13 Logic and Resolution Proof

In this chapter, you learn about *logic*, an important addition to your knowledge of problem-solving paradigms.

Like the other paradigms, logic has both seductive advantages and bothersome disadvantages. On the positive side, the ideas of logic, having matured for centuries, are concise and universally understood, like Latin. Moreover, until recently, logicians have focused on proving theorems about what can be proved. Consequently, when a problem can be attacked by logic successfully, you are in luck, for you know exactly what you can and cannot do.

On the negative side, logic can be a procrustean bed, for concentrating on logic can lead to concentrating on the mathematics of logic, deflecting attention away from valuable problem-solving methods that resist mathematical analysis.

First, you learn how to handle the notation used in logic; building on that notation, you see how rules of inference, such as *modus ponens*, *modus tolens*, and *resolution*, make it possible to create new expressions from existing ones. Then, you explore the notion of proof, and you use *proof by refutation* and *resolution theorem proving*.

In the course of learning about logic, you are exposed to a blizzard of new concepts. Accordingly, the key points in this chapter are illustrated with ridiculously simple examples designed to keep human intuition fully engaged. These examples stick to the blocks world, showing how one relation can be deduced from others.

RULES OF INFERENCE

You know that something is a bird if it has feathers or if it flies and lays eggs. This knowledge was expressed before, in Chapter 7, in the form of if–then rules:

I3	If	the animal has feathers
	then	it is a bird
I4	If	the animal flies it lays eggs
	then	it is a bird

In this section, you see the same sort of knowledge expressed in the language of logic, and you learn about the rules of inference that make it possible to use knowledge expressed in that language.

Logic Has a Traditional Notation

In logic, to express the sense of the antecedent–consequent rule concerning feathers and birds, you need a way to capture the idea that something has feathers and that something is a bird. You capture such ideas by using **predicates**, for predicates are **functions** that map object **arguments** into true or false values.

For example, with the normal way of interpreting the object Albatross and the predicates Feathers and Bird, you can say, informally, that the following are true expressions:

Feathers(Albatross)

Bird(Albatross)

Now suppose you say that the following is a true expression:

Feathers(Squigs)

Evidently, Squigs is a symbol that denotes something that has feathers, constraining what Squigs can possibly name, for Squigs satisfies the Feathers predicate.

You can express other constraints with other predicates, such as Flies and Lays-eggs. In fact, you can limit the objects that Squigs can name to those objects that satisfy *both* predicates together by saying that both of the following expressions are true:

Flies(Squigs)

Lays-eggs(Squigs)

There is a more traditional way to express this idea, however. You simply combine the first expression and the second expression and say that the *combination* is true:

Flies(Squigs) and Lays-eggs(Squigs)

Of course, you can also insist that Squigs names something that satisfies either of the two predicates. You specify this constraint as follows:

Flies(Squigs) or Lays-eggs(Squigs)

Logicians prefer a different notation, however. They like to write *and* as & and *or* as $\vee$.[†]

Now you can rewrite the expressions you wrote before, recasting them as a logician would write them:

Flies(Squigs)&Lays-eggs(Squigs)

Flies(Squigs) $\vee$ Lays-eggs(Squigs)

When expressions are joined by &, they form a **conjunction**, and each part is called a **conjunct**. Similarly, when expressions are joined by $\vee$, they form a **disjunction**, and each part is called a **disjunct**.

Note that & and $\vee$ are called **logical connectives** because they map combinations of true and false to true or false.

In addition to & and $\vee$, there are two other essential connectives: one is *not*, written as $\neg$, and the other is *implies*, written as $\Rightarrow$. Consider this:

$\neg$Feathers(Suzie)

For this expression to be true, Suzie must denote something for which Feathers(Suzie) is not true. That is, Suzie must be something for which the predicate Feathers is *not satisfied.*

Moving on, using $\Rightarrow$, here is an expression that resembles one of the antecedent–consequent rules:

Feathers(Suzie) $\Rightarrow$ Bird(Suzie)

Saying that the value of this expression is true constrains what Suzie can denote. One allowed possibility is that Suzie is something for which both Feathers(Suzie) and Bird(Suzie) are true. Naturally, the definition of $\Rightarrow$ also allows both Feathers(Suzie) and Bird(Suzie) to be false. Curiously, another possibility, allowed by the definition of $\Rightarrow$, is that Feathers(Suzie) is false but Bird(Suzie) is true. If Feathers(Suzie) is true and Bird(Suzie) is false, however, then the expression Feathers(Suzie) $\Rightarrow$ Bird(Suzie) is false.

Perhaps it is time to be more precise about the $\Rightarrow$, &, $\vee$, and $\neg$ connectives, before it is too late. Thinking of them as functions, it is easy to define them by listing the approved value for each possible combination of arguments. Such a list is shown in figure 13.1, which contains diagrams that are called **truth tables**.

Note that the connectives have an accepted **precedence**. In ordinary arithmetic, a unary minus sign has precedence higher than that of a plus sign, so you can write $-a + b$, meaning $(-a) + b$, not $-(a + b)$. Similarly, because $\neg$ has precedence higher than that of $\vee$, you can write $\neg E_1 \vee E_2$, meaning $(\neg E_1) \vee E_2$, without any possibility of confusion with $\neg(E_1 \vee E_2)$.

[†]Actually, most logicians write *and* as $\wedge$, instead of as &. In this book, & is used because it is easy for beginners to distinguish & from $\vee$.

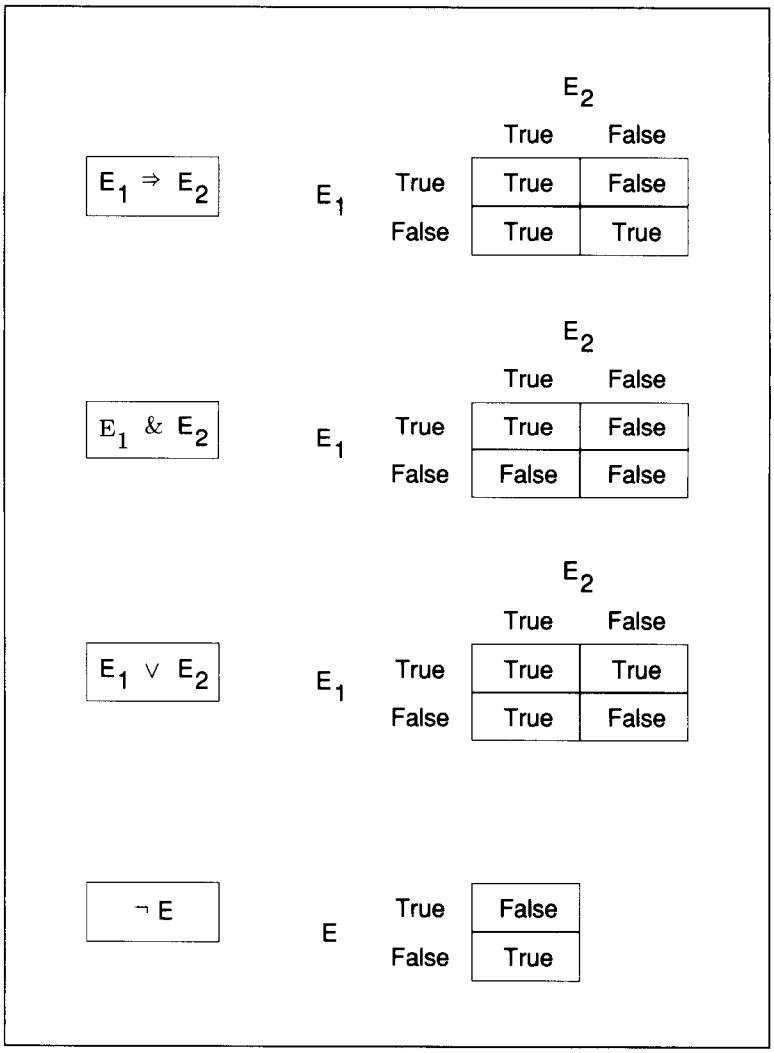

Figure 13.1 Truth tables show what ⇒, &, ∨, and ¬ do.

The accepted precedence is ¬ first, followed by & and ∨, with ⇒ bringing up the rear. A good habit is to use parentheses liberally, even when not strictly necessary, to reduce the likelihood of a mistake.

Note that the truth-table definition for ⇒ indicates that the values of $E_1 \Rightarrow E_2$ are the same as the values of $\neg E_1 \vee E_2$ for all combinations of values for E_1 and E_2. Consequently, and important to note, $\neg E_1 \vee E_2$ can be substituted for $E_1 \Rightarrow E_2$, and vice versa, at any time. Rules for reversible substitution are expressed by a ⇔ symbol:

$$E_1 \Rightarrow E_2 \Leftrightarrow \neg E_1 \vee E_2$$

Truth tables also demonstrate other useful properties of logical connectives, which are listed here, partly for the sake of completeness and partly because

you need some of them to deal with forthcoming examples. First, the & and ∨ connectives are **commutative**:

$$E_1 \& E_2 \Leftrightarrow E_2 \& E_1$$
$$E_1 \vee E_2 \Leftrightarrow E_2 \vee E_1$$

Next, they are **distributive**:

$$E_1 \& (E_2 \vee E_3) \Leftrightarrow (E_1 \& E_2) \vee (E_1 \& E_3)$$
$$E_1 \vee (E_2 \& E_3) \Leftrightarrow (E_1 \vee E_2) \& (E_1 \vee E_3)$$

In addition, they are **associative**:

$$E_1 \& (E_2 \& E_3) \Leftrightarrow (E_1 \& E_2) \& E_3$$
$$E_1 \vee (E_2 \vee E_3) \Leftrightarrow (E_1 \vee E_2) \vee E_3$$

They obey **de Morgan's laws**:

$$\neg(E_1 \& E_2) \Leftrightarrow (\neg E_1) \vee (\neg E_2)$$
$$\neg(E_1 \vee E_2) \Leftrightarrow (\neg E_1) \& (\neg E_2)$$

And finally, two ¬ symbols annihilate each other:

$$\neg(\neg E_1) \Leftrightarrow E_1$$

Quantifiers Determine When Expressions Are True

To signal that an expression is universally true, you use a symbol meaning *for all*, written as ∀, as well as a variable standing in for possible objects. In the following example, the expression, when true, says that any object having feathers is a bird:

$$\forall x[\text{Feathers}(x) \Rightarrow \text{Bird}(x)]$$

Like other expressions, $\forall x[\text{Feathers}(x) \Rightarrow \text{Bird}(x)]$ can be true or false. If true, a ∀ expression means that you get a true expression when you substitute any object for x inside the square brackets. For example, if $\forall x[\text{Feathers}(x) \Rightarrow \text{Bird}(x)]$ is true, then certainly Feathers(Squigs) ⇒ Bird(Squigs) is true and Feathers(Suzie) ⇒ Bird(Suzie) is true.

When an expression is surrounded by the square brackets associated with a quantifier, the expression is said to lie within the **scope** of that quantifier. The expression $\text{Feathers}(x) \Rightarrow \text{Bird}(x)$ therefore lies within the scope of the ∀ quantifier.

Because true expressions starting with ∀ say something about all possible object-for-variable substitutions within their scope, they are said to be **universally quantified**. Consequently, ∀ is called the **universal quantifier**.

Some expressions, although not always true, are true at least for some objects. Logic captures this idea using a symbol meaning *there exists*, written as ∃, used like this:

$$\exists x[\text{Bird}(x)]$$

When true, this expression means that there is at least one possible object, that, when substituted for x, makes the expression inside the square brackets true. Perhaps Bird(Squigs) is true; in any case, something like Bird(Squigs) is true.

Expressions with $\exists$ are said to be **existentially quantified**. The symbol $\exists$ is called the **existential quantifier**.

Logic Has a Rich Vocabulary

One problem with logic is that there is a large vocabulary to keep straight. For reference, let us gather together and complete the common elements of that vocabulary now, by way of figure 13.2 and the following definitions:

- A world's **objects** are terms.
- **Variables** ranging over a world's objects are terms.
- **Functions** are terms. The arguments to functions and the values returned are terms.

Terms are the only things that appear as arguments to predicates.

- **Atomic formulas** are individual predicates, together with arguments.
- **Literals** are atomic formulas and negated atomic formulas.
- **Well-formed formulas**, generally referred to, regrettably, by the abbreviation *wffs*, are defined recursively: literals are wffs; wffs connected together by, $\neg$, &, $\vee$, and $\Rightarrow$ are wffs; and wffs surrounded by quantifiers are also wffs.

For wffs, there are some special cases:

- A wff in which all the variables, if any, are inside the scope of corresponding quantifiers is a **sentence**. These are sentences:

$$\forall x[\text{Feathers}(x) \Rightarrow \text{Bird}(x)]$$
$$\text{Feathers(Albatross)} \Rightarrow \text{Bird(Albatross)}$$

Variables such as x, appearing within the scope of corresponding quantifiers, are said to be **bound**. Variables that are not bound are **free**. The following expression is not a sentence, because it contains a free variable, y:

$$\forall x[\text{Feathers}(x) \vee \neg\text{Feathers}(y)]$$

Note carefully that variables can represent objects only; variables cannot represent predicates. Consequently, this discussion is limited to a kind of logic called **first-order predicate calculus**. A more advanced topic, **second-order predicate calculus** permits variables representing predicates. A less advanced topic, **propositional calculus**, permits no variables of any kind.

- A wff consisting of a disjunction of literals is a **clause**.

Generally, the word *expression* is used interchangeably with wff, for using a lot of wffs makes it difficult to think about logic, instead of kennels.

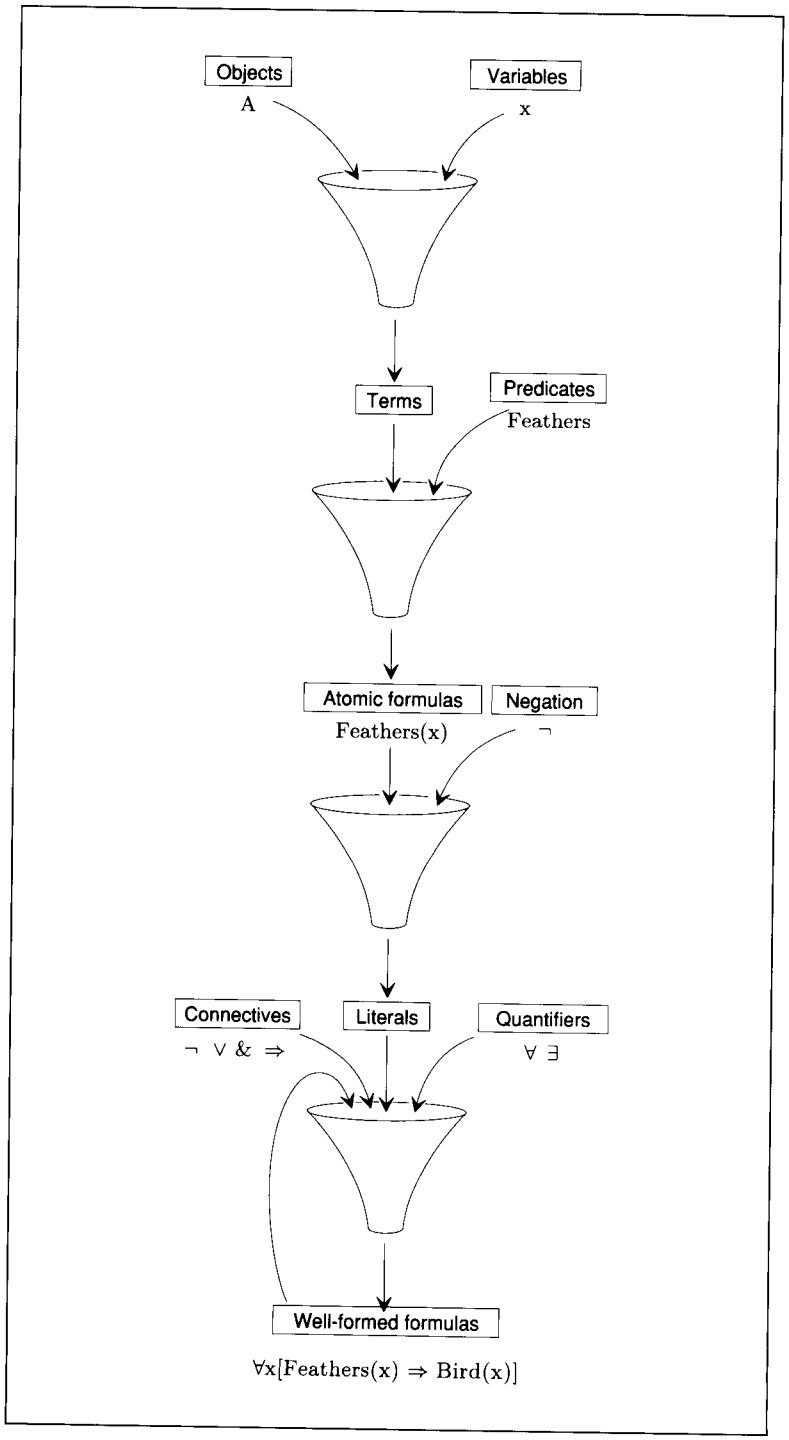

Figure 13.2 The vocabulary of logic. Informally, the sample well-formed formula says this: Stating that the unspecified object, x, satisfies the predicate Feathers implies that x satisfies the predicate Bird.

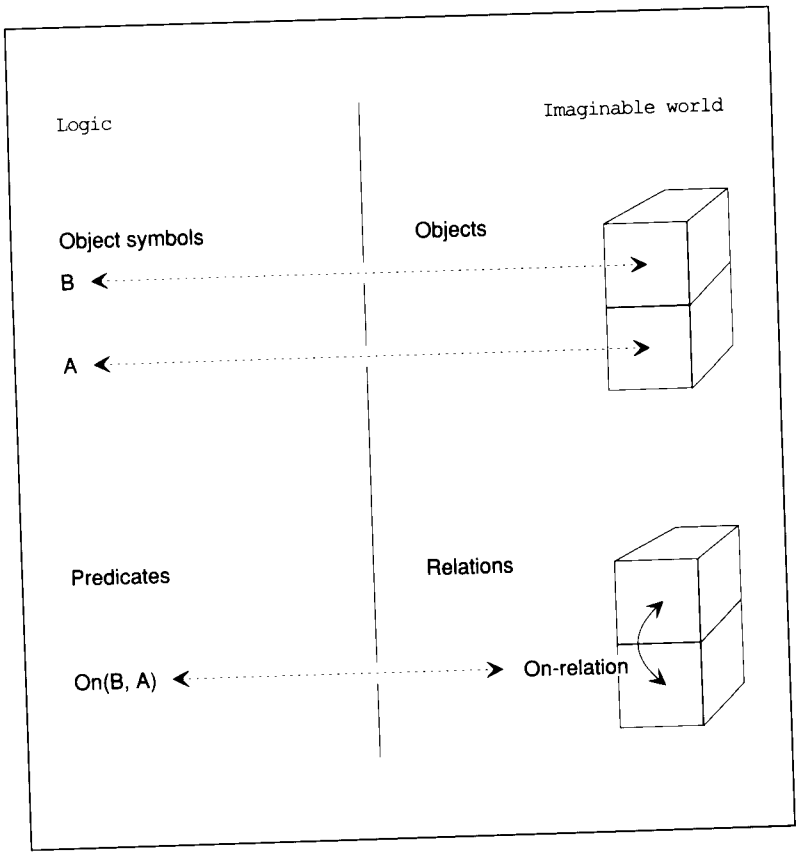

Figure 13.3 An interpretation is an accounting for how objects and relations map to object symbols and predicates.

Interpretations Tie Logic Symbols to Worlds

Ultimately, the point of logic is to say something about an imaginable world. Consequently, object symbols and predicates must be related to more tangible things. As figure 13.3 illustrates, the two symbol categories correspond to two world categories:

- Objects in a world correspond to object symbols in logic. In the example shown in figure 13.3, the object symbols A and B on the left correspond to two things in the imaginable world shown on the right.
- Relations in a world correspond to predicates in logic. Whenever a relation holds with respect to some objects, the corresponding predicate is true when applied to the corresponding object symbols. In the example, the logic-world predicate, On, applied to object symbols B and A, is true because the imaginable-world relation, On-relation, holds between the two imaginable-world objects.

An **interpretation** is a full accounting of the correspondence between objects and object symbols, and between relations and predicates.

Proofs Tie Axioms to Consequences

Now you are ready to explore the notion of **proof**. Suppose that you are told that both of the following expressions are true:

$$\text{Feathers(Squigs)}$$
$$\forall x[\text{Feathers}(x) \Rightarrow \text{Bird}(x)]$$

From the perspective of interpretations, to say that such expressions are true means that you are restricting the interpretations for the object symbols and predicates to those objects and relations for which the implied imaginable-world relations hold. Any such interpretation is said to be a **model** for the expressions.

When you are told Feathers(Squigs) and $\forall x[\text{Feathers}(x) \Rightarrow \text{Bird}(x)]$ are true, those expressions are called **axioms**. Now suppose that you are asked to show that all interpretations that make the axioms true also make the following expression true:

$$\text{Bird(Squigs)}$$

If you succeed, you have **proved** that Bird(Squigs) is a **theorem** with respect to the axioms:

- Said in the simplest terms, you prove that an expression is a theorem when you *show* that the theorem must be true, *given* that the axioms are true.
- Said in the fanciest terms, you prove that an expression is a theorem when you show that any model for the axioms is also a model for the theorem. You say that the theorem **logically follows** from the axioms.

The way to prove a theorem is to use a **proof procedure**. Proof procedures use manipulations called **sound rules of inference** that produce new expressions from old expressions such that, said precisely, models of the old expressions are guaranteed to be models of the new ones too.

The most straightforward proof procedure is to apply sound rules of inference to the axioms, and to the results of applying sound rules of inference, until the desired theorem appears.

Note that proving a theorem is not the same as showing that an expression is **valid**, meaning that the expression is true for all possible interpretations of the symbols. Similarly, proving a theorem is not the same as showing that an expression is **satisfiable**, meaning that it is true for some possible interpretation of the symbols.

The most straightforward sound rule of inference used in proof procedures is *modus ponens*. **Modus ponens** says this: If there is an axiom of the form $E_1 \Rightarrow E_2$, and there is another axiom of the form E_1, then E_2 logically follows.

If E_2 is the theorem to be proved, you are done. If not, you might as well add E_2 to the axioms, for it will always be true when all the rest of the axioms are true. Continuing with *modus ponens* on an ever-increasing

list of axioms may eventually show that the desired theorem is true, thus proving the theorem.

For the feathers-and-bird example, the axioms are just about right for the application of *modus ponens*. First, however, you must specialize the second expression. You have $\forall x$[Feathers(x) $\Rightarrow$ Bird(x)]. Because you are dealing with interpretations for which Feathers(x) $\Rightarrow$ Bird(x) is true for all x, it must be true for the special case where x is Squigs. Consequently, Feathers(Squigs) $\Rightarrow$ Bird(Squigs) must be true.

Now, the first expression, Feathers(Squigs), and the specialization of the second expression, Feathers(Squigs) $\Rightarrow$ Bird(Squigs), fit *modus ponens* exactly, once you substitute Feathers(Squigs) for E_1 and Bird(Squigs) for E_2. You conclude that Bird(Squigs) must be true. The theorem is proved.

Resolution Is a Sound Rule of Inference

One of the most important rules of inference is *resolution*. **Resolution** says this: If there is an axiom of the form $E_1 \vee E_2$, and there is another axiom of the form $\neg E_2 \vee E_3$, then $E_1 \vee E_3$ logically follows. The expression $E_1 \vee E_3$ is called the **resolvent** of $E_1 \vee E_2$ and $\neg E_2 \vee E_3$.

Let us look at the various possibilities to see whether resolution is believable. First, suppose E_2 is true; then $\neg E_2$ must be false. But if $\neg E_2$ is false, from the second expression, then E_3 must be true. But if E_3 is true, then surely $E_1 \vee E_3$ is true. Second, suppose that E_2 is false. Then, from the first expression, E_1 must be true. But if E_1 is true, then surely $E_1 \vee E_3$ is true. You conclude that the resolvent, $E_1 \vee E_3$, must be true as long as both $E_1 \vee E_2$ and $\neg E_2 \vee E_3$ are true.

It is easy to generalize resolution such that there can be any number of disjuncts, including just one, in either of the two resolving expressions. The only demand is that one resolving expression must have a disjunct that is the negation of a disjunct in the other resolving expression. Once generalized, you can use resolution to reach the same conclusion about Squigs that you reached before with *modus ponens*.

The first step is to specialize the quantified expression to Squigs. The next step is to rewrite it, eliminating $\Rightarrow$, producing these:

Feathers(Squigs)

$\neg$Feathers(Squigs) $\vee$ Bird(Squigs)

So written, resolution obviously applies, dropping out Feathers(Squigs) and $\neg$Feathers(Squigs), producing Bird(Squigs).

As a matter of fact, this example suggests a general truth: *Modus ponens* can be viewed as a special case of resolution, because anything concluded with *modus ponens* can be concluded with resolution as well. To see why, let one expression be E_1, and let the other be $E_1 \Rightarrow E_2$. According to *modus ponens*, E_2 must be true. But you know that $E_1 \Rightarrow E_2$ can be rewritten as $\neg E_1 \vee E_2$. So rewritten, resolution can be applied, dropping

out the E_1 and the $\neg E_1$, producing E_2, which is the same result that you obtained using *modus ponens*.

Similarly, resolution subsumes another rule of inference called *modus tolens*. **Modus tolens** says this: If there is an axiom of the form $E_1 \Rightarrow E_2$, and there is another axiom of the form $\neg E_2$, then $\neg E_1$ logically follows.

RESOLUTION PROOFS

To prove a theorem, one obvious strategy is to search forward from the axioms using sound rules of inference, hoping to stumble across the theorem eventually. In this section, you learn about another strategy, the one used in resolution theorem proving, that requires you to show that the negation of a theorem cannot be true:

- Assume that the negation of the theorem is true.
- Show that the axioms and the assumed negation of the theorem together force an expression to be true that cannot be true.
- Conclude that the assumed negation of the theorem cannot be true because it leads to a contradiction.
- Conclude that the theorem must be true because the assumed negation of the theorem cannot be true.

Proving a theorem by showing its negation cannot be true is called **proof by refutation.**

Resolution Proves Theorems by Refutation

Consider the Squigs example again. Recall that you know from the axioms the following:

$$\neg\text{Feathers(Squigs)} \vee \text{Bird(Squigs)}$$
$$\text{Feathers(Squigs)}$$

Adding the negation of the expression to be proved, you have this list:

$$\neg\text{Feathers(Squigs)} \vee \text{Bird(Squigs)}$$
$$\text{Feathers(Squigs)}$$
$$\neg\text{Bird(Squigs)}$$

Resolving the first and second axiom, as before, permits you to add a new expression to the list:

$$\neg\text{Feathers(Squigs)} \vee \text{Bird(Squigs)}$$
$$\text{Feathers(Squigs)}$$
$$\neg\text{Bird(Squigs)}$$
$$\text{Bird(Squigs)}$$

But now there is a contradiction. All the things in the list are supposed to be true. But it cannot be that Bird(Squigs) and ¬Bird(Squigs) are both true. Consequently, the assumption that led to this contradiction must be

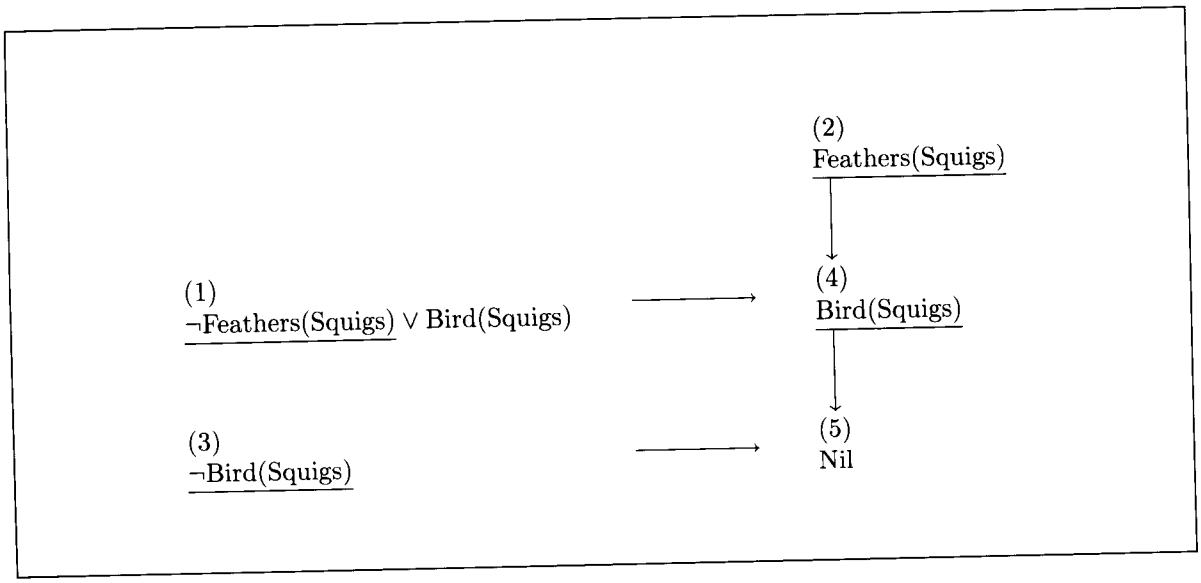

Figure 13.4 A tree recording the resolutions needed to prove Bird(Squigs).

false; that is, the negation of the theorem, ¬Bird(Squigs), must be false; hence, the theorem, Bird(Squigs), must be true, which is what you set out to show.

The traditional way to recognize that the theorem is proved is to wait until resolution happens on a literal and that literal's contradicting negation. The result is an empty clause—one with nothing in it—which by convention is written as **Nil**. When resolution produces Nil, you are guaranteed that resolution has produced manifestly contradictory expressions. Consequently, production of Nil is the signal that resolution has proved the theorem.

Usually, it is illuminating to use a treelike diagram to record how clauses get resolved together on the way to producing an empty clause. Figure 13.4 is the tree for the proof.

Using Resolution Requires Axioms to Be in Clause Form

Now that you have the general idea of how proof by resolution works, it is time to understand various manipulations that make harder proofs possible. Basically, the point of these manipulations is to transform arbitrary logic expressions into a form that enables resolution. Specifically, you need a way to transform the given axioms into equivalent, new axioms that are all disjunctions of literals. Said another way, you want the new axioms to be in **clause form**.

An axiom involving blocks illustrates the manipulations. Although the axiom is a bit artificial, so as to exercise all the transformation steps, the axiom's message is simple. First, a brick is on something that is not a pyramid; second, there is nothing that a brick is on and that is on the

brick as well; and third, there is nothing that is not a brick and also is the same thing as the brick:

$$\forall x[\text{Brick}(x) \Rightarrow (\exists y[\text{On}(x, y)\&\neg\text{Pyramid}(y)] \\ \&\neg\exists y[\text{On}(x, y)\&\text{On}(y, x)] \\ \&\forall y[\neg\text{Brick}(y) \Rightarrow \neg\text{Equal}(x, y)])]$$

As given, however, the axiom cannot be used to produce resolvents because it is not in clause form. Accordingly, the axiom has to be transformed into one or more equivalent axioms in clause form. You soon see that the transformation leads to four new axioms and requires the introduction of another function, Support:

$$\neg\text{Brick}(x) \vee \text{On}(x, \text{Support}(x))$$
$$\neg\text{Brick}(w) \vee \neg\text{Pyramid}(\text{Support}(w))$$
$$\neg\text{Brick}(u) \vee \neg\text{On}(u, y) \vee \neg\text{On}(y, u)$$
$$\neg\text{Brick}(v) \vee \text{Brick}(z) \vee \neg\text{Equal}(v, z)$$

Next, let us consider the steps needed to transform arbitrary logical expressions into clause form. Once explained, the steps will be summarized in a procedure.

- Eliminate implications.

The first thing to do is to get rid of all the implications. This step is easy: All you need to do is to substitute $\neg E_1 \vee E_2$ for $E_1 \Rightarrow E_2$. For the example, you have to make two such substitutions, leaving you with this:

$$\forall x[\neg\text{Brick}(x) \vee (\exists y[\text{On}(x, y)\&\neg\text{Pyramid}(y)] \\ \&\neg\exists y[\text{On}(x, y)\&\text{On}(y, x)] \\ \&\forall y[\neg(\neg\text{Brick}(y)) \vee \neg\text{Equal}(x, y)])]$$

- Move negations down to the atomic formulas.

Doing this step requires a number of identities, one for dealing with the negation of & expressions, one for $\vee$ expressions, one for $\neg$ expressions, and one each for $\forall$ and $\exists$:

$$\neg(E_1\&E_2) \rightarrow (\neg E_1) \vee (\neg E_2)$$
$$\neg(E_1 \vee E_2) \rightarrow (\neg E_1)\&(\neg E_2)$$
$$\neg(\neg E_1) \rightarrow E_1$$
$$\neg\forall x[E_1(x)] \rightarrow \exists x[\neg E_1(x)]$$
$$\neg\exists x[E_1(x)] \rightarrow \forall x[\neg E_1(x)]$$

For the example, you need the third identity, which eliminates the double negations, and you need the final identity, which eliminates an $\exists$ and introduces another $\forall$, leaving this:

$$\forall x[\neg\text{Brick}(x) \vee (\exists y[\text{On}(x, y)\&\neg\text{Pyramid}(y)] \\ \&\forall y[\neg\text{On}(x, y) \vee \neg\text{On}(y, x)] \\ \&\forall y[\text{Brick}(y) \vee \neg\text{Equal}(x, y)])]$$

- Eliminate existential quantifiers.

Unfortunately, the procedure for eliminating existential quantifiers is a little obscure, so you must work hard to understand it. Let us begin by looking closely at the part of the axiom involving $\exists$:

$$\exists y[\text{On}(x, y)\&\neg\text{Pyramid}(y)]$$

Reflect on what this expression means. Evidently, if someone gives you some particular object x, you will be able to identify an object for y that makes the expression true. Said another way, there is a function that takes argument x and returns a proper y. You do not necessarily know how the function works, but such a function must exist. Let us call it, for the moment, Magic(x).

Using the new function, you no longer need to say that y exists, for you have a way of producing the proper y in any circumstance. Consequently, you can rewrite the expression as follows:

$$\text{On}(x, \text{Magic}(x))\&\neg\text{Pyramid}(\text{Magic}(x))$$

Functions that eliminate the need for existential quantifiers are called **Skolem functions**. Note carefully that the Skolem function, Magic(x), must depend on x, for otherwise it could not produce a y that depends on a particular x. The general rule is that the universal quantifiers determine which arguments Skolem functions need: There is one argument for each universally quantified variable whose scope contains the Skolem function.

Here then is the evolving axiom, after eliminating the $\exists$ and introducing the Skolem function, which you now can call the Support function:

$$\begin{aligned}\forall x[\neg\text{Brick}(x) \vee ((&\text{On}(x, \text{Support}(x))\&\neg\text{Pyramid}(\text{Support}(x)))\\ &\&\forall y[\neg\text{On}(x, y) \vee \neg\text{On}(y, x)]\\ &\&\forall y[\text{Brick}(y) \vee \neg\text{Equal}(x, y)])]\end{aligned}$$

- Rename variables, as necessary, so that no two variables are the same.

The quantifiers do not care what their variable names are. Accordingly, you can rename any duplicates so that each quantifier has a unique name. You do this renaming because you want to move all the universal quantifiers together at the left of each expression in the next step, without confounding them. In the example, the substitutions leave this:

$$\begin{aligned}\forall x[\neg\text{Brick}(x) \vee ((&\text{On}(x, \text{Support}(x))\&\neg\text{Pyramid}(\text{Support}(x)))\\ &\&\forall y[\neg\text{On}(x, y) \vee \neg\text{On}(y, x)]\\ &\&\forall z[\text{Brick}(z) \vee \neg\text{Equal}(x, z)])]\end{aligned}$$

- Move the universal quantifiers to the left.

This step works because, by now, each quantifier uses a unique variable name—no confusion results from leftward movement. In the example, the result is as follows:

$$\forall x \forall y \forall z [\neg \text{Brick}(x) \vee ((\text{On}(x, \text{Support}(x)) \& \neg \text{Pyramid}(\text{Support}(x))) \\ \& \neg \text{On}(x, y) \vee \neg \text{On}(y, x) \\ \& \text{Brick}(z) \vee \neg \text{Equal}(x, z))]$$

- Move the disjunctions down to the literals.

This step requires you to move the $\vee$s inside the &s; to do this movement, you need to use one of the distributive laws:

$$E_1 \vee (E_2 \& E_3) \Leftrightarrow (E_1 \vee E_2) \& (E_1 \vee E_3)$$

For the example, let us do the work in two steps:

$$\forall x \forall y \forall z [(\neg \text{Brick}(x) \vee (\text{On}(x, \text{Support}(x)) \& \neg \text{Pyramid}(\text{Support}(x)))) \\ \& (\neg \text{Brick}(x) \vee \neg \text{On}(x, y) \vee \neg \text{On}(y, x)) \\ \& (\neg \text{Brick}(x) \vee \text{Brick}(z) \vee \neg \text{Equal}(x, z))]$$

$$\forall x \forall y \forall z [(\neg \text{Brick}(x) \vee \text{On}(x, \text{Support}(x))) \\ \& (\neg \text{Brick}(x) \vee \neg \text{Pyramid}(\text{Support}(x))) \\ \& (\neg \text{Brick}(x) \vee \neg \text{On}(x, y) \vee \neg \text{On}(y, x)) \\ \& (\neg \text{Brick}(x) \vee \text{Brick}(z) \vee \neg \text{Equal}(x, z))]$$

- Eliminate the conjunctions.

Actually, you do not really eliminate them. Instead, you simply write each part of a conjunction as though it were a separate axiom. This way of writing a conjunction makes sense, because each part of a conjunction must be true if the whole conjunction is true. Here is the result:

$$\forall x [\neg \text{Brick}(x) \vee \text{On}(x, \text{Support}(x))]$$
$$\forall x [\neg \text{Brick}(x) \vee \neg \text{Pyramid}(\text{Support}(x))]$$
$$\forall x \forall y [\neg \text{Brick}(x) \vee \neg \text{On}(x, y) \vee \neg \text{On}(y, x)]$$
$$\forall x \forall z [\neg \text{Brick}(x) \vee \text{Brick}(z) \vee \neg \text{Equal}(x, z)]$$

- Rename all the variables, as necessary, so that no two variables are the same.

There is no problem with renaming variables at this step, for you are merely renaming the universally quantified variables in each part of a conjunction. Because each of the conjoined parts must be true for any variable values, it does not matter whether the variables have different names for each part. Here is the result for the example:

$$\forall x [\neg \text{Brick}(x) \vee \text{On}(x, \text{Support}(x))]$$
$$\forall w [\neg \text{Brick}(w) \vee \neg \text{Pyramid}(\text{Support}(w))]$$
$$\forall u \forall y [\neg \text{Brick}(u) \vee \neg \text{On}(u, y) \vee \neg \text{On}(y, u)]$$
$$\forall v \forall z [\neg \text{Brick}(v) \vee \text{Brick}(z) \vee \neg \text{Equal}(v, z)]$$

- Eliminate the universal quantifiers.

Actually, you do not really eliminate them. You just adopt a convention whereby all variables at this point are presumed to be universally quantified. Now, the example looks like this:

$$\neg\text{Brick}(x) \vee \text{On}(x, \text{Support}(x))$$
$$\neg\text{Brick}(w) \vee \neg\text{Pyramid}(\text{Support}(w))$$
$$\neg\text{Brick}(u) \vee \neg\text{On}(u, y) \vee \neg\text{On}(y, u)$$
$$\neg\text{Brick}(v) \vee \text{Brick}(z) \vee \neg\text{Equal}(v, z)$$

The result is now in clause form, as required when you wish to use resolution. Each clause consists of a disjunction of literals. Taking the whole set of clauses together, you have an implied & on the top level, literals on the bottom level, and $\vee$s in between. Each clause's variables are different, and all variables are implicitly universally quantified. To summarize, here is the procedure for translating axioms into clause form:

To put axioms into clause form,

- ▷ Eliminate the implications.
- ▷ Move the negations down to the atomic formulas.
- ▷ Eliminate the existential quantifiers.
- ▷ Rename the variables, if necessary.
- ▷ Move the universal quantifiers to the left.
- ▷ Move the disjunctions down to the literals.
- ▷ Eliminate the conjunctions.
- ▷ Rename the variables, if necessary.
- ▷ Eliminate the universal quantifiers.

Here is the procedure for doing resolution proof:

To prove a theorem using resolution,

- ▷ Negate the theorem to be proved, and add the result to the list of axioms.
- ▷ Put the list of axioms into clause form.
- ▷ Until there is no resolvable pair of clauses,
  - ▷ Find resolvable clauses and resolve them.
  - ▷ Add the results of resolution to the list of clauses.
  - ▷ If Nil is produced, stop and report that the theorem is true.
- ▷ Stop and report that the theorem is false.

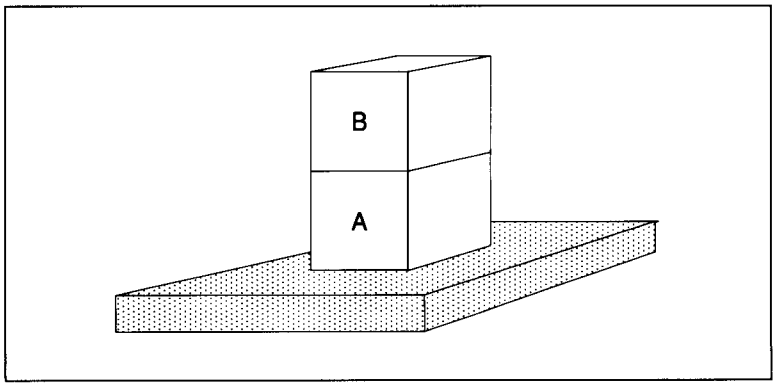

Figure 13.5 Some fodder for a proof.

Before discussing which clause pairs to resolve at any given point, let us work out an example. The following axioms account for the observed block relations in figure 13.5:

$$\text{On(B, A)}$$
$$\text{On(A, Table)}$$

These axioms, of course, are already in clause form. Let us use them to show that B is above the table:

$$\text{Above(B, Table)}$$

To show this, you need the clause form of two universally quantified expressions. The first says that being on an object implies being above that object. The second says that one object is above another if there is an object in between:

$$\forall x \forall y[\text{On}(x, y) \Rightarrow \text{Above}(x, y)]$$
$$\forall x \forall y \forall z[\text{Above}(x, y) \& \text{Above}(y, z) \Rightarrow \text{Above}(x, z)]$$

After you go through the procedure for reduction to clause form, these axioms look like this:

$$\neg\text{On}(u, v) \vee \text{Above}(u, v)$$
$$\neg\text{Above}(x, y) \vee \neg\text{Above}(y, z) \vee \text{Above}(x, z)$$

Recall that the expression to be proved is Above(B, Table). No conversion is needed after negation:

$$\neg\text{Above(B, Table)}$$

Next, all the clauses need identifying numbers to make it easy to refer to them:

$$\neg\text{On}(u, v) \vee \text{Above}(u, v) \quad (1)$$
$$\neg\text{Above}(x, y) \vee \neg\text{Above}(y, z) \vee \text{Above}(x, z) \quad (2)$$
$$\text{On(B, A)} \quad (3)$$
$$\text{On(A, Table)} \quad (4)$$
$$\neg\text{Above(B, Table)} \quad (5)$$

Now you can start. First, you resolve clause 2 and clause 5 by specializing x to B and z to Table so that the final part of clause 2 looks exactly like the expression negated in clause 5, producing clause 6:

$$\neg\text{Above}(\text{B}, y) \vee \neg\text{Above}(y, \text{Table}) \vee \underline{\text{Above}(\text{B}, \text{Table})} \quad (2)$$

$$\underline{\neg\text{Above}(\text{B}, \text{Table})} \quad (5)$$

$$\neg\text{Above}(\text{B}, y) \vee \neg\text{Above}(y, \text{Table}) \quad (6)$$

Now, you can resolve (1) with clause 6 by replacing u with y and specializing v to Table:

$$\neg\text{On}(y, \text{Table}) \vee \underline{\text{Above}(y, \text{Table})} \quad (1)$$

$$\neg\text{Above}(\text{B}, y) \vee \underline{\neg\text{Above}(y, \text{Table})} \quad (6)$$

$$\neg\text{On}(y, \text{Table}) \vee \neg\text{Above}(\text{B}, y) \quad (7)$$

Curiously, it pays to use (1) again with clause 7, with u specialized to B and v replaced by y:

$$\neg\text{On}(\text{B}, y) \vee \underline{\text{Above}(\text{B}, y)} \quad (1)$$

$$\neg\text{On}(y, \text{Table}) \vee \underline{\neg\text{Above}(\text{B}, y)} \quad (7)$$

$$\neg\text{On}(\text{B}, y) \vee \neg\text{On}(y, \text{Table}) \quad (8)$$

Now, let us use clause 3 and clause 8, specializing y to A:

$$\underline{\text{On}(\text{B}, \text{A})} \quad (3)$$

$$\underline{\neg\text{On}(\text{B}, \text{A})} \vee \neg\text{On}(\text{A}, \text{Table}) \quad (8)$$

$$\neg\text{On}(\text{A}, \text{Table}) \quad (9)$$

Now, clause 4 and clause 9 resolve to Nil, the empty clause.

$$\underline{\text{On}(\text{A}, \text{Table})} \quad (4)$$

$$\underline{\neg\text{On}(\text{A}, \text{Table})} \quad (9)$$

$$\text{Nil} \quad (10)$$

You must be finished: You have arrived at a contradiction, so the negation of the theorem, ¬Above(B, Table), must be false. Hence, the theorem, Above(B, Table), must be true.

Proof Is Exponential

One big question is, How can you be so shrewd as to pick just the right clauses to resolve? The answer is that you take advantage of two ideas:

- First, you can be sure that every resolution involves the negated theorem or a clause derived—directly or indirectly—using the negated theorem.
- Second, you know where you are, and you know where you are going, so you can note the difference and use your intuition.

Unfortunately, there are limits to what you can express if you restrict yourself to the mathematically attractive concepts in pure logic. For example, pure logic does not allow you to express concepts such as difference, as

required by means–ends analysis, or heuristic distances, as required by best-first search. Theorem provers can use such concepts, but then a large fraction of the problem-solving burden rests on knowledge lying outside the statement, in logical notation, of what is known and what is to be done.

Although some search strategies require you to separate yourself considerably from pure logic, others do not. One such strategy, the **unit-preference strategy**, gives preference to resolutions involving the clauses with the smallest number of literals. The **set-of-support strategy** allows only resolutions involving the negated theorem or new clauses derived—directly or indirectly—using the negated theorem. The **breadth-first strategy** first resolves all possible pairs of the initial clauses, then resolves all possible pairs of the resulting set together with the initial set, level by level. All these strategies are said to be **complete** because they are guaranteed to find a proof if the theorem logically follows from the axioms. Unfortunately, there is another side:

- All resolution search strategies, like many searches, are subject to the **exponential-explosion problem**, preventing success for proofs that require long chains of inference.
- All resolution search strategies are subject to a version of the **halting problem**, for search is not guaranteed to terminate unless there actually is a proof.

In fact, all complete proof procedures for the first-order predicate calculus are subject to the halting problem. Complete proof procedures are said to be **semidecidable** because they are guaranteed to tell you whether an expression is a theorem only if the expression is indeed a theorem.

Resolution Requires Unification

To resolve two clauses, two literals must match exactly, except that one is negated. Sometimes, literals match exactly as they stand; sometimes, literals can be made to match by an appropriate substitution.

In the examples so far, the matching part of resolution was easy: The same constant appeared in the same place, obviously matching, or a constant appeared in the place occupied by a universally quantified variable, matching because the variable could be the observed constant as well as any other.

You need a better way to keep track of substitutions, and you need the rules by which substitutions can be made. First, let us agree to denote substitutions as follows:

$$\{v_1 \rightarrow \mathrm{C}; v_2 \rightarrow v_3; v_4 \rightarrow f(\ldots)\}$$

This expression mean that the variable v_1 is replaced by the constant C, the variable v_2 is replaced by the variable v_3, and the variable v_4 is replaced by a function, f, together with the function's arguments.† The rules for

†Other authors denote the same substitution by $\{\mathrm{C}/v_1, v_3/v_2, f(\ldots)/v_4\}$, which is easier to write but harder to keep straight.

such substitutions say that you can replace a variable by any term that does not contain the same variable:

- You may replace a variable by a constant. That is, you can have the substitution $\{v_1 \rightarrow C\}$.
- You may replace a variable by a variable. That is, you can have the substitution $\{v_2 \rightarrow v_3\}$.
- You may replace a variable by a function expression, as long as the function expression does not contain the variable. That is, you can have the substitution $\{v_4 \rightarrow f(\ldots)\}$.

A substitution that makes two clauses resolvable is called a **unifier**, and the process of finding such substitutions is called **unification**. There are many procedures for unification. For the examples, however, inspection will do.

Traditional Logic Is Monotonic

Suppose that an expression is a theorem with respect to a certain set of axioms. Is the expression still a theorem after the addition of some new axioms? Surely it must be, for you can do the proof using the old axioms exclusively, ignoring the new ones.

Because new axioms only add to the list of provable theorems and never cause any to be withdrawn, traditional logic is said to be **monotonic**.

The monotonicity property is incompatible with some natural ways of thinking, however. Suppose that you are told all birds fly, from which you conclude that some particular bird flies, a perfectly reasonable conclusion, given what you know. Then, someone points out that penguins do not fly, nor do dead birds. Adding these new facts can block your already-made conclusion, but cannot stop a theorem prover; only amending the initial axioms can do that.

Research on this sort of problem has led to the development of logics that are said to be **nonmonotonic**.

Theorem Proving Is Suitable for Certain Problems, but Not for All Problems

Logic is terrific for some jobs, and is not so good for others. But because logic is unbeatable for what it was developed to do, logic is seductive. People try to use logic for all hard problems, rather than for only those for which it is suited. That is like using a hammer to drive screws, just because hammers are good at dealing with nails, which are simply one kind of fastener.

Consequently, when using logic and a theorem prover seems unambiguously right, review these caveats:

- Theorem provers may take too long.

Complete theorem provers require search, and the search is inherently exponential. Methods for speeding up search, such as set-of-support resolution,

reduce the size of the exponent associated with the search, but do not change the exponential character.

- Theorem provers may not help you to solve practical problems, even if they do their work instantaneously.

Some knowledge resists embodiment in axioms. Formulating a problem in logic may require enormous effort, whereas solving the problem formulated in another way may be simple.

- Logic is weak as a representation for certain kinds of knowledge.

The notation of pure logic does not allow you to express such notions as heuristic distances, or state differences, or the idea that one particular approach is particularly fast, or the idea that some manipulation works well, but only if done fewer than three times. Theorem provers can use such knowledge, but you must represent that knowledge using concepts other than those of pure logic.

SUMMARY

- Logic concentrates on using knowledge in a rigorous, provably correct way; other problem-solving paradigms concentrate on the knowledge itself.
- Logic has a traditional notation, requiring you to become familiar with the symbols for *implies*, *and*, *or*, and *not*. These symbols are $\Rightarrow$, &, $\vee$, and $\neg$.
- A universally quantified expression is true for all values of the quantified variable. An existentially quantified expression is true for at least one value.
- An interpretation is an account of how object symbols, predicates, and functions map to objects, relations, and functions in some imaginable world. A model of a set of expressions is an interpretation for which the implied imaginable-world relations hold.
- A theorem logically follows from assumed axioms if there is a series of steps connecting the theorem to the axioms using sound rules of inference.
- The most obvious rule of inference is *modus ponens*. Another, more general rule of inference is resolution.
- Resolution theorem proving uses resolution as the rule of inference and refutation as the strategy. Resolution requires transforming axioms and the negated theorem to clause form. Resolution also requires a variable substitution process called unification.
- The set-of-support strategy dictates using only resolutions in which at least one resolvent descends from the negation of the theorem to be proved.

- Logic is seductive, because it often works neatly. There are caveats that you must obey, however, for logic is just one of many tools that you should have in your workshop.

BACKGROUND

The development of the resolution method for theorem proving is generally credited to J. A. Robinson [1965, 1968].

PROLOG is a popular programming language based on logic. For an excellent introduction to PROLOG, see PROLOG *Programming for Artificial Intelligence* (second edition), by Ivan Bratko [1990]. PROLOG was developed by Alain Colmerauer and his associates [Colmerauer, H. Kanoui, R. Pasero, and P. Roussel 1973; Colmerauer 1982].

For an excellent treatment of the role of logic in artificial intelligence, see the papers of Patrick J. Hayes [1977].