} in terms of fluents (including F itself) at time t and the actions that may have occurred at time t. Now, the truth value of F^{t+1} can be set in one of two ways: either the action at time t causes F to be true at $t+1$, or F was already true at time t and the action at time t does not cause it to be false. An axiom of this form is called a **successor-state**

SUCCESSOR-STATE AXIOM

axiom and has this schema:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t) .$$

One of the simplest successor-state axioms is the one for $HaveArrow$. Because there is no action for reloading, the $ActionCausesF^t$ part goes away and we are left with

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t) . \tag{7.2}$$

For the agent's location, the successor-state axioms are more elaborate. For example, $L^{t+1}_{1,1}$ is true if either (a) the agent moved *Forward* from $[1,2]$ when facing south, or from $[2,1]$ when facing west; or (b) $L^t_{1,1}$ was already true and the action did not cause movement (either because the action was not $Forward$ or because the action bumped into a wall). Written out in propositional logic, this becomes

$$\begin{aligned} L^{t+1}_{1,1} \quad \Leftrightarrow \quad & (L^t_{1,1} \wedge (\neg Forward^t \vee Bump^{t+1})) \\ \vee \quad & (L^t_{1,2} \wedge (South^t \wedge Forward^t)) \\ \vee \quad & (L^t_{2,1} \wedge (West^t \wedge Forward^t)) . \end{aligned} \tag{7.3}$$

Exercise 7.26 asks you to write out axioms for the remaining wumpus world fluents.

Given a complete set of successor-state axioms and the other axioms listed at the beginning of this section, the agent will be able to ASK and answer any answerable question about the current state of the world. For example, in Section 7.2 the initial sequence of percepts and actions is

$$\begin{aligned}
&\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 \ ; \ Forward^0 \\
&\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 \ ; \ TurnRight^1 \\
&\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 \ ; \ TurnRight^2 \\
&\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 \ ; \ Forward^3 \\
&\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 \ ; \ TurnRight^4 \\
&\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 \ ; \ Forward^5 \\
&Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6
\end{aligned}$$

At this point, we have $\text{ASK}(KB, L^6_{1,2}) = true$, so the agent knows where it is. Moreover, $\text{ASK}(KB, W_{1,3}) = true$ and $\text{ASK}(KB, P_{3,1}) = true$, so the agent has found the wumpus and one of the pits. The most important question for the agent is whether a square is OK to move into, that is, the square contains no pit nor live wumpus. It's convenient to add axioms for this, having the form

$$OK^t_{x,y} \ \Leftrightarrow \ \neg P_{x,y} \wedge \neg (W_{x,y} \wedge WumpusAlive^t) \ .$$

Finally, $\text{ASK}(KB, OK^6_{2,2}) = true$, so the square $[2, 2]$ is OK to move into. In fact, given a sound and complete inference algorithm such as DPLL, the agent can answer any answerable question about which squares are OK—and can do so in just a few milliseconds for small-to-medium wumpus worlds.

Solving the representational and inferential frame problems is a big step forward, but a pernicious problem remains: we need to confirm that *all* the necessary preconditions of an action hold for it to have its intended effect. We said that the *Forward* action moves the agent ahead unless there is a wall in the way, but there are many other unusual exceptions that could cause the action to fail: the agent might trip and fall, be stricken with a heart attack, be carried away by giant bats, etc. Specifying all these exceptions is called the **qualification problem**. There is no complete solution within logic; system designers have to use good judgment in deciding how detailed they want to be in specifying their model, and what details they want to leave out. We will see in Chapter 13 that probability theory allows us to summarize all the exceptions without explicitly naming them.

QUALIFICATION PROBLEM

7.7.2 A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straightforwardly with condition–action rules and with problem-solving algorithms from Chapters 3 and 4 to produce a **hybrid agent** for the wumpus world. Figure 7.20 shows one possible way to do this. The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the *atemporal* axioms—those that don't depend on t, such as the axiom relating the breeziness of squares to the presence of pits. At each time step, the new percept sentence is added along with all the axioms that depend on t, such

HYBRID AGENT

as the successor-state axioms. (The next section explains why the agent doesn't need axioms for *future* time steps.) Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited.

The main body of the agent program constructs a plan based on a decreasing priority of goals. First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave. Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares. Route planning is done with A* search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations. These are determined by asking where $\text{ASK}(KB, \neg W_{x,y})$ is false—that is, where it is *not* known that there is *not* a wumpus. The function PLAN-SHOT (not shown) uses PLAN-ROUTE to plan a sequence of actions that will line up this shot. If this fails, the program looks for a square to explore that is not provably unsafe—that is, a square for which $\text{ASK}(KB, \neg OK^t_{x,y})$ returns false. If there is no such square, then the mission is impossible and the agent retreats to $[1, 1]$ and climbs out of the cave.

7.7.3 Logical state estimation

The agent program in Figure 7.20 works quite well, but it has one major weakness: as time goes by, the computational expense involved in the calls to ASK goes up and up. This happens mainly because the required inferences have to go back further and further in time and involve more and more proposition symbols. Obviously, this is unsustainable—we cannot have an agent whose time to process each percept grows in proportion to the length of its life! What we really need is a *constant* update time—that is, independent of t. The obvious answer is to save, or **cache**, the results of inference, so that the inference process at the next time step can build on the results of earlier steps instead of having to start again from scratch.

CACHING

As we saw in Section 4.4, the past history of percepts and all their ramifications can be replaced by the **belief state**—that is, some representation of the set of all possible current states of the world.[12] The process of updating the belief state as new percepts arrive is called **state estimation**. Whereas in Section 4.4 the belief state was an explicit list of states, here we can use a logical sentence involving the proposition symbols associated with the current time step, as well as the atemporal symbols. For example, the logical sentence

$$WumpusAlive^1 \wedge L^1_{2,1} \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \tag{7.4}$$

represents the set of all states at time 1 in which the wumpus is alive, the agent is at $[2, 1]$, that square is breezy, and there is a pit in $[3, 1]$ or $[2, 2]$ or both.

Maintaining an exact belief state as a logical formula turns out not to be easy. If there are n fluent symbols for time t, then there are 2^n possible states—that is, assignments of truth values to those symbols. Now, the set of belief states is the powerset (set of all subsets) of the set of physical states. There are 2^n physical states, hence 2^{2^n} belief states. Even if we used the most compact possible encoding of logical formulas, with each belief state represented

[12] We can think of the percept history itself as a representation of the belief state, but one that makes inference increasingly expensive as the history gets longer.

```
function HYBRID-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter,bump,scream]
  persistent: KB, a knowledge base, initially the atemporal "wumpus physics"
              t, a counter, initially 0, indicating time
              plan, an action sequence, initially empty

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL the KB the temporal "physics" sentences for time t
  safe ← {[x, y] : ASK(KB, OK^t_{x,y}) = true}
  if ASK(KB, Glitter^t) = true then
    plan ← [Grab] + PLAN-ROUTE(current, {[1,1]}, safe) + [Climb]
  if plan is empty then
    unvisited ← {[x, y] : ASK(KB, L^{t'}_{x,y}) = false for all t' ≤ t}
    plan ← PLAN-ROUTE(current, unvisited ∩ safe, safe)
  if plan is empty and ASK(KB, HaveArrow^t) = true then
    possible_wumpus ← {[x, y] : ASK(KB, ¬ W_{x,y}) = false}
    plan ← PLAN-SHOT(current, possible_wumpus, safe)
  if plan is empty then   // no choice but to take a risk
    not_unsafe ← {[x, y] : ASK(KB, ¬ OK^t_{x,y}) = false}
    plan ← PLAN-ROUTE(current, unvisited ∩ not_unsafe, safe)
  if plan is empty then
    plan ← PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
  action ← POP(plan)
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action

function PLAN-ROUTE(current,goals,allowed) returns an action sequence
  inputs: current, the agent's current position
          goals, a set of squares; try to plan a route to one of them
          allowed, a set of squares that can form part of the route

  problem ← ROUTE-PROBLEM(current, goals,allowed)
  return A*-GRAPH-SEARCH(problem)
```

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to decide what actions to take.

by a unique binary number, we would need numbers with $\log_2(2^{2^n}) = 2^n$ bits to label the current belief state. That is, exact state estimation may require logical formulas whose size is exponential in the number of symbols.

One very common and natural scheme for *approximate* state estimation is to represent belief states as conjunctions of literals, that is, 1-CNF formulas. To do this, the agent program simply tries to prove X^t and $\neg X^t$ for each symbol X^t (as well as each atemporal symbol whose truth value is not yet known), given the belief state at $t-1$. The conjunction of

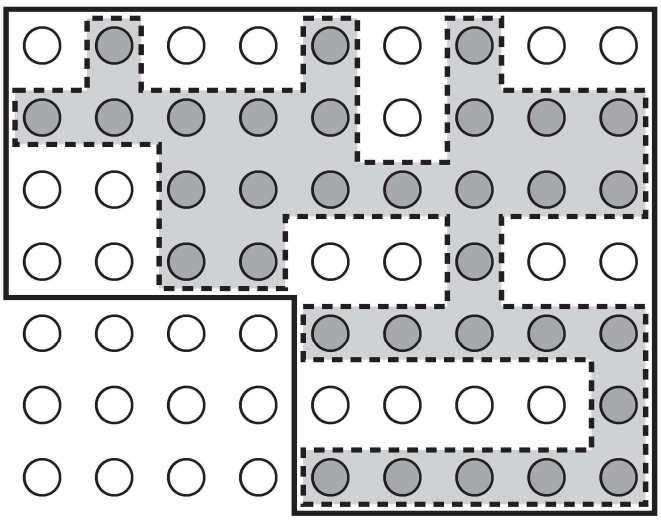


Figure 7.21 Depiction of a 1-CNF belief state (bold outline) as a simply representable, conservative approximation to the exact (wiggly) belief state (shaded region with dashed outline). Each possible world is shown as a circle; the shaded ones are consistent with all the percepts.

provable literals becomes the new belief state, and the previous belief state is discarded.

It is important to understand that this scheme may lose some information as time goes along. For example, if the sentence in Equation (7.4) were the true belief state, then neither $P_{3,1}$ nor $P_{2,2}$ would be provable individually and neither would appear in the 1-CNF belief state. (Exercise 7.27 explores one possible solution to this problem.) On the other hand, because every literal in the 1-CNF belief state is proved from the previous belief state, and the initial belief state is a true assertion, we know that entire 1-CNF belief state must be true. Thus, *the set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history.* As illustrated in Figure 7.21, the 1-CNF belief state acts as a simple outer envelope, or **conservative approximation**, around the exact belief state. We see this idea of conservative approximations to complicated sets as a recurring theme in many areas of AI.

CONSERVATIVE APPROXIMATION

7.7.4 Making plans by propositional inference

The agent in Figure 7.20 uses logical inference to determine which squares are safe, but uses A* search to make plans. In this section, we show how to make plans by logical inference. The basic idea is very simple:

1. Construct a sentence that includes
   (a) $Init^0$, a collection of assertions about the initial state;
   (b) $Transition^1, \ldots, Transition^t$, the successor-state axioms for all possible actions at each time up to some maximum time t;
   (c) the assertion that the goal is achieved at time t: $HaveGold^t \wedge ClimbedOut^t$.

2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the planning problem is impossible.
3. Assuming a model is found, extract from the model those variables that represent actions and are assigned $true$. Together they represent a plan to achieve the goals.

A propositional planning procedure, SATPLAN, is shown in Figure 7.22. It implements the basic idea just given, with one twist. Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps t, up to some maximum conceivable plan length $T_{\max}$. In this way, it is guaranteed to find the shortest plan if one exists. Because of the way SATPLAN searches for a solution, this approach cannot be used in a partially observable environment; SATPLAN would just set the unobservable variables to the values it needs to create a solution.

```
function SATPLAN(init, transition, goal, T_max) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
          T_max, an upper limit for plan length

  for t = 0 to T_max do
    cnf ← TRANSLATE-TO-SAT(init, transition, goal, t)
    model ← SAT-SOLVER(cnf)
    if model is not null then
      return EXTRACT-SOLUTION(model)
  return failure
```

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t. If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned $true$ in the model. If no model exists, then the process is repeated with the goal moved one step later.

The key step in using SATPLAN is the construction of the knowledge base. It might seem, on casual inspection, that the wumpus world axioms in Section 7.7.1 suffice for steps 1(a) and 1(b) above. There is, however, a significant difference between the requirements for entailment (as tested by ASK) and those for satisfiability. Consider, for example, the agent's location, initially $[1,1]$, and suppose the agent's unambitious goal is to be in $[2,1]$ at time 1. The initial knowledge base contains $L^0_{1,1}$ and the goal is $L^1_{2,1}$. Using ASK, we can prove $L^1_{2,1}$ if *Forward*^{0} is asserted, and, reassuringly, we cannot prove $L^1_{2,1}$ if, say, *Shoot*^{0} is asserted instead. Now, SATPLAN will find the plan [*Forward*^{0}]; so far, so good. Unfortunately, SATPLAN also finds the plan [*Shoot*^{0}]. How could this be? To find out, we inspect the model that SATPLAN constructs: it includes the assignment $L^0_{2,1}$, that is, the agent can be in $[2,1]$ at time 1 by being there at time 0 and shooting. One might ask, "Didn't we say the agent is in $[1,1]$ at time 0?" Yes, we did, but we didn't tell the agent that it can't be in two places at once! For entailment, $L^0_{2,1}$ is unknown and cannot, therefore, be used in a proof; for satisfiability,

on the other hand, $L^0_{2,1}$ is unknown and can, therefore, be set to whatever value helps to make the goal true. For this reason, SATPLAN is a good debugging tool for knowledge bases because it reveals places where knowledge is missing. In this particular case, we can fix the knowledge base by asserting that, at each time step, the agent is in exactly one location, using a collection of sentences similar to those used to assert the existence of exactly one wumpus. Alternatively, we can assert $\neg L^0_{x,y}$ for all locations other than $[1, 1]$; the successor-state axiom for location takes care of subsequent time steps. The same fixes also work to make sure the agent has only one orientation.

SATPLAN has more surprises in store, however. The first is that it finds models with impossible actions, such as shooting with no arrow. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (7.3)) say about actions whose preconditions are not satisfied. The axioms *do* predict correctly that nothing will happen when such an action is executed (see Exercise 10.14), but they do *not* say that the action cannot be executed! To avoid generating plans with illegal actions, we must add **precondition axioms** stating that an action occurrence requires the preconditions to be satisfied.[13] For example, we need to say, for each time t, that

PRECONDITION AXIOMS

$$Shoot^t \Rightarrow HaveArrow^t .$$

This ensures that if a plan selects the *Shoot* action at any time, it must be the case that the agent has an arrow at that time.

SATPLAN's second surprise is the creation of plans with multiple simultaneous actions. For example, it may come up with a model in which both $Forward^0$ and $Shoot^0$ are true, which is not allowed. To eliminate this problem, we introduce **action exclusion axioms**: for every pair of actions A^t_i and A^t_j we add the axiom

ACTION EXCLUSION AXIOM

$$\neg A^t_i \vee \neg A^t_j .$$

It might be pointed out that walking forward and shooting at the same time is not so hard to do, whereas, say, shooting and grabbing at the same time is rather impractical. By imposing action exclusion axioms only on pairs of actions that really do interfere with each other, we can allow for plans that include multiple simultaneous actions—and because SATPLAN finds the shortest legal plan, we can be sure that it will take advantage of this capability.

To summarize, SATPLAN finds models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and the action exclusion axioms. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious "solutions." Any model satisfying the propositional sentence will be a valid plan for the original problem. Modern SAT-solving technology makes the approach quite practical. For example, a DPLL-style solver has no difficulty in generating the 11-step solution for the wumpus world instance shown in Figure 7.2.

This section has described a declarative approach to agent construction: the agent works by a combination of asserting sentences in the knowledge base and performing logical inference. This approach has some weaknesses hidden in phrases such as "for each time t" and

[13] Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.

"for each square $[x, y]$." For any practical agent, these phrases have to be implemented by code that generates instances of the general sentence schema automatically for insertion into the knowledge base. For a wumpus world of reasonable size—one comparable to a smallish computer game—we might need a 100×100 board and 1000 time steps, leading to knowledge bases with tens or hundreds of millions of sentences. Not only does this become rather impractical, but it also illustrates a deeper problem: we know something about the wumpus world—namely, that the "physics" works the same way across all squares and all time steps—that we cannot express directly in the language of propositional logic. To solve this problem, we need a more expressive language, one in which phrases like "for each time t" and "for each square $[x, y]$" can be written in a natural way. First-order logic, described in Chapter 8, is such a language; in first-order logic a wumpus world of any size and duration can be described in about ten sentences rather than ten million or ten trillion.

7.8 SUMMARY

We have introduced knowledge-based agents and have shown how to define a logic with which such agents can reason about the world. The main points are as follows:

- Intelligent agents need knowledge about the world in order to reach good decisions.
- Knowledge is contained in agents in the form of **sentences** in a **knowledge representation language** that are stored in a **knowledge base**.
- A knowledge-based agent is composed of a knowledge base and an inference mechanism. It operates by storing sentences about the world in its knowledge base, using the inference mechanism to infer new sentences, and using these sentences to decide what action to take.
- A representation language is defined by its **syntax**, which specifies the structure of sentences, and its **semantics**, which defines the **truth** of each sentence in each **possible world** or **model**.
- The relationship of **entailment** between sentences is crucial to our understanding of reasoning. A sentence α entails another sentence β if β is true in all worlds where α is true. Equivalent definitions include the **validity** of the sentence $\alpha \Rightarrow \beta$ and the **unsatisfiability** of the sentence $\alpha \wedge \neg\beta$.
- Inference is the process of deriving new sentences from old ones. **Sound** inference algorithms derive *only* sentences that are entailed; **complete** algorithms derive *all* sentences that are entailed.
- **Propositional logic** is a simple language consisting of **proposition symbols** and **logical connectives**. It can handle propositions that are known true, known false, or completely unknown.
- The set of possible models, given a fixed propositional vocabulary, is finite, so entailment can be checked by enumerating models. Efficient **model-checking** inference algorithms for propositional logic include backtracking and local search methods and can often solve large problems quickly.

- **Inference rules** are patterns of sound inference that can be used to find proofs. The **resolution** rule yields a complete inference algorithm for knowledge bases that are expressed in **conjunctive normal form**. **Forward chaining** and **backward chaining** are very natural reasoning algorithms for knowledge bases in **Horn form**.
- **Local search** methods such as WALKSAT can be used to find solutions. Such algorithms are sound but not complete.
- Logical **state estimation** involves maintaining a logical sentence that describes the set of possible states consistent with the observation history. Each update step requires inference using the transition model of the environment, which is built from **successor-state axioms** that specify how each **fluent** changes.
- Decisions within a logical agent can be made by SAT solving: finding possible models specifying future action sequences that reach the goal. This approach works only for fully observable or sensorless environments.
- Propositional logic does not scale to environments of unbounded size because it lacks the expressive power to deal concisely with time, space, and universal patterns of relationships among objects.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

John McCarthy's paper "Programs with Common Sense" (McCarthy, 1958, 1968) promulgated the notion of agents that use logical reasoning to mediate between percepts and actions. It also raised the flag of declarativism, pointing out that telling an agent what it needs to know is an elegant way to build software. Allen Newell's (1982) article "The Knowledge Level" makes the case that rational agents can be described and analyzed at an abstract level defined by the knowledge they possess rather than the programs they run. The declarative and procedural approaches to AI are analyzed in depth by Boden (1977). The debate was revived by, among others, Brooks (1991) and Nilsson (1991), and continues to this day (Shaparau *et al.*, 2008). Meanwhile, the declarative approach has spread into other areas of computer science such as networking (Loo *et al.*, 2006).

Logic itself had its origins in ancient Greek philosophy and mathematics. Various logical principles—principles connecting the syntactic structure of sentences with their truth and falsity, with their meaning, or with the validity of arguments in which they figure—are scattered in the works of Plato. The first known systematic study of logic was carried out by Aristotle, whose work was assembled by his students after his death in 322 B.C. as a treatise called the *Organon*. Aristotle's **syllogisms** were what we would now call inference rules. Although the syllogisms included elements of both propositional and first-order logic, the system as a whole lacked the compositional properties required to handle sentences of arbitrary complexity.

SYLLOGISM

The closely related Megarian and Stoic schools (originating in the fifth century B.C. and continuing for several centuries thereafter) began the systematic study of the basic logical connectives. The use of truth tables for defining connectives is due to Philo of Megara. The

Stoics took five basic inference rules as valid without proof, including the rule we now call Modus Ponens. They derived a number of other rules from these five, using, among other principles, the deduction theorem (page 249) and were much clearer about the notion of proof than was Aristotle. A good account of the history of Megarian and Stoic logic is given by Benson Mates (1953).

The idea of reducing logical inference to a purely mechanical process applied to a formal language is due to Wilhelm Leibniz (1646–1716), although he had limited success in implementing the ideas. George Boole (1847) introduced the first comprehensive and workable system of formal logic in his book *The Mathematical Analysis of Logic*. Boole's logic was closely modeled on the ordinary algebra of real numbers and used substitution of logically equivalent expressions as its primary inference method. Although Boole's system still fell short of full propositional logic, it was close enough that other mathematicians could quickly fill in the gaps. Schröder (1877) described conjunctive normal form, while Horn form was introduced much later by Alfred Horn (1951). The first comprehensive exposition of modern propositional logic (and first-order logic) is found in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation").

The first mechanical device to carry out logical inferences was constructed by the third Earl of Stanhope (1753–1816). The Stanhope Demonstrator could handle syllogisms and certain inferences in the theory of probability. William Stanley Jevons, one of those who improved upon and extended Boole's work, constructed his "logical piano" in 1869 to perform inferences in Boolean logic. An entertaining and instructive history of these and other early mechanical devices for reasoning is given by Martin Gardner (1968). The first published computer program for logical inference was the Logic Theorist of Newell, Shaw, and Simon (1957). This program was intended to model human thought processes. Martin Davis (1957) had actually designed a program that came up with a proof in 1954, but the Logic Theorist's results were published slightly earlier.

Truth tables as a method of testing validity or unsatisfiability in propositional logic were introduced independently by Emil Post (1921) and Ludwig Wittgenstein (1922). In the 1930s, a great deal of progress was made on inference methods for first-order logic. In particular, Gödel (1930) showed that a complete procedure for inference in first-order logic could be obtained via a reduction to propositional logic, using Herbrand's theorem (Herbrand, 1930). We take up this history again in Chapter 9; the important point here is that the development of efficient propositional algorithms in the 1960s was motivated largely by the interest of mathematicians in an effective theorem prover for first-order logic. The Davis–Putnam algorithm (Davis and Putnam, 1960) was the first effective algorithm for propositional resolution but was in most cases much less efficient than the DPLL backtracking algorithm introduced two years later (1962). The full resolution rule and a proof of its completeness appeared in a seminal paper by J. A. Robinson (1965), which also showed how to do first-order reasoning without resort to propositional techniques.

Stephen Cook (1971) showed that deciding satisfiability of a sentence in propositional logic (the SAT problem) is NP-complete. Since deciding entailment is equivalent to deciding unsatisfiability, it is co-NP-complete. Many subsets of propositional logic are known for which the satisfiability problem is polynomially solvable; Horn clauses are one such subset.

The linear-time forward-chaining algorithm for Horn clauses is due to Dowling and Gallier (1984), who describe their algorithm as a dataflow process similar to the propagation of signals in a circuit.

Early theoretical investigations showed that DPLL has polynomial average-case complexity for certain natural distributions of problems. This potentially exciting fact became less exciting when Franco and Paull (1983) showed that the same problems could be solved in constant time simply by guessing random assignments. The random-generation method described in the chapter produces much harder problems. Motivated by the empirical success of local search on these problems, Koutsoupias and Papadimitriou (1992) showed that a simple hill-climbing algorithm can solve *almost all* satisfiability problem instances very quickly, suggesting that hard problems are rare. Moreover, Schöning (1999) exhibited a randomized hill-climbing algorithm whose *worst-case* expected run time on 3-SAT problems (that is, satisfiability of 3-CNF sentences) is $O(1.333^n)$—still exponential, but substantially faster than previous worst-case bounds. The current record is $O(1.324^n)$ (Iwama and Tamaki, 2004). Achlioptas *et al.* (2004) and Alekhnovich *et al.* (2005) exhibit families of 3-SAT instances for which all known DPLL-like algorithms require exponential running time.

On the practical side, efficiency gains in propositional solvers have been marked. Given ten minutes of computing time, the original DPLL algorithm in 1962 could only solve problems with no more than 10 or 15 variables. By 1995 the SATZ solver (Li and Anbulagan, 1997) could handle 1,000 variables, thanks to optimized data structures for indexing variables. Two crucial contributions were the **watched literal** indexing technique of Zhang and Stickel (1996), which makes unit propagation very efficient, and the introduction of clause (i.e., constraint) learning techniques from the CSP community by Bayardo and Schrag (1997). Using these ideas, and spurred by the prospect of solving industrial-scale circuit verification problems, Moskewicz *et al.* (2001) developed the CHAFF solver, which could handle problems with millions of variables. Beginning in 2002, SAT competitions have been held regularly; most of the winning entries have either been descendants of CHAFF or have used the same general approach. RSAT (Pipatsrisawat and Darwiche, 2007), the 2007 winner, falls in the latter category. Also noteworthy is MINISAT (Een and Sörensson, 2003), an open-source implementation available at `http://minisat.se` that is designed to be easily modified and improved. The current landscape of solvers is surveyed by Gomes *et al.* (2008).

Local search algorithms for satisfiability were tried by various authors throughout the 1980s; all of the algorithms were based on the idea of minimizing the number of unsatisfied clauses (Hansen and Jaumard, 1990). A particularly effective algorithm was developed by Gu (1989) and independently by Selman *et al.* (1992), who called it GSAT and showed that it was capable of solving a wide range of very hard problems very quickly. The WALKSAT algorithm described in the chapter is due to Selman *et al.* (1996).

The "phase transition" in satisfiability of random k-SAT problems was first observed by Simon and Dubois (1989) and has given rise to a great deal of theoretical and empirical research—due, in part, to the obvious connection to phase transition phenomena in statistical physics. Cheeseman *et al.* (1991) observed phase transitions in several CSPs and conjecture that all NP-hard problems have a phase transition. Crawford and Auton (1993) located the 3-SAT transition at a clause/variable ratio of around 4.26, noting that this coincides with a

sharp peak in the run time of their SAT solver. Cook and Mitchell (1997) provide an excellent summary of the early literature on the problem.

SATISFIABILITY THRESHOLD CONJECTURE

The current state of theoretical understanding is summarized by Achlioptas (2009). The **satisfiability threshold conjecture** states that, for each k, there is a sharp satisfiability threshold r_k, such that as the number of variables $n \to \infty$, instances below the threshold are *satisfiable* with probability 1, while those above the threshold are *unsatisfiable* with probability 1. The conjecture was not quite proved by Friedgut (1999): a sharp threshold exists but its location might depend on n even as $n \to \infty$. Despite significant progress in asymptotic analysis of the threshold location for large k (Achlioptas and Peres, 2004; Achlioptas *et al.*, 2007), all that can be proved for $k = 3$ is that it lies in the range [3.52,4.51]. Current theory suggests that a peak in the run time of a SAT solver is not necessarily related to the satisfiability threshold, but instead to a phase transition in the solution distribution and structure of SAT instances. Empirical results due to Coarfa *et al.* (2003) support this view. In fact, algorithms such as **survey propagation** (Parisi and Zecchina, 2002; Maneva *et al.*, 2007) take advantage of special properties of random SAT instances near the satisfiability threshold and greatly outperform general SAT solvers on such instances.

SURVEY PROPAGATION

The best sources for information on satisfiability, both theoretical and practical, are the *Handbook of Satisfiability* (Biere *et al.*, 2009) and the regular *International Conferences on Theory and Applications of Satisfiability Testing*, known as SAT.

The idea of building agents with propositional logic can be traced back to the seminal paper of McCulloch and Pitts (1943), which initiated the field of neural networks. Contrary to popular supposition, the paper was concerned with the implementation of a Boolean circuit-based agent design in the brain. Circuit-based agents, which perform computation by propagating signals in hardware circuits rather than running algorithms in general-purpose computers, have received little attention in AI, however. The most notable exception is the work of Stan Rosenschein (Rosenschein, 1985; Kaelbling and Rosenschein, 1990), who developed ways to compile circuit-based agents from declarative descriptions of the task environment. (Rosenschein's approach is described at some length in the second edition of this book.) The work of Rod Brooks (1986, 1989) demonstrates the effectiveness of circuit-based designs for controlling robots—a topic we take up in Chapter 25. Brooks (1991) argues that circuit-based designs are *all* that is needed for AI—that representation and reasoning are cumbersome, expensive, and unnecessary. In our view, neither approach is sufficient by itself. Williams *et al.* (2003) show how a hybrid agent design not too different from our wumpus agent has been used to control NASA spacecraft, planning sequences of actions and diagnosing and recovering from faults.

The general problem of keeping track of a partially observable environment was introduced for state-based representations in Chapter 4. Its instantiation for propositional representations was studied by Amir and Russell (2003), who identified several classes of environments that admit efficient state-estimation algorithms and showed that for several other classes the problem is intractable. The **temporal-projection** problem, which involves determining what propositions hold true after an action sequence is executed, can be seen as a special case of state estimation with empty percepts. Many authors have studied this problem because of its importance in planning; some important hardness results were established by

TEMPORAL-PROJECTION

Liberatore (1997). The idea of representing a belief state with propositions can be traced to Wittgenstein (1922).

Logical state estimation, of course, requires a logical representation of the effects of actions—a key problem in AI since the late 1950s. The dominant proposal has been the **situation calculus** formalism (McCarthy, 1963), which is couched within first-order logic. We discuss situation calculus, and various extensions and alternatives, in Chapters 10 and 12. The approach taken in this chapter—using temporal indices on propositional variables—is more restrictive but has the benefit of simplicity. The general approach embodied in the SATPLAN algorithm was proposed by Kautz and Selman (1992). Later generations of SATPLAN were able to take advantage of the advances in SAT solvers, described earlier, and remain among the most effective ways of solving difficult problems (Kautz, 2006).

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem unsolvable within first-order logic, and it spurred a great deal of research into nonmonotonic logics. Philosophers from Dreyfus (1972) to Crockett (1994) have cited the frame problem as one symptom of the inevitable failure of the entire AI enterprise. The solution of the frame problem with successor-state axioms is due to Ray Reiter (1991). Thielscher (1999) identifies the inferential frame problem as a separate idea and provides a solution. In retrospect, one can see that Rosenschein's (1985) agents were using circuits that implemented successor-state axioms, but Rosenschein did not notice that the frame problem was thereby largely solved. Foo (2001) explains why the discrete-event control theory models typically used by engineers do not have to explicitly deal with the frame problem: because they are dealing with prediction and control, not with explanation and reasoning about counterfactual situations.

Modern propositional solvers have wide applicability in industrial applications. The application of propositional inference in the synthesis of computer hardware is now a standard technique having many large-scale deployments (Nowick *et al.*, 1993). The SATMC satisfiability checker was used to detect a previously unknown vulnerability in a Web browser user sign-on protocol (Armando *et al.*, 2008).

The wumpus world was invented by Gregory Yob (1975). Ironically, Yob developed it because he was bored with games played on a rectangular grid: the topology of his original wumpus world was a dodecahedron, and we put it back in the boring old grid. Michael Genesereth was the first to suggest that the wumpus world be used as an agent testbed.

EXERCISES

7.1 Suppose the agent has progressed to the point shown in Figure 7.4(a), page 239, having perceived nothing in [1,1], a breeze in [2,1], and a stench in [1,2], and is now concerned with the contents of [1,3], [2,2], and [3,1]. Each of these can contain a pit, and at most one can contain a wumpus. Following the example of Figure 7.5, construct the set of possible worlds. (You should find 32 of them.) Mark the worlds in which the KB is true and those in which

each of the following sentences is true:

α_2 = "There is no pit in [2,2]."
α_3 = "There is a wumpus in [1,3]."

Hence show that $KB \models \alpha_2$ and $KB \models \alpha_3$.

7.2 (Adapted from Barwise and Etchemendy (1993).) Given the following, can you prove that the unicorn is mythical? How about magical? Horned?

> If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.

7.3 Consider the problem of deciding whether a propositional logic sentence is true in a given model.

a. Write a recursive algorithm PL-TRUE?(s, m) that returns $true$ if and only if the sentence s is true in the model m (where m assigns a truth value for every symbol in s). The algorithm should run in time linear in the size of the sentence. (Alternatively, use a version of this function from the online code repository.)

b. Give three examples of sentences that can be determined to be true or false in a *partial* model that does not specify a truth value for some of the symbols.

c. Show that the truth value (if any) of a sentence in a partial model cannot be determined efficiently in general.

d. Modify your PL-TRUE? algorithm so that it can sometimes judge truth from partial models, while retaining its recursive structure and linear run time. Give three examples of sentences whose truth in a partial model is *not* detected by your algorithm.

e. Investigate whether the modified algorithm makes TT-ENTAILS? more efficient.

7.4 Which of the following are correct?

a. $False \models True$.
b. $True \models False$.
c. $(A \wedge B) \models (A \Leftrightarrow B)$.
d. $A \Leftrightarrow B \models A \vee B$.
e. $A \Leftrightarrow B \models \neg A \vee B$.
f. $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$.
g. $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$.
h. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B)$.
i. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$.
j. $(A \vee B) \wedge \neg(A \Rightarrow B)$ is satisfiable.
k. $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ is satisfiable.
l. $(A \Leftrightarrow B) \Leftrightarrow C$ has the same number of models as $(A \Leftrightarrow B)$ for any fixed set of proposition symbols that includes A, B, C.

7.5 Prove each of the following assertions:

a. α is valid if and only if $True \models \alpha$.
b. For any α, $False \models \alpha$.
c. $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.
d. $\alpha \equiv \beta$ if and only if the sentence $(\alpha \Leftrightarrow \beta)$ is valid.
e. $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

7.6 Prove, or find a counterexample to, each of the following assertions:

a. If $\alpha \models \gamma$ or $\beta \models \gamma$ (or both) then $(\alpha \wedge \beta) \models \gamma$
b. If $\alpha \models (\beta \wedge \gamma)$ then $\alpha \models \beta$ and $\alpha \models \gamma$.
c. If $\alpha \models (\beta \vee \gamma)$ then $\alpha \models \beta$ or $\alpha \models \gamma$ (or both).

7.7 Consider a vocabulary with only four propositions, A, B, C, and D. How many models are there for the following sentences?

a. $B \vee C$.
b. $\neg A \vee \neg B \vee \neg C \vee \neg D$.
c. $(A \Rightarrow B) \wedge A \wedge \neg B \wedge C \wedge D$.

7.8 We have defined four binary logical connectives.

a. Are there any others that might be useful?
b. How many binary connectives can there be?
c. Why are some of them not very useful?

7.9 Using a method of your choice, verify each of the equivalences in Figure 7.11 (page 249).

7.10 Decide whether each of the following sentences is valid, unsatisfiable, or neither. Verify your decisions using truth tables or the equivalence rules of Figure 7.11 (page 249).

a. $Smoke \Rightarrow Smoke$
b. $Smoke \Rightarrow Fire$
c. $(Smoke \Rightarrow Fire) \Rightarrow (\neg Smoke \Rightarrow \neg Fire)$
d. $Smoke \vee Fire \vee \neg Fire$
e. $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))$
f. $(Smoke \Rightarrow Fire) \Rightarrow ((Smoke \wedge Heat) \Rightarrow Fire)$
g. $Big \vee Dumb \vee (Big \Rightarrow Dumb)$

7.11 Any propositional logic sentence is logically equivalent to the assertion that each possible world in which it would be false is not the case. From this observation, prove that any sentence can be written in CNF.

7.12 Use resolution to prove the sentence $\neg A \wedge \neg B$ from the clauses in Exercise 7.20.

7.13 This exercise looks into the relationship between clauses and implication sentences.

a. Show that the clause $(\neg P_1 \vee \cdots \vee \neg P_m \vee Q)$ is logically equivalent to the implication sentence $(P_1 \wedge \cdots \wedge P_m) \Rightarrow Q$.

b. Show that every clause (regardless of the number of positive literals) can be written in the form $(P_1 \wedge \cdots \wedge P_m) \Rightarrow (Q_1 \vee \cdots \vee Q_n)$, where the Ps and Qs are proposition symbols. A knowledge base consisting of such sentences is in **implicative normal form** or **Kowalski form** (Kowalski, 1979).

IMPLICATIVE NORMAL FORM

c. Write down the full resolution rule for sentences in implicative normal form.

7.14 According to some political pundits, a person who is radical (R) is electable (E) if he/she is conservative (C), but otherwise is not electable.

a. Which of the following are correct representations of this assertion?

(i) $(R \wedge E) \iff C$

(ii) $R \Rightarrow (E \iff C)$

(iii) $R \Rightarrow ((C \Rightarrow E) \vee \neg E)$

b. Which of the sentences in (a) can be expressed in Horn form?

7.15 This question considers representing satisfiability (SAT) problems as CSPs.

a. Draw the constraint graph corresponding to the SAT problem

$$(\neg X_1 \vee X_2) \wedge (\neg X_2 \vee X_3) \wedge \ldots \wedge (\neg X_{n-1} \vee X_n)$$

for the particular case $n = 5$.

b. How many solutions are there for this general SAT problem as a function of n?

c. Suppose we apply BACKTRACKING-SEARCH (page 215) to find *all* solutions to a SAT CSP of the type given in (a). (To find *all* solutions to a CSP, we simply modify the basic algorithm so it continues searching after each solution is found.) Assume that variables are ordered $X_1, \ldots, X_n$ and $false$ is ordered before $true$. How much time will the algorithm take to terminate? (Write an $O(\cdot)$ expression as a function of n.)

d. We know that SAT problems in Horn form can be solved in linear time by forward chaining (unit propagation). We also know that every tree-structured binary CSP with discrete, finite domains can be solved in time linear in the number of variables (Section 6.5). Are these two facts connected? Discuss.

7.16 Explain why every nonempty propositional clause, by itself, is satisfiable. Prove rigorously that every set of five 3-SAT clauses is satisfiable, provided that each clause mentions exactly three distinct variables. What is the smallest set of such clauses that is unsatisfiable? Construct such a set.

7.17 A propositional *2-CNF* expression is a conjunction of clauses, each containing *exactly 2* literals, e.g.,

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee D) \wedge (\neg C \vee G) \wedge (\neg D \vee G) \ .$$

a. Prove using resolution that the above sentence entails G.

b. Two clauses are *semantically distinct* if they are not logically equivalent. How many semantically distinct 2-CNF clauses can be constructed from n proposition symbols?

c. Using your answer to (b), prove that propositional resolution always terminates in time polynomial in n given a 2-CNF sentence containing no more than n distinct symbols.

d. Explain why your argument in (c) does not apply to 3-CNF.

7.18 Consider the following sentence:

$$[(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)] \Rightarrow [(Food \wedge Drinks) \Rightarrow Party] .$$

a. Determine, using enumeration, whether this sentence is valid, satisfiable (but not valid), or unsatisfiable.

b. Convert the left-hand and right-hand sides of the main implication into CNF, showing each step, and explain how the results confirm your answer to (a).

c. Prove your answer to (a) using resolution.

DISJUNCTIVE NORMAL FORM

7.19 A sentence is in **disjunctive normal form** (DNF) if it is the disjunction of conjunctions of literals. For example, the sentence $(A \wedge B \wedge \neg C) \vee (\neg A \wedge C) \vee (B \wedge \neg C)$ is in DNF.

a. Any propositional logic sentence is logically equivalent to the assertion that some possible world in which it would be true is in fact the case. From this observation, prove that any sentence can be written in DNF.

b. Construct an algorithm that converts any sentence in propositional logic into DNF. (*Hint*: The algorithm is similar to the algorithm for conversion to CNF given in Section 7.5.2.)

c. Construct a simple algorithm that takes as input a sentence in DNF and returns a satisfying assignment if one exists, or reports that no satisfying assignment exists.

d. Apply the algorithms in (b) and (c) to the following set of sentences:

$$A \Rightarrow B$$
$$B \Rightarrow C$$
$$C \Rightarrow \neg A .$$

e. Since the algorithm in (b) is very similar to the algorithm for conversion to CNF, and since the algorithm in (c) is much simpler than any algorithm for solving a set of sentences in CNF, why is this technique not used in automated reasoning?

7.20 Convert the following set of sentences to clausal form.

S1: $A \Leftrightarrow (B \vee E)$.
S2: $E \Rightarrow D$.
S3: $C \wedge F \Rightarrow \neg B$.
S4: $E \Rightarrow B$.
S5: $B \Rightarrow F$.
S6: $B \Rightarrow C$

Give a trace of the execution of DPLL on the conjunction of these clauses.

7.21 Is a randomly generated 4-CNF sentence with n symbols and m clauses more or less likely to be solvable than a randomly generated 3-CNF sentence with n symbols and m clauses? Explain.

7.22 Minesweeper, the well-known computer game, is closely related to the wumpus world. A minesweeper world is a rectangular grid of N squares with M invisible mines scattered among them. Any square may be probed by the agent; instant death follows if a mine is probed. Minesweeper indicates the presence of mines by revealing, in each probed square, the *number* of mines that are directly or diagonally adjacent. The goal is to probe every unmined square.

a. Let $X_{i,j}$ be true iff square $[i, j]$ contains a mine. Write down the assertion that exactly two mines are adjacent to [1,1] as a sentence involving some logical combination of $X_{i,j}$ propositions.

b. Generalize your assertion from (a) by explaining how to construct a CNF sentence asserting that k of n neighbors contain mines.

c. Explain precisely how an agent can use DPLL to prove that a given square does (or does not) contain a mine, ignoring the global constraint that there are exactly M mines in all.

d. Suppose that the global constraint is constructed from your method from part (b). How does the number of clauses depend on M and N? Suggest a way to modify DPLL so that the global constraint does not need to be represented explicitly.

e. Are any conclusions derived by the method in part (c) invalidated when the global constraint is taken into account?

f. Give examples of configurations of probe values that induce *long-range dependencies* such that the contents of a given unprobed square would give information about the contents of a far-distant square. (*Hint*: consider an $N \times 1$ board.)

7.23 How long does it take to prove $KB \models \alpha$ using DPLL when α is a literal *already contained in* KB? Explain.

7.24 Trace the behavior of DPLL on the knowledge base in Figure 7.16 when trying to prove Q, and compare this behavior with that of the forward-chaining algorithm.

7.25 Write a successor-state axiom for the $Locked$ predicate, which applies to doors, assuming the only actions available are $Lock$ and $Unlock$.

7.26 Section 7.7.1 provides some of the successor-state axioms required for the wumpus world. Write down axioms for all remaining fluent symbols.

7.27 Modify the HYBRID-WUMPUS-AGENT to use the 1-CNF logical state estimation method described on page 271. We noted on that page that such an agent will not be able to acquire, maintain, and use more complex beliefs such as the disjunction $P_{3,1} \vee P_{2,2}$. Suggest a method for overcoming this problem by defining additional proposition symbols, and try it out in the wumpus world. Does it improve the performance of the agent?

8 FIRST-ORDER LOGIC

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

In Chapter 7, we showed how a knowledge-based agent could represent the world in which it operates and deduce what actions to take. We used propositional logic as our representation language because it sufficed to illustrate the basic concepts of logic and knowledge-based agents. Unfortunately, propositional logic is too puny a language to represent knowledge of complex environments in a concise way. In this chapter, we examine **first-order logic**,[1] which is sufficiently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages and has been studied intensively for many decades. We begin in Section 8.1 with a discussion of representation languages in general; Section 8.2 covers the syntax and semantics of first-order logic; Sections 8.3 and 8.4 illustrate the use of first-order logic for simple representations.

FIRST-ORDER LOGIC

8.1 REPRESENTATION REVISITED

In this section, we 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

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

The famous **Sapir–Whorf hypothesis** 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 ...
- Relations: these can be unary relations or **properties** such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...
- Functions: father of, best friend, third inning of, one more than, beginning of ...

PROPERTY

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

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

TEMPORAL LOGIC

make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) "first class" status within the logic, rather than simply defining them within the knowledge base.

HIGHER-ORDER LOGIC

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, ⟨the crown, King John⟩. The "brother" and "on head" relations are binary relations—that is, they relate pairs of objects. The model also contains unary relations, or properties: the "person" property is true of both Richard and John; the "king" property is true only of John (presumably because Richard is dead at this point); and the "crown" property is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

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

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"

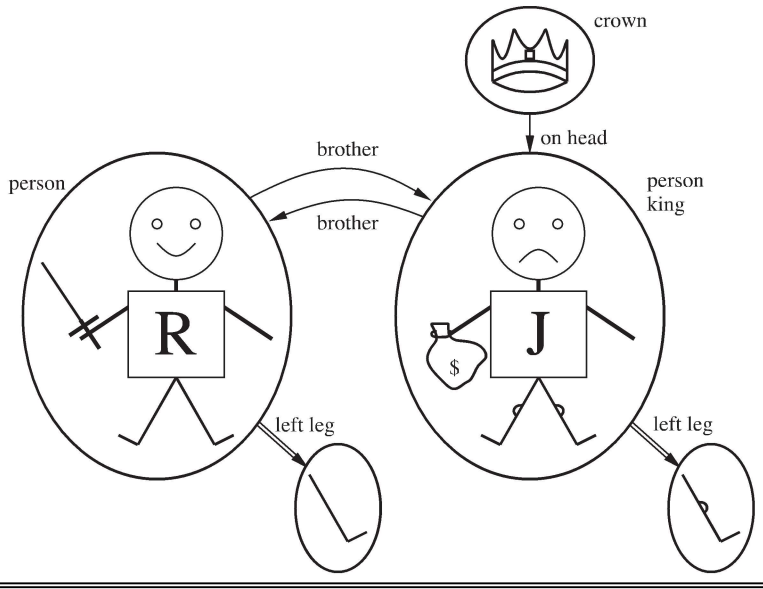


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

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.

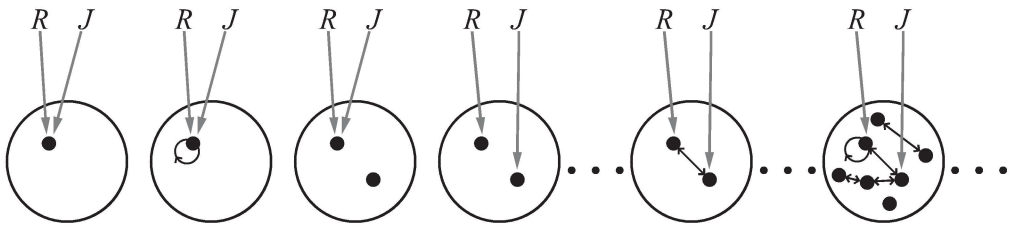


Figure 8.4 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

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.

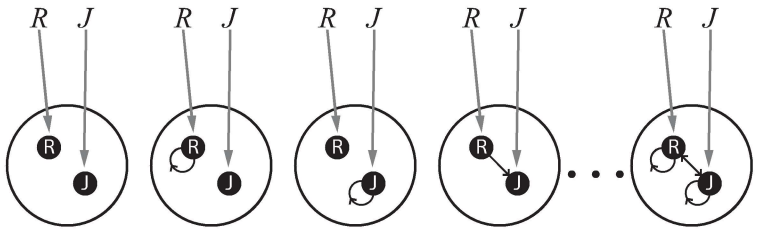


Figure 8.5 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

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

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:

ASSERTION

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

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

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

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

QUERY
GOAL

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

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

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

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.

SUBSTITUTION
BINDING LIST

8.3.2 The kinship domain

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

Clearly, the objects in our domain are people. 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

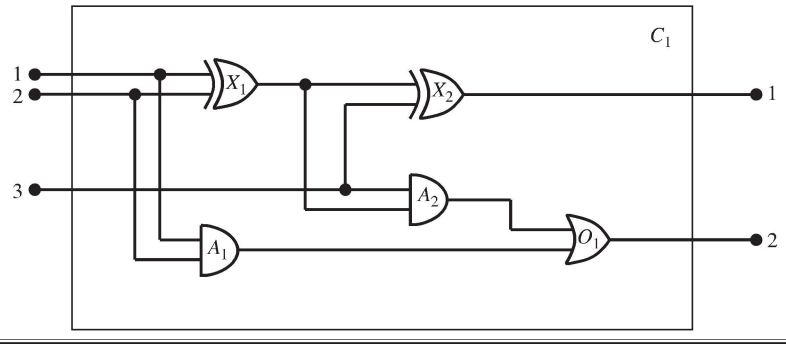


Figure 8.6 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one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

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

The answers are substitutions for the variables i_1, i_2, and i_3 such that the resulting sentence is entailed by the knowledge base. 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?

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

This final query will return a complete input–output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification**. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out. (See Exercise 8.26.) 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 $1 \neq 0$.

8.5 SUMMARY

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

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

BIBLIOGRAPHICAL AND HISTORICAL NOTES

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

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

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

McCarthy (1958) was primarily responsible for the introduction of first-order logic as a tool for building AI systems. The prospects for logic-based AI were advanced significantly by Robinson's (1965) development of resolution, a complete procedure for first-order inference described in Chapter 9. The logicist approach took root at Stanford University. Cordell Green (1969a, 1969b) developed a first-order reasoning system, QA3, leading to the first attempts to build a logical robot at SRI (Fikes and Nilsson, 1971). First-order logic was applied by Zohar Manna and Richard Waldinger (1971) for reasoning about programs and later by Michael Genesereth (1984) for reasoning about circuits. In Europe, logic programming (a restricted form of first-order reasoning) was developed for linguistic analysis (Colmerauer *et al.*, 1973) and for general declarative systems (Kowalski, 1974). Computational logic was also well entrenched at Edinburgh through the LCF (Logic for Computable Functions) project (Gordon *et al.*, 1979). These developments are chronicled further in Chapters 9 and 12.

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

Reactions to the Whorf hypothesis (Whorf, 1956) and the problem of language and thought in general, appear in several recent books (Gumperz and Levinson, 1996; Bowerman and Levinson, 2001; Pinker, 2003; Gentner and Goldin-Meadow, 2003). The "theory" theory (Gopnik and Glymour, 2002; Tenenbaum *et al.*, 2007) views children's learning about the world as analogous to the construction of scientific theories. Just as the predictions of a machine learning algorithm depend strongly on the vocabulary supplied to it, so will the child's formulation of theories depend on the linguistic environment in which learning occurs.

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

EXERCISES

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

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

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

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

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

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

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

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

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

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

8.6 Which of the following are valid (necessarily true) sentences?

a. $(\exists x\ x = x)\ \Rightarrow\ (\forall\, y\ \ \exists z\ y = z)$.

b. $\forall\, x\ \ P(x) \vee \neg P(x)$.

c. $\forall\, x\ \ \mathit{Smart}(x) \vee (x = x)$.

8.7 Consider a version of the semantics for first-order logic in which models with empty domains are allowed. Give at least two examples of sentences that are valid according to the

standard semantics but not according to the new semantics. Discuss which outcome makes more intuitive sense for your examples.

8.8 Does the fact $\neg Spouse(George, Laura)$ follow from the facts $Jim \neq George$ and $Spouse(Jim, Laura)$? If so, give a proof; if not, supply additional axioms as needed. What happens if we use $Spouse$ as a unary function symbol instead of a binary predicate?

8.9 This exercise uses the function $MapColor$ and predicates $In(x, y)$, $Borders(x, y)$, and $Country(x)$, whose arguments are geographical regions, along with constant symbols for various regions. In each of the following we give an English sentence and a number of candidate logical expressions. For each of the logical expressions, state whether it (1) correctly expresses the English sentence; (2) is syntactically invalid and therefore meaningless; or (3) is syntactically valid but does not express the meaning of the English sentence.

a. Paris and Marseilles are both in France.

(i) $In(Paris \wedge Marseilles, France)$.
(ii) $In(Paris, France) \wedge In(Marseilles, France)$.
(iii) $In(Paris, France) \vee In(Marseilles, France)$.

b. There is a country that borders both Iraq and Pakistan.

(i) $\exists c \;\; Country(c) \wedge Border(c, Iraq) \wedge Border(c, Pakistan)$.
(ii) $\exists c \;\; Country(c) \Rightarrow [Border(c, Iraq) \wedge Border(c, Pakistan)]$.
(iii) $[\exists c \;\; Country(c)] \Rightarrow [Border(c, Iraq) \wedge Border(c, Pakistan)]$.
(iv) $\exists c \;\; Border(Country(c), Iraq \wedge Pakistan)$.

c. All countries that border Ecuador are in South America.

(i) $\forall c \;\; Country(c) \wedge Border(c, Ecuador) \Rightarrow In(c, SouthAmerica)$.
(ii) $\forall c \;\; Country(c) \Rightarrow [Border(c, Ecuador) \Rightarrow In(c, SouthAmerica)]$.
(iii) $\forall c \;\; [Country(c) \Rightarrow Border(c, Ecuador)] \Rightarrow In(c, SouthAmerica)$.
(iv) $\forall c \;\; Country(c) \wedge Border(c, Ecuador) \wedge In(c, SouthAmerica)$.

d. No region in South America borders any region in Europe.

(i) $\neg[\exists c, d \;\; In(c, SouthAmerica) \wedge In(d, Europe) \wedge Borders(c, d)]$.
(ii) $\forall c, d \;\; [In(c, SouthAmerica) \wedge In(d, Europe)] \Rightarrow \neg Borders(c, d)]$.
(iii) $\neg\forall c \;\; In(c, SouthAmerica) \Rightarrow \exists d \;\; In(d, Europe) \wedge \neg Borders(c, d)$.
(iv) $\forall c \;\; In(c, SouthAmerica) \Rightarrow \forall d \;\; In(d, Europe) \Rightarrow \neg Borders(c, d)$.

e. No two adjacent countries have the same map color.

(i) $\forall x, y \;\; \neg Country(x) \vee \neg Country(y) \vee \neg Borders(x, y) \vee$
$\neg(MapColor(x) = MapColor(y))$.
(ii) $\forall x, y \;\; (Country(x) \wedge Country(y) \wedge Borders(x, y) \wedge \neg(x = y)) \Rightarrow$
$\neg(MapColor(x) = MapColor(y))$.
(iii) $\forall x, y \;\; Country(x) \wedge Country(y) \wedge Borders(x, y) \wedge$
$\neg(MapColor(x) = MapColor(y))$.
(iv) $\forall x, y \;\; (Country(x) \wedge Country(y) \wedge Borders(x, y)) \Rightarrow MapColor(x \neq y)$.

8.10 Consider a vocabulary with the following symbols:

$Occupation(p, o)$: Predicate. Person p has occupation o.
$Customer(p1, p2)$: Predicate. Person $p1$ is a customer of person $p2$.
$Boss(p1, p2)$: Predicate. Person $p1$ is a boss of person $p2$.
$Doctor$, $Surgeon$, $Lawyer$, $Actor$: Constants denoting occupations.
$Emily$, Joe: Constants denoting people.

Use these symbols to write the following assertions in first-order logic:

a. Emily is either a surgeon or a lawyer.
b. Joe is an actor, but he also holds another job.
c. All surgeons are doctors.
d. Joe does not have a lawyer (i.e., is not a customer of any lawyer).
e. Emily has a boss who is a lawyer.
f. There exists a lawyer all of whose customers are doctors.
g. Every surgeon has a lawyer.

8.11 Complete the following exercises about logical senntences:

a. Translate into *good, natural* English (no xs or ys!):

$$\forall x, y, l \;\; SpeaksLanguage(x, l) \wedge SpeaksLanguage(y, l) \Rightarrow Understands(x, y) \wedge Understands(y, x).$$

b. Explain why this sentence is entailed by the sentence

$$\forall x, y, l \;\; SpeaksLanguage(x, l) \wedge SpeaksLanguage(y, l) \Rightarrow Understands(x, y).$$

c. Translate into first-order logic the following sentences:

(i) Understanding leads to friendship.
(ii) Friendship is transitive.

Remember to define all predicates, functions, and constants you use.

8.12 Rewrite the first two Peano axioms in Section 8.3.3 as a single axiom that defines $NatNum(x)$ so as to exclude the possibility of natural numbers except for those generated by the successor function.

8.13 Equation (8.4) on page 306 defines the conditions under which a square is breezy. Here we consider two other ways to describe this aspect of the wumpus world.

DIAGNOSTIC RULE

a. We can write **diagnostic rules** leading from observed effects to hidden causes. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit; and if a square is not breezy, then no adjacent square contains a pit. Write these two rules in first-order logic and show that their conjunction is logically equivalent to Equation (8.4).

CAUSAL RULE

b. We can write **causal rules** leading from cause to effect. One obvious causal rule is that a pit causes all adjacent squares to be breezy. Write this rule in first-order logic, explain why it is incomplete compared to Equation (8.4), and supply the missing axiom.

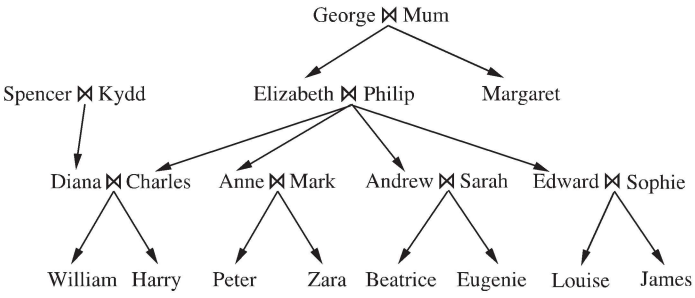


Figure 8.7 A typical family tree. The symbol "⋈" connects spouses and arrows point to children.

8.14 Write axioms describing the predicates *Grandchild*, *Greatgrandparent*, *Ancestor*, *Brother*, *Sister*, *Daughter*, *Son*, *FirstCousin*, *BrotherInLaw*, *SisterInLaw*, *Aunt*, and *Uncle*. Find out the proper definition of mth cousin n times removed, and write the definition in first-order logic. Now write down the basic facts depicted in the family tree in Figure 8.7. Using a suitable logical reasoning system, TELL it all the sentences you have written down, and ASK it who are Elizabeth's grandchildren, Diana's brothers-in-law, Zara's great-grandparents, and Eugenie's ancestors.

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

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

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

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

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

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

8.19 Assuming predicates $Parent(p, q)$ and $Female(p)$ and constants *Joan* and *Kevin*, with the obvious meanings, express each of the following sentences in first-order logic. (You may use the abbreviation $\exists^1$ to mean "there exists exactly one.")

a. Joan has a daughter (possibly more than one, and possibly sons as well).

b. Joan has exactly one daughter (but may have sons as well).

c. Joan has exactly one child, a daughter.

d. Joan and Kevin have exactly one child together.

e. Joan has at least one child with Kevin, and no children with anyone else.

8.20 Arithmetic assertions can be written in first-order logic with the predicate symbol $<$, the function symbols $+$ and $\times$, and the constant symbols 0 and 1. Additional predicates can also be defined with biconditionals.

a. Represent the property "x is an even number."

b. Represent the property "x is prime."

c. Goldbach's conjecture is the conjecture (unproven as yet) that every even number is equal to the sum of two primes. Represent this conjecture as a logical sentence.

8.21 In Chapter 6, we used equality to indicate the relation between a variable and its value. For instance, we wrote $WA = red$ to mean that Western Australia is colored red. Representing this in first-order logic, we must write more verbosely $ColorOf(WA) = red$. What incorrect inference could be drawn if we wrote sentences such as $WA = red$ directly as logical assertions?

8.22 Write in first-order logic the assertion that every key and at least one of every pair of socks will eventually be lost forever, using only the following vocabulary: $Key(x)$, x is a key; $Sock(x)$, x is a sock; $Pair(x, y)$, x and y are a pair; Now, the current time; $Before(t_1, t_2)$, time t_1 comes before time t_2; $Lost(x, t)$, object x is lost at time t.

8.23 For each of the following sentences in English, decide if the accompanying first-order logic sentence is a good translation. If not, explain why not and correct it. (Some sentences may have more than one error!)

a. No two people have the same social security number.

$$\neg\exists\, x, y, n \;\; Person(x) \wedge Person(y) \;\Rightarrow\; [HasSS\#(x, n) \wedge HasSS\#(y, n)].$$

b. John's social security number is the same as Mary's.

$$\exists\, n \;\; HasSS\#(John, n) \wedge HasSS\#(Mary, n).$$

c. Everyone's social security number has nine digits.

$$\forall\, x, n \;\; Person(x) \;\Rightarrow\; [HasSS\#(x, n) \wedge Digits(n, 9)].$$

d. Rewrite each of the above (uncorrected) sentences using a function symbol $SS\#$ instead of the predicate $HasSS\#$.

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

a. Some students took French in spring 2001.

b. Every student who takes French passes it.

c. Only one student took Greek in spring 2001.

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

e. Every person who buys a policy is smart.

f. No person buys an expensive policy.

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

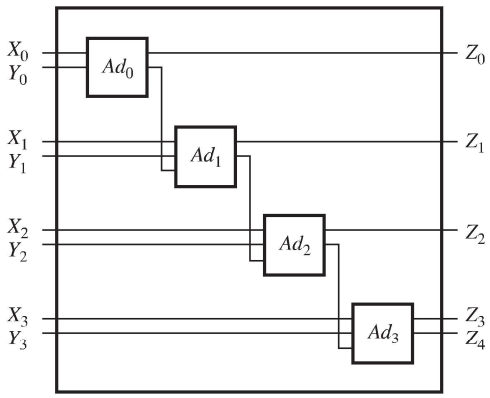


Figure 8.8 A four-bit adder. Each Ad_i is a one-bit adder, as in Figure 8.6 on page 309.

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

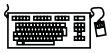
i. A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth.

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

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

l. All Greeks speak the same language. (Use $Speaks(x, l)$ to mean that person x speaks language l.)

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



8.26 Extend the vocabulary from Section 8.4 to define addition for n-bit binary numbers. Then encode the description of the four-bit adder in Figure 8.8, and pose the queries needed to verify that it is in fact correct.

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

8.28 Consider a first-order logical knowledge base that describes worlds containing people, songs, albums (e.g., "Meet the Beatles") and disks (i.e., particular physical instances of CDs). The vocabulary contains the following symbols:

$CopyOf(d, a)$: Predicate. Disk d is a copy of album a.
$Owns(p, d)$: Predicate. Person p owns disk d.
$Sings(p, s, a)$: Album a includes a recording of song s sung by person p.
$Wrote(p, s)$: Person p wrote song s.
$McCartney$, $Gershwin$, $BHoliday$, Joe, $EleanorRigby$, $TheManILove$, $Revolver$: Constants with the obvious meanings.

Express the following statements in first-order logic:

a. Gershwin wrote "The Man I Love."

b. Gershwin did not write "Eleanor Rigby."

c. Either Gershwin or McCartney wrote "The Man I Love."

d. Joe has written at least one song.

e. Joe owns a copy of *Revolver*.

f. Every song that McCartney sings on *Revolver* was written by McCartney.

g. Gershwin did not write any of the songs on *Revolver*.

h. Every song that Gershwin wrote has been recorded on some album. (Possibly different songs are recorded on different albums.)

i. There is a single album that contains every song that Joe has written.

j. Joe owns a copy of an album that has Billie Holiday singing "The Man I Love."

k. Joe owns a copy of every album that has a song sung by McCartney. (Of course, each different album is instantiated in a different physical CD.)

l. Joe owns a copy of every album on which all the songs are sung by Billie Holiday.

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. STORE(s) stores a sentence s into the knowledge base and 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

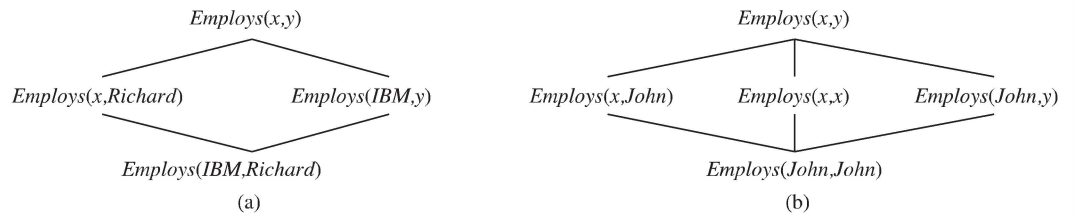


Figure 9.2 (a) The subsumption lattice whose lowest node is $Employs(IBM, Richard)$. (b) The subsumption lattice for the sentence $Employs(John, John)$.

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.

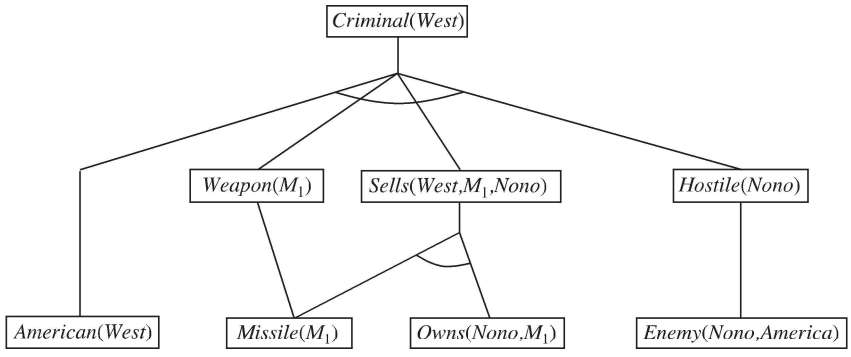


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

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

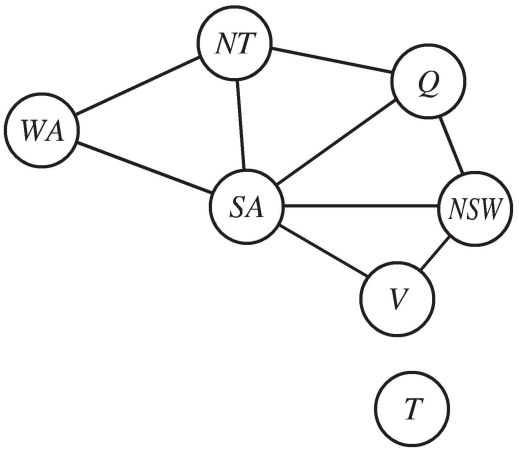
$$Missile(x) \Rightarrow Weapon(x) \;.$$

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.

9.4 BACKWARD CHAINING

The second major family of logical inference algorithms uses the **backward chaining** approach introduced in Section 7.5 for definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. We describe the basic algorithm, and then we describe how it is used in **logic programming**, which is the most widely used form of automated reasoning. We also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we look at the close connection between logic programming and constraint satisfaction problems.

9.4.1 A backward-chaining algorithm

Figure 9.6 shows a backward-chaining algorithm for definite clauses. FOL-BC-ASK(KB, $goal$) will be proved if the knowledge base contains a clause of the form $lhs \Rightarrow goal$, where lhs (left-hand side) is a list of conjuncts. An atomic fact like $American(West)$ is considered as a clause whose lhs is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query $Person(x)$ could be proved with the substitution $\{x/John\}$ as well as with $\{x/Richard\}$. So we implement FOL-BC-ASK as a **generator**—a function that returns multiple times, each time giving one possible result.

GENERATOR

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 9.7 is the proof tree for deriving $Criminal(West)$ from sentences (9.3) through (9.10).

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we show how backward chaining is used in logic programming systems.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ'

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first,rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

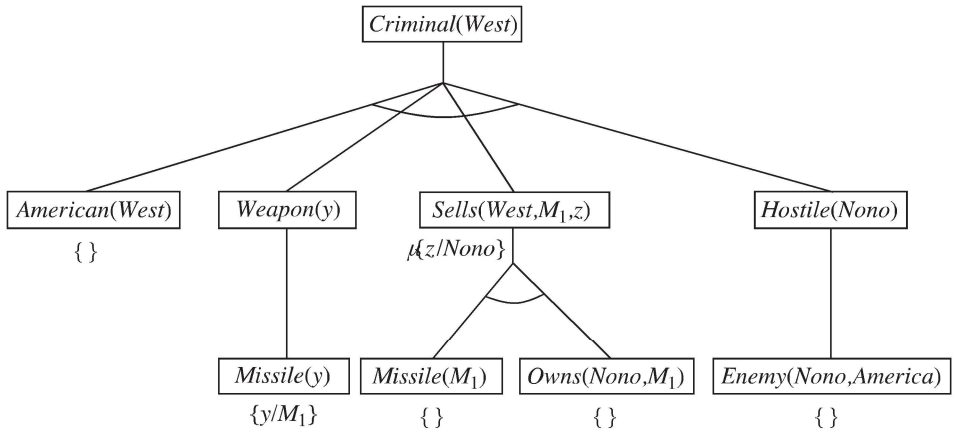


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

9.4.2 Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal described in Chapter 7: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski's equation,

$$Algorithm = Logic + Control\ .$$

PROLOG

Prolog is the most widely used logic programming language. It is used primarily as a rapid-prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants—the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written "backwards" from what we are used to; instead of $A \wedge B \Rightarrow C$ in Prolog we have `C :- A, B`. Here is a typical example:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

The notation `[E|L]` denotes a list whose first element is `E` and whose rest is `L`. Here is a Prolog program for `append(X,Y,Z)`, which succeeds if list `Z` is the result of appending lists `X` and `Y`:

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

In English, we can read these clauses as (1) appending an empty list with a list `Y` produces the same list `Y` and (2) `[A|Z]` is the result of appending `[A|X]` onto `Y`, provided that `Z` is the result of appending `X` onto `Y`. In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually much more powerful, however, because it describes a *relation* that holds among three arguments, rather than a *function* computed from two arguments. For example, we can ask the query `append(X,Y,[1,2])`: what two lists can be appended to give `[1,2]`? We get back the solutions

```
X=[]     Y=[1,2];
X=[1]    Y=[2];
X=[1,2]  Y=[]
```

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects of Prolog fall outside standard logical inference:

- Prolog uses the database semantics of Section 8.2.8 rather than first-order semantics, and this is apparent in its treatment of equality and negation (see Section 9.4.5).
- There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference. For example, the

goal "`X is 4+3`" succeeds with `X` bound to 7. On the other hand, the goal "`5 is X+Y`" fails, because the built-in functions do not do arbitrary equation solving.[5]

- There are built-in predicates that have side effects when executed. These include input–output predicates and the `assert/retract` predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.

Prolog's design represents a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed.

9.4.3 Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure 9.6, with the program as the knowledge base. We say "essentially" because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two.

First, our implementation had to explicitly manage the iteration over possible results generated by each of the subfunctions. Prolog interpreters have a global data structure,

CHOICE POINT

a stack of **choice points**, to keep track of the multiple possibilities that we considered in FOL-BC-OR. This global stack is more efficient, and it makes debugging easier, because the debugger can move up and down the stack.

Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating substitutions. Instead of explicitly constructing substitutions, Prolog has logic variables that remember their current binding. At any point in time, every variable in the program either is unbound or is bound to some value. Together, these variables and values implicitly define the substitution for the current branch of the proof. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already bound variable results in a failure of unification. When a path in the search fails, Prolog will back up to a previous choice point, and then it might have to unbind some variables. This is done

TRAIL

by keeping track of all the variables that have been bound in a stack called the **trail**. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

Even the most efficient Prolog interpreters require several thousand machine instructions per inference step because of the cost of index lookup, unification, and building the recursive call stack. In effect, the interpreter always behaves as if it has never seen the program before; for example, it has to *find* clauses that match the goal. A compiled Prolog

[5] Note that if the Peano axioms are provided, such goals can be solved by inference within a Prolog program.