Part II

Knowledge Representation

Chapter 4

Knowledge Representation Issues

In Chapter 1, we discussed the role that knowledge plays in AI systems. In succeeding chapters up until now, though, we have paid little attention to knowledge and its importance as we instead focused on basic frameworks for building search-based problem-solving programs. These methods are sufficiently general that we have been able to discuss them without reference to how the knowledge they need is to be represented. For example, in discussing the best-first search algorithm, we hid all the references to domain-specific knowledge in the generation of successors and the computation of the h' function. Although these methods are useful and form the skeleton of many of the methods we are about to discuss, their problem-solving power is limited precisely because of their generality. As we look in more detail at ways of representing knowledge, it becomes clear that particular knowledge representation models allow for more specific, more powerful problem-solving mechanisms that operate on them. In this part of the book, we return to the topic of knowledge and examine specific techniques that can be used for representing and manipulating knowledge within programs.

4.1 Representations and Mappings

In order to solve the complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanisms for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. But before we can talk about them individually, we must consider the following point that pertains to all discussions of representation, namely that we are dealing with two different kinds of entities:

- Facts: truths in some relevant world. These are the things we want to represent.
- Representations of facts in some chosen formalism. These are the things we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:

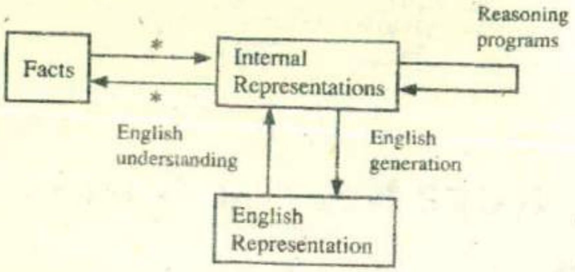


Figure 4.1: Mappings between Facts and Representations

- The *knowledge level*, at which facts (including each agent's behaviors and current goals) are described.
- The *symbol level*, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

See Newell [1982] for a detailed exposition of this view in the context of agents and their goals and behaviors. In the rest of our discussion here, we will follow a model more like the one shown in Figure 4.1. Rather than thinking of one level on top of another, we will focus on facts, on representations, and on the two-way mappings that must exist between them. We will call these links *representation mappings*. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

One representation of facts is so common that it deserves special mention: natural language (particularly English) sentences. Regardless of the representation for facts that we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. In this case, we must also have mapping functions from English sentences to the representation we are actually going to use and from it back to sentences. Figure 4.1 shows how these three kinds of objects relate to each other.

Let's look at a simple example using mathematical logic as the representational formalism. Consider the English sentence:

Spot is a dog.

The fact represented by that English sentence can also be represented in logic as:

$dog(Spot)$

Suppose that we also have a logical representation of the fact that all dogs have tails:

$\forall x : dog(x) \rightarrow hastail(x)$

Then, using the deductive mechanisms of logic, we may generate the new representation object:

hastail(*Spot*)

Using an appropriate backward mapping function, we could then generate the English sentence:

Spot has a tail.

Or we could make use of this representation of a new fact to cause us to take some appropriate action or to derive representations of additional facts.

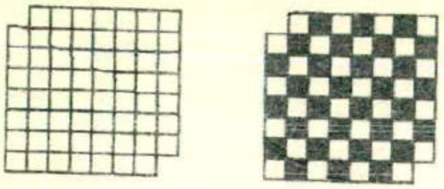
It is important to keep in mind that usually the available mapping functions are not one-to-one. In fact, they are often not even functions but rather many-to-many relations. (In other words, each object in the domain may map to several elements in the range, and several elements in the domain may map to the same element of the range.) This is particularly true of the mappings involving English representations of facts. For example, the two sentences "All dogs have tails" and "Every dog has a tail" could both represent the same fact, namely that every dog has at least one tail. On the other hand, the former could represent either the fact that every dog has at least one tail or the fact that each dog has several tails. The latter may represent either the fact that every dog has at least one tail or the fact that there is a tail that every dog has. As we will see shortly, when we try to convert English sentences into some other representation, such as logical propositions, we must first decide what facts the sentences represent and then convert those facts into the new representation.

The starred links of Figure 4.1 are key components of the design of any knowledge-based program. To see why, we need to understand the role that the internal representation of a fact plays in a program. What an AI program does is to manipulate the internal representations of the facts it is given. This manipulation should result in new structures that can also be interpreted as internal representations of facts. More precisely, these structures should be the internal representations of facts that correspond to the answer to the problem described by the starting set of facts.

Sometimes, a good representation makes the operation of a reasoning program not only correct but trivial. A well-known example of this occurs in the context of the mutilated checkerboard problem, which can be stated as follows:

The Mutilated Checkerboard Problem. Consider a normal checker board from which two squares, in opposite corners, have been removed. The task is to cover all the remaining squares exactly with dominoes, each of which covers two squares. No overlapping, either of dominoes on top of each other or of dominoes over the boundary of the mutilated board are allowed. Can this task be done?

One way to solve this problem is to try to enumerate, exhaustively, all possible tilings to see if one works. But suppose one wants to be more clever. Figure 4.2 shows three ways in which the mutilated checkerboard could be represented (to a person). The first



Number of
black squares = 30

Number of
white squares = 32

(c)

Figure 4.2: Three Representations of a Mutilated Checkerboard

representation does not directly suggest the answer to the problem. The second may; the third does, when combined with the single additional fact that each domino must cover exactly one white square and one black square. Even for human problem solvers a representation shift may make an enormous difference in problem-solving effectiveness. Recall that we saw a slightly less dramatic version of this phenomenon with respect to a problem-solving program in Section 1.3.1, where we considered two different ways of representing a tic-tac-toe board, one of which was as a magic square.

Figure 4.3 shows an expanded view of the starred part of Figure 4.1. The dotted line across the top represents the abstract reasoning process that a program is intended to model. The solid line across the bottom represents the concrete reasoning process that a particular program performs. This program successfully models the abstract process to the extent that, when the backward representation mapping is applied to the program's output, the appropriate final facts are actually generated. If either the program's operation or one of the representation mappings is not faithful to the problem that is being modeled, then the final facts will probably not be the desired ones. The key role that is played by the nature of the representation mapping is apparent from this figure. If no good mapping can be defined for a problem, then no matter how good the program to solve the problem is, it will not be able to produce answers that correspond to real answers to the problem.

It is interesting to note that Figure 4.3 looks very much like the sort of figure that might appear in a general programming book as a description of the relationship between an abstract data type (such as a set) and a concrete implementation of that type (e.g., as a linked list of elements). There are some differences, though, between this figure and the formulation usually used in programming texts (such as Aho *et al.* [1983]). For example, in data type design it is expected that the mapping that we are calling the backward representation mapping is a function (i.e., every representation corresponds to only one fact) and that it is onto (i.e., there is at least one representation for every fact). Unfortunately, in many AI domains, it may not be possible to come up with such a representation mapping, and we may have to live with one that gives less ideal results. But the main idea of what we are doing is the same as what programmers always do, namely to find concrete implementations of abstract concepts.

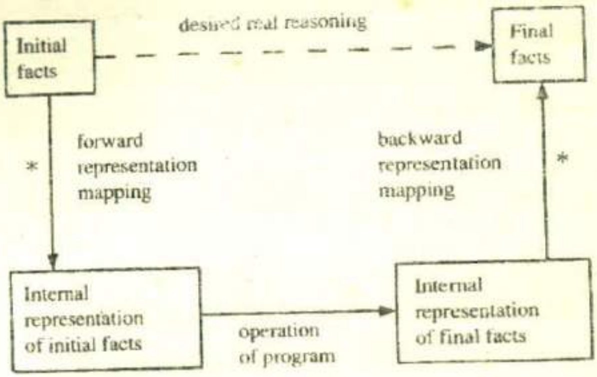

Initial facts
desired real reasoning
Final facts
forward representation mapping
backward representation mapping
*
*
Internal representation of initial facts
operation of program
Internal representation of final facts

Figure 4.3: Representation of Facts

4.2 Approaches to Knowledge Representation

A good system for the representation of knowledge in a particular domain should possess the following four properties:

Representational Adequacy —the ability to represent all of the kinds of knowledge that are needed in that domain.

Inferential Adequacy—the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.

Inferential Efficiency—the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.

Acquisitional Efficiency—the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

Unfortunately, no single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist. Many programs rely on more than one technique. In the chapters that follow, the most important of these techniques are described in detail. But in this section, we provide a simple, example-based introduction to the important ideas.

Simple Relational Knowledge

The simplest way to represent declarative facts is as a set of relations of the same sort used in database systems. Figure 4.4 shows an example of such a relational system.

Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right

Figure 4.4: Simple Relational Knowledge

The reason that this representation is simple is that standing alone it provides very weak inferential capabilities But knowledge represented in this form may serve as the input to more powerful inference engines. For example, given just the facts of Figure 4.4, it is not possible even to answer the simple question, "Who is the heaviest player?" But if a procedure for finding the heaviest player is provided, then these facts will enable the procedure to compute an answer. If, instead, we are provided with a set of rules for deciding which hitter to put up against a given pitcher (based on right- and left-handedness, say), then this same relation can provide at least some of the information required by those rules.

Providing support for relational knowledge is what database systems are designed to do. Thus we do not need to discuss this kind of knowledge representation structure further here. The practical issues that arise in linking a database system that provides this kind of support to a knowledge representation system that provides some of the other capabilities that we are about to discuss have already been solved in several commercial products.

Inheritable Knowledge

The relational knowledge of Figure 4.4 corresponds to a set of attributes and associated values that together describe the objects of the knowledge base. Knowledge about objects, their attributes, and their values need not be as simple as that shown in our example. In particular, it is possible to augment the basic representation with inference mechanisms that operate on the structure of the representation. For this to be effective, the structure must be designed to correspond to the inference mechanisms that are desired. One of the most useful forms of inference is *property inheritance*, in which elements of specific classes inherit attributes and values from more general classes in which they are included.

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy. Figure 4.5 shows some additional baseball knowledge inserted into a structure that is so arranged. Lines represent attributes. Boxed nodes represent objects and values of attributes of objects. These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line. The structure shown in the figure is a *slot-and-filler structure*. It may also be called a *semantic network* or a collection of *frames*. In the latter case each individual frame represents the collection of attributes and values associated with a particular node. Figure 4.6 shows the node for baseball player displayed as a frame.

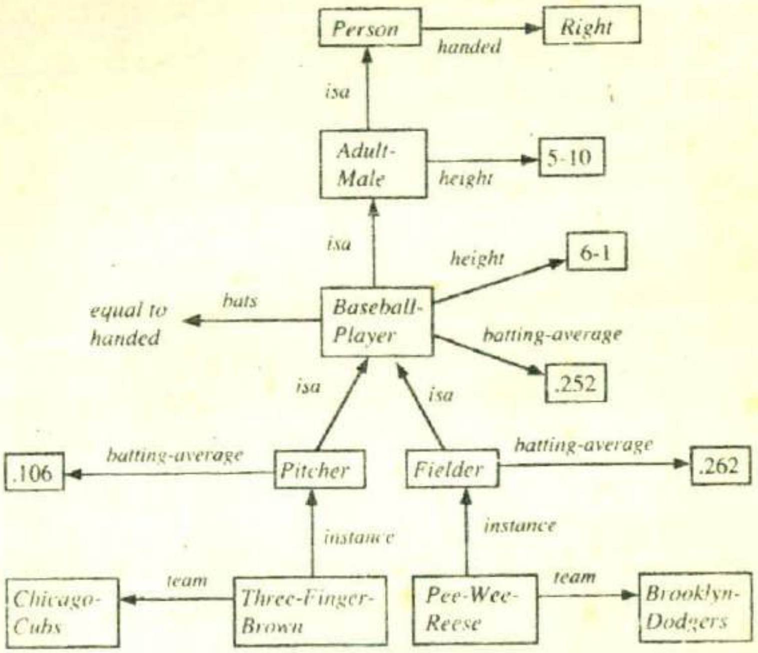


Figure 4.5: Inheritable Knowledge

Do not be put off by the confusion in terminology here. There is so much flexibility in the way that this (and the other structures described in this section) can be used to solve particular representation problems that it is difficult to reserve precise words for particular representations. Usually the use of the term *frame system* implies somewhat more structure on the attributes and the inference mechanisms that are available to apply to them than does the term *semantic network*.

In Chapter 9 we discuss structures such as these in substantial detail. But to get an idea of how these structures support inference using the knowledge they contain, we discuss them briefly here. All of the objects and most of the attributes shown in this example have been chosen to correspond to the baseball domain, and they have no general significance. The two exceptions to this are the attribute *isa*, which is being used to show class inclusion, and the attribute *instance*, which is being used to show class membership. These two specific (and generally useful) attributes provide the basis for property inheritance as an inference technique. Using this technique, the knowledge base can support retrieval both of facts that have been explicitly stored and of facts that can be derived from those that are explicitly stored.

An idealized form of the property inheritance algorithm can be stated as follows:

Baseball-Player

isa :	*Adult-Male*
bats :	(EQUAL *handed*)
height :	6-1
batting-average :	.252

Figure 4.6: Viewing a Node as a Frame

Algorithm: Property Inheritance

To retrieve a value *V* for attribute *A* of an instance object *O*:

1. Find *O* in the knowledge base.
2. If there is a value there for the attribute *A*, report that value.
3. Otherwise, see if there is a value for the attribute *instance*. If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute *A*. If one is found, report it.
5. Otherwise, do until there is no value for the *isa* attribute or until an answer is found:
   (a) Get the value of the *isa* attribute and move to that node.
   (b) See if there is a value for the attribute *A*. If there is, report it.

This procedure is simplistic. It does not say what we should do if there is more than one value of the *instance* or *isa* attribute. But it does describe the basic mechanism of inheritance. We can apply this procedure to our example knowledge base to derive answers to the following queries:

- *team*(*Pee-Wee-Reese*) = *Brooklyn-Dodgers*. This attribute had a value stored explicitly in the knowledge base.
- *batting-average*(*Three-Finger-Brown*) = .106. Since there is no value for batting average stored explicitly for Three Finger Brown, we follow the *instance* attribute to *Pitcher* and extract the value stored there. Now we observe one of the critical characteristics of property inheritance, namely that it may produce default values that are not guaranteed to be correct but that represent "best guesses" in the face of a lack of more precise information. In fact, in 1906, Brown's batting average was .204.
- *height*(*Pee-Wee-Reese*) = 6-1. This represents another default inference. Notice here that because we get to it first, the more specific fact about the height of baseball players overrides a more general fact about the height of adult males.

$$\begin{aligned}
&\forall x : \mathit{Ball}(x) \wedge \mathit{Fly}(x) \wedge \mathit{Fair}(x) \wedge \mathit{Infield\text{-}Catchable}(x) \wedge \\
&\quad \mathit{Occupied\text{-}Base}(\mathit{First}) \wedge \mathit{Occupied\text{-}Base}(\mathit{Second}) \wedge (\mathit{Outs} < 2) \wedge \\
&\quad \neg[\mathit{Line\text{-}Drive}(x) \vee \mathit{Attempted\text{-}B}\text{[illegible]}(x)] \\
&\rightarrow \mathit{Infield\text{-}Fly}(x)
\end{aligned}$$

$$\forall x, y : \mathit{Batter}(x) \wedge \mathit{batted}(x, y) \wedge \mathit{Infield\text{-}Fly}(y) \rightarrow \mathit{Out}(x)$$

Figure 4.7: Inferential Knowledge

- *bats(Three-Finger-Brown) = Right.* To get a value for the attribute *bats* required going up the *isa* hierarchy to the class *Baseball-Player*. But what we found there was not a value but a rule for computing a value. This rule required another value (that for *handed*) as input. So the entire process must be begun again recursively to find a value for *handed*. This time, it is necessary to go all the way up to *Person* to discover that the default value for handedness for people is *Right*. Now the rule for *bats* can be applied, producing the result *Right*. In this case, that turns out to be wrong, since Brown is a switch hitter (i.e., he can hit both left- and right-handed).

Inferential Knowledge

Property inheritance is a powerful form of inference, but it is not the only useful form. Sometimes all the power of traditional logic (and sometimes even more than that) is necessary to describe the inferences that are needed. Figure 4.7 shows two examples of the use of first-order predicate logic to represent additional knowledge about baseball.

Of course, this knowledge is useless unless there is also an inference procedure that can exploit it (just as the default knowledge in the previous example would have been useless without our algorithm for moving through the knowledge structure). The required inference procedure now is one that implements the standard logical rules of inference. There are many such procedures, some of which reason forward from given facts to conclusions, others of which reason backward from desired conclusions to given facts. One of the most commonly used of these procedures is *resolution*, which exploits a proof by contradiction strategy. Resolution is described in detail in Chapter 5.

Recall that we hinted at the need for something besides stored primitive values with the *bats* attribute of our previous example. Logic provides a powerful structure in which to describe relationships among values. It is often useful to combine this, or some other powerful description language, with an *isa* hierarchy. In general, in fact, all of the techniques we are describing here should not be regarded as complete and incompatible ways of representing knowledge. Instead, they should be viewed as building blocks of a complete representational system.

Procedural Knowledge

So far, our examples of baseball knowledge have concentrated on relatively static, declarative facts. But another, equally useful, kind of knowledge is operational, or procedural knowledge, that specifies what to do when. Procedural knowledge can be

```
Baseball-Player
    isa :              Adult-Male
    bats :             (lambda (x)
                         (prog ()
                                L1
                            (cond ((caddr x) (return (caddr x)))
                                  (t (setq x (eval (cadr x)))
                                     (cond (x (go L1))
                                           (t (return nil)))))))
    height :           6-1
    batting-average :  .252
```

Figure 4.8: Using LISP Code to Define a Value

represented in programs in many ways. The most common way is simply as code (in some programming language such as LISP) for doing something. The machine uses the knowledge when it executes the code to perform a task. Unfortunately, this way of representing procedural knowledge gets low scores with respect to the properties of inferential adequacy (because it is very difficult to write a program that can reason about another program's behavior) and acquisitional efficiency (because the process of updating and debugging large pieces of code becomes unwieldy).

As an extreme example, compare the representation of the way to compute the value of *bats* shown in Figure 4.6 to one in LISP shown in Figure 4.8. Although the LISP one will work given a particular way of storing attributes and values in a list, it does not lend itself to being reasoned about in the same straightforward way as the representation of Figure 4.6 does. The LISP representation is slightly more powerful since it makes explicit use of the name of the node whose value for *handed* is to be found. But if this matters, the simpler representation can be augmented to do this as well.

Because of this difficulty in reasoning with LISP, attempts have been made to find other ways of representing procedural knowledge so that it can relatively easily be manipulated both by other programs and by people.

The most commonly used technique for representing procedural knowledge in AI programs is the use of production rules. Figure 4.9 shows an example of a production rule that represents a piece of operational knowledge typically possessed by a baseball player.

Production rules, particularly ones that are augmented with information on how they are to be used, are more procedural than are the other representation methods discussed in this chapter. But making a clean distinction between declarative and procedural knowledge is difficult. Although at an intuitive level such a distinction makes some sense, at a formal level it disappears, as discussed in Section 6.1. In fact, as you can see, the structure of the declarative knowledge of Figure 4.7 is not substantially different from that of the operational knowledge of Figure 4.9. The important difference is in how the knowledge is used by the procedures that manipulate it.

```
If:     ninth inning, and
        score is close, and
        less than 2 outs, and
        first base is vacant, and
        batter is better hitter than next batter,
Then:   walk the batter.
```

Figure 4.9: Procedural Knowledge as Rules

4.3 Issues in Knowledge Representation

Before embarking on a discussion of specific mechanisms that have been used to represent various kinds of real-world knowledge, we need briefly to discuss several issues that cut across all of them:

- Are any attributes of objects so basic that they occur in almost every problem domain? If there are, we need to make sure that they are handled appropriately in each of the mechanisms we propose. If such attributes exist, what are they?
- Are there any important relationships that exist among attributes of objects?
- At what level should knowledge be represented? Is there a good set of *primitives* into which all knowledge can be broken down? Is it helpful to use such primitives?
- How should sets of objects be represented?
- Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?

We will talk about each of these questions briefly in the next five sections.

4.3.1 Important Attributes

There are two attributes that are of very general significance, and we have already seen their use: *instance* and *isa*. These attributes are important because they support property inheritance. They are called a variety of things in AI systems, but the names do not matter. What does matter is that they represent class membership and class inclusion and that class inclusion is transitive. In slot-and-filler systems, such as those described in Chapters 9 and 10, these attributes are usually represented explicitly in a way much like that shown in Figures 4.5 and 4.6. In logic-based systems, these relationships may be represented this way or they may be represented implicitly by a set of predicates describing particular classes. See Section 5.2 for some examples of this.

4.3.2 Relationships among Attributes

The attributes that we use to describe objects are themselves entities that we represent. What properties do they have independent of the specific knowledge they encode? There are four such properties that deserve mention here.

- Inverses
- Existence in an *isa* hierarchy
- Techniques for reasoning about values
- Single-valued attributes

Inverses

Entities in the world are related to each other in many different ways. But as soon as we decide to describe those relationships as attributes, we commit to a perspective in which we focus on one object and look for binary relationships between it and others. Attributes are those relationships. So, for example, in Figure 4.5, we used the attributes *instance*, *isa*, and *team*. Each of these was shown in the figure with a directed arrow, originating at the object that was being described and terminating at the object representing the value of the specified attribute. But we could equally well have focused on the object representing the value. If we do that, then there is still a relationship between the two entities, although it is a different one since the original relationship was not symmetric (although some relationships, such as *sibling*, are). In many cases, it is important to represent this other view of relationships. There are two good ways to do this.

The first is to represent both relationships in a single representation that ignores focus. Logical representations are usually interpreted as doing this. For example, the assertion:

team(*Pee-Wee-Reese*, *Brooklyn-Dodgers*)

can equally easily be interpreted as a statement about Pee Wee Reese or about the Brooklyn Dodgers. How it is actually used depends on the other assertions that a system contains.

The second approach is to use attributes that focus on a single entity but to use them in pairs, one the inverse of the other. In this approach, we would represent the team information with two attributes:

- one associated with Pee Wee Reese:

  team = *Brooklyn-Dodgers*

- one associated with Brooklyn Dodgers:

  team-members = *Pee-Wee-Reese*, . . .

This is the approach that is taken in semantic net and frame-based systems. When it is used, it is usually accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slots by forcing them to be declared and then checking each time a value is added to one attribute that the corresponding value is added to the inverse.

An Isa Hierarchy of Attributes

Just as there are classes of objects and specialized subsets of those classes, there are attributes and specializations of attributes. Consider, for example, the attribute *height*. It is actually a specialization of the more general attribute *physical-size* which is, in turn, a specialization of *physical-attribute*. These generalization-specialization relationships are important for attributes for the same reason that they are important for other concepts—they support inheritance. In the case of attributes, they support inheriting information about such things as constraints on the values that the attribute can have and mechanisms for computing those values.

Techniques for Reasoning about Values

Sometimes values of attributes are specified explicitly when a knowledge base is created. We saw several examples of that in the baseball example of Figure 4.5. But often the reasoning system must reason about values it has not been given explicitly. Several kinds of information can play a role in this reasoning, including:

- Information about the type of the value. For example, the value of *height* must be a number measured in a unit of length.

- Constraints on the value, often stated in terms of related entities. For example, the age of a person cannot be greater than the age of either of that person's parents.

- Rules for computing the value when it is needed. We showed an example of such a rule in Figure 4.5 for the *bats* attribute. These rules are called *backward* rules. Such rules have also been called *if-needed rules*.

- Rules that describe actions that should be taken if a value ever becomes known. These rules are called *forward* rules, or sometimes *if-added rules*.

We discuss forward and backward rules again in Chapter 6, in the context of rule-based knowledge representation.

Single-Valued Attributes

A specific but very useful kind of attribute is one that is guaranteed to take a unique value. For example, a baseball player can, at any one time, have only a single height and be a member of only one team. If there is already a value present for one of these attributes and a different value is asserted, then one of two things has happened. Either a change has occurred in the world or there is now a contradiction in the knowledge base that needs to be resolved. Knowledge-representation systems have taken several different approaches to providing support for single-valued attributes, including:

- Introduce an explicit notation for temporal interval. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.

- Assume that the only temporal interval that is of interest is now. So if a new value is asserted, replace the old value.

- Provide no explicit support. Logic-based systems are in this category. But in these systems, knowledge-base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.

4.3.3 Choosing the Granularity of Representation

Regardless of the particular representation formalism we choose, it is necessary to answer the question "At what level of detail should the world be represented?" Another way this question is often phrased is "What should be our primitives?" Should there be a small number of low-level ones or should there be a larger number covering a range of granularities? A brief example illustrates the problem. Suppose we are interested in the following fact:

John spotted Sue.

We could represent this as[1]

spotted(agent(John),
object(Sue))

Such a representation would make it easy to answer questions such as:

Who spotted Sue?

But now suppose we want to know:

Did John see Sue?

The obvious answer is "yes," but given only the one fact we have, we cannot discover that answer. We could, of course, add other facts, such as

$spotted(x, y) \rightarrow saw(x, y)$

We could then infer the answer to the question.

An alternative solution to this problem is to represent the fact that spotting is really a special type of seeing explicitly in the representation of the fact. We might write something such as

saw(agent(John),
object(Sue),
timespan(briefly))

[1] The arguments *agent* and *object* are usually called *cases*. They represent roles involved in the event. This semantic way of analyzing sentences contrasts with the probably more familiar syntactic approach in which sentences have a surface subject, direct object, indirect object, and so forth. We will discuss case grammar [Fillmore, 1968] and its use in natural language understanding in Section 15.3.2. For the moment, you can safely assume that the cases mean what their names suggest.

In this representation, we have broken the idea of *spotting* apart into more primitive concepts of *seeing* and *timespan*. Using this representation, the fact that John saw Sue is immediately accessible. But the fact that he spotted her is more difficult to get to.

The major advantage of converting all statements into a representation in terms of a small set of primitives is that the rules that are used to derive inferences from that knowledge need be written only in terms of the primitives rather than in terms of the many ways in which the knowledge may originally have appeared. Thus what is really being argued for is simply some sort of canonical form. Several AI programs, including those described by Schank and Abelson [1977] and Wilks [1972], are based on knowledge bases described in terms of a small number of low-level primitives.

There are several arguments against the use of low-level primitives. One is that simple high-level facts may require a lot of storage when broken down into primitives. Much of that storage is really wasted since the low-level rendition of a particular high-level concept will appear many times, once for each time the high-level concept is referenced. For example, suppose that actions are being represented as combinations of a small set of primitive actions. Then the fact that John punched Mary might be represented as shown in Figure 4.10(*a*). The representation says that there was physical contact between John's fist and Mary. The contact was caused by John propelling his fist toward Mary, and in order to do that John first went to where Mary was.[2] But suppose we also know that Mary punched John. Then we must also store the structure shown in Figure 4.10(*b*). If, however, punching were represented simply as punching, then most of the detail of both structures could be omitted from the structures themselves. It could instead be stored just once in a common representation of the concept of punching.

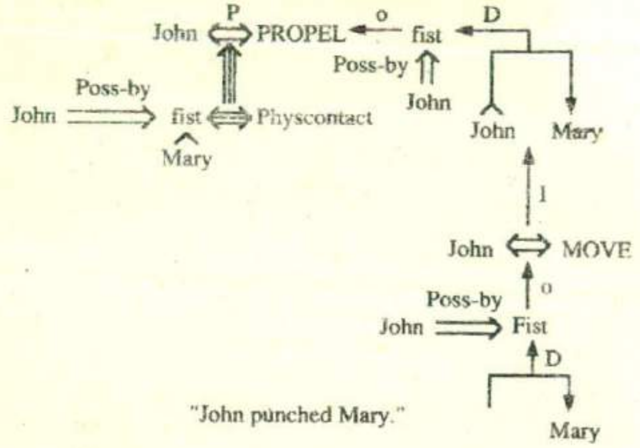
A second but related problem is that if knowledge is initially presented to the system in a relatively high-level form, such as English, then substantial work must be done to reduce the knowledge into primitive form. Yet, for many purposes, this detailed primitive representation may be unnecessary. Both in understanding language and in interpreting the world that we see, many things appear that later turn out to be irrelevant. For the sake of efficiency, it may be desirable to store these things at a very high level and then to analyze in detail only those inputs that appear to be important.

A third problem with the use of low-level primitives is that in many domains, it is not at all clear what the primitives should be. And even in domains in which there may be an obvious set of primitives, there may not be enough information present in each use of the high-level constructs to enable them to be converted into their primitive components. When this is true, there is no way to avoid representing facts at a variety of granularities.

The classical example of this sort of situation is provided by kinship terminology [Lindsay, 1963]. There exists at least one obvious set of primitives: mother, father, son, daughter, and possibly brother and sister. But now suppose we are told that Mary is Sue's cousin. An attempt to describe the cousin relationship in terms of the primitives could produce any of the following interpretations:

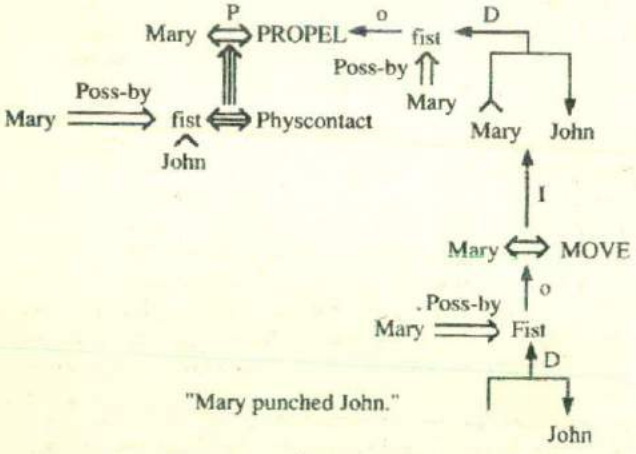
- Mary = *daughter*(*brother*(*mother*(Sue)))
- Mary = *daughter*(*sister*(*mother*(Sue)))

[2] The representation shown in this example is called *conceptual dependency* and is discussed in detail in Section 10.1.



"John punched Mary."

(*a*)



"Mary punched John."

(*b*)

Figure 4.10: Redundant Representations

- Mary = *daughter*(*brother*(*father*(Sue)))
- Mary = *daughter*(*sister*(*father*(Sue)))

If we do not already know that Mary is female, then of course there are four more possibilities as well. Since in general we may have no way of choosing among these representations, we have no choice but to represent the fact using the nonprimitive relation *cousin*.

The other way to solve this problem is to change our primitives. We could use the set: *parent*, *child*, *sibling*, *male*, and *female*. Then the fact that Mary is Sue's cousin could be represented as

- Mary = *child*(*sibling*(*parent*(Sue)))

But now the primitives incorporate some generalizations that may or may not be appropriate. The main point to be learned from this example is that even in very simple domains, the correct set of primitives is not obvious.

In less well-structured domains, even more problems arise. For example, given just the fact

John broke the window.

a program would not be able to decide if John's actions consisted of the primitive sequence:

1. Pick up a hard object.
2. Hurl the object through the window.

or the sequence:

1. Pick up a hard object.
2. Hold onto the object while causing it to crash into the window.

or the single action:

1. Cause hand (or foot) to move fast and crash into the window.

or the single action:

1. Shut the window so hard that the glass breaks.

As these examples have shown, the problem of choosing the correct granularity of representation for a particular body of knowledge is not easy. Clearly, the lower the level we choose, the less inference required to reason with it in some cases, but the more inference required to create the representation from English and the more room it takes to store, since many inferences will be represented many times. The answer for any particular task domain must come to a large extent from the domain itself—to what use is the knowledge to be put?

One way of looking at the question of whether there exists a good set of low-level primitives is that it is a question of the existence of a unique representation. Does there exist a single, canonical way in which large bodies of knowledge can be represented independently of how they were originally stated? Another, closely related, uniqueness question asks whether individual objects can be represented uniquely and independently of how they are described. This issue is raised in the following quotation from Quine [1961] and discussed in Woods [1975]:

> The phrase *Evening Star* names a certain large physical object of spherical form, which is hurtling through space some scores of millions of miles from here. The phrase *Morning Star* names the same thing, as was probably first established by some observant Babylonian. But the two phrases cannot be regarded as having the same meaning; otherwise that Babylonian could have dispensed with his observations and contented himself with reflecting on the meaning of his words. The meanings, then, being different from one another, must be other than the named object, which is one and the same in both cases.

In order for a program to be able to reason as did the Babylonian, it must be able to handle several distinct representations that turn out to stand for the same object.

We discuss the question of the correct granularity of representation, as well as issues involving redundant storage of information, throughout the next several chapters, particularly in the section on conceptual dependency, since that theory explicitly proposes that a small set of low-level primitives should be used for representing actions.

4.3.4 Representing Sets of Objects

It is important to be able to represent sets of objects for several reasons. One is that there are some properties that are true of sets that are not true of the individual members of a set. As examples, consider the assertions that are being made in the sentences "There are more sheep than people in Australia" and "English speakers can be found all over the world." The only way to represent the facts described in these sentences is to attach assertions to the sets representing people, sheep, and English speakers, since, for example, no single English speaker can be found all over the world. The other reason that it is important to be able to represent sets of objects is that if a property is true of all (or even most) elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every element of the set. We have already looked at ways of doing that, both in logical representations through the use of the universal quantifier and in slot-and-filler structures, where we used nodes to represent sets and inheritance to propagate set-level assertions down to individuals. As we consider ways to represent sets, we will want to consider both of these uses of set-level representations. We will also need to remember that the two uses must be kept distinct. Thus if we assert something like *large*(*Elephant*), it must be clear whether we are asserting some property of the set itself (i.e., that the set of elephants is large) or some property that holds for individual elements of the set (i.e., that anything that is an elephant is large).

There are three obvious ways in which sets may be represented. The simplest is just by a name. This is essentially what we did in Section 4.2 when we used the node named *Baseball-Player* in our semantic net and when we used predicates such as *Ball* and

Batter in our logical representation. This simple representation does make it possible to associate predicates with sets. But it does not, by itself, provide any information about the set it represents. It does not, for example, tell how to determine whether a particular object is a member of the set or not.

There are two ways to state a definition of a set and its elements. The first is to list the members. Such a specification is called an *extensional* definition. The second is to provide a rule that, when a particular object is evaluated, returns true or false depending on whether the object is in the set or not. Such a rule is called an *intensional* definition. For example, an extensional description of the set of our sun's planets on which people live is $\{Earth\}$. An intensional description is

$$\{x : sun\text{-}planet(x) \land human\text{-}inhabited(x)\}$$

For simple sets, it may not matter, except possibly with respect to efficiency concerns, which representation is used. But the two kinds of representations can function differently in some cases.

One way in which extensional and intensional representations differ is that they do not necessarily correspond one-to-one with each other. For example, the extensionally defined set $\{Earth\}$ has many intensional definitions in addition to the one we just gave. Others include:

$$\{x : sun\text{-}planet(x) \land nth\text{-}farthest\text{-}from\text{-}sun(x, 3)\}$$
$$\{x : sun\text{-}planet(x) \land nth\text{-}biggest(x, 5)\}$$

Thus, while it is trivial to determine whether two sets are identical if extensional descriptions are used, it may be very difficult to do so using intensional descriptions.

Intensional representations have two important properties that extensional ones lack, however. The first is that they can be used to describe infinite sets and sets not all of whose elements are explicitly known. Thus we can describe intensionally such sets as prime numbers (of which there are infinitely many) or kings of England (even though we do not know who all of them are or even how many of them there have been). The second thing we can do with intensional descriptions is to allow them to depend on parameters that can change, such as time or spatial location. If we do that, then the actual set that is represented by the description will change as a function of the value of those parameters. To see the effect of this, consider the sentence, "The president of the United States used to be a Democrat," uttered when the current president is a Republican. This sentence can mean two things. The first is that the specific person who is now president was once a Democrat. This meaning can be captured straightforwardly with an extensional representation of "the president of the United States." We just specify the individual. But there is a second meaning, namely that there was once someone who was the president and who was a Democrat. To represent the meaning of "the president of the United States" given this interpretation requires an intensional description that depends on time. Thus we might write *president(t)*, where *president* is some function that maps instances of time onto instances of people, namely U.S. presidents.

4.3.5 Finding the Right Structures as Needed

Recall that in Chapter 2, we briefly touched on the problem of matching rules against state descriptions during the problem-solving process. This same issue now rears its head with respect to locating appropriate knowledge structures that have been stored in memory.

For example, suppose we have a script (a description of a class of events in terms of contexts, participants, and subevents) that describes the typical sequence of events in a restaurant.[3] This script would enable us to take a text such as

> John went to Steak and Ale last night. He ordered a large rare steak, paid his bill, and left.

and answer "yes" to the question

> Did John eat dinner last night?

Notice that nowhere in the story was John's eating anything mentioned explicitly. But the fact that when one goes to a restaurant one eats will be contained in the restaurant script. If we know in advance to use the restaurant script, then we can answer the question easily. But in order to be able to reason about a variety of things, a system must have many scripts for everything from going to work to sailing around the world. How will it select the appropriate one each time? For example, nowhere in our story was the word "restaurant" mentioned.

In fact, in order to have access to the right structure for describing a particular situation, it is necessary to solve all of the following problems.[4].

- How to perform an initial selection of the most appropriate structure.
- How to fill in appropriate details from the current situation.
- How to find a better structure if the one chosen initially turns out not to be appropriate.
- What to do if none of the available structures is appropriate.
- When to create and remember a new structure.

There is no good, general purpose method for solving all these problems. Some knowledge-representation techniques solve some of them. In this section we survey some solutions to two of these problems: how to select an initial structure to consider and how to find a better structure if that one turns out not to be a good match.

Selecting an Initial Structure

Selecting candidate knowledge structures to match a particular problem-solving situation is a hard problem; there are several ways in which it can be done. Three important approaches are the following:

[3] We discuss such a script in detail in Chapter 10.

[4] This list is taken from Minsky [1975].

- Index the structures directly by the significant English words that can be used to describe them. For example, let each verb have associated with it a structure that describes its meaning. This is the approach taken in conceptual dependency theory, discussed in Chapter 10. Even for selecting simple structures, such as those representing the meanings of individual words, though, this approach may not be adequate, since many words may have several distinct meanings. For example, the word "fly" has a different meaning in each of the following sentences:

  - John flew to New York. (He rode in a plane from one place to another.)
  - John flew a kite. (He held a kite that was up in the air.)
  - John flew down the street. (He moved very rapidly.)
  - John flew into a rage. (An idiom)

  Another problem with this approach is that it is only useful when there is an English description of the problem to be solved.

- Consider each major concept as a pointer to all of the structures (such as scripts) in which it might be involved. This may produce several sets of prospective structures. For example, the concept *Steak* might point to two scripts, one for restaurant and one for supermarket. The concept *Bill* might point to a restaurant and a shopping script. Take the intersection of those sets to get the structure(s), preferably precisely one, that involves all the content words. Given the pointers just described and the story about John's trip to Steak and Ale, the restaurant script would be evoked. One important problem with this method is that if the problem description contains any even slightly extraneous concepts, then the intersection of their associated structures will be empty. This might occur if we had said, for example, "John rode his bicycle to Steak and Ale last night." Another problem is that it may require a great deal of computation to compute all of the possibility sets and then to intersect them. However, if computing such sets and intersecting them could be done in parallel, then the time required to produce an answer would be reasonable even if the total number of computations is large. For an exploration of this parallel approach to clue intersection, see Fahlman [1979].

- Locate one major clue in the problem description and use it to select an initial structure. As other clues appear, use them to refine the initial selection or to make a completely new one if necessary. For a discussion of this approach, see Charniak [1978]. The major problem with this method is that in some situations there is not an easily identifiable major clue. A second problem is that it is necessary to anticipate which clues are going to be important and which are not. But the relative importance of clues can change dramatically from one situation to another. For example, in many contexts, the color of the objects involved is not important. But if we are told "The light turned red," then the color of the light is the most important feature to consider.

None of these proposals seems to be the complete answer to the problem. It often turns out, unfortunately, that the more complex the knowledge structures are, the harder it is to tell when a particular one is appropriate.

Revising the Choice When Necessary

Once we find a candidate knowledge structure, we must attempt to do a detailed match of it to the problem at hand. Depending on the representation we are using, the details of the matching process will vary. It may require variables to be bound to objects. It may require attributes to have their values compared. In any case, if values that satisfy the required restrictions as imposed by the knowledge structure can be found, they are put into the appropriate places in the structure. If no appropriate values can be found, then a new structure must be selected. The way in which the attempt to instantiate this first structure failed may provide useful cues as to which one to try next. If, on the other hand, appropriate values can be found, then the current structure can be taken to be appropriate for describing the current situation. But, of course, that situation may change. Then information about what happened (for example, we walked around the room we were looking at) may be useful in selecting a new structure to describe the revised situation.

As was suggested above, the process of instantiating a structure in a particular situation often does not proceed smoothly. When the process runs into a snag, though, it is often not necessary to abandon the effort and start over. Rather, there are a variety of things that can be done.

- Select the fragments of the current structure that do correspond to the situation and match them against candidate alternatives. Choose the best match. If the current structure was at all close to being appropriate, much of the work that has been done to build substructures to fit into it will be preserved.

- Make an excuse for the current structure's failure and continue to use it. For example, a proposed chair with only three legs might simply be broken. Or there might be another object in front of it which occludes one leg. Part of the structure should contain information about the features for which it is acceptable to make excuses. Also, there are general heuristics, such as the fact that a structure is more likely to be appropriate if a desired feature is missing (perhaps because it is hidden from view) than if an inappropriate feature is present. For example, a person with one leg is more plausible than a person with a tail.

- Refer to specific stored links between structures to suggest new directions in which to explore. An example of this sort of linking among a set of frames is shown in the similarity network shown in Figure 4.11.[5]

- If the knowledge structures are stored in an *isa* hierarchy, traverse upward in it until a structure is found that is sufficiently general that it does not conflict with the evidence. Either use this structure if it is specific enough to provide the required knowledge or consider creating a new structure just below the matching one.

4.4 The Frame Problem

So far in this chapter, we have seen several methods for representing knowledge that would allow us to form complex state descriptions for a search program. Another issue

[5]This example is taken from Minsky [1975].

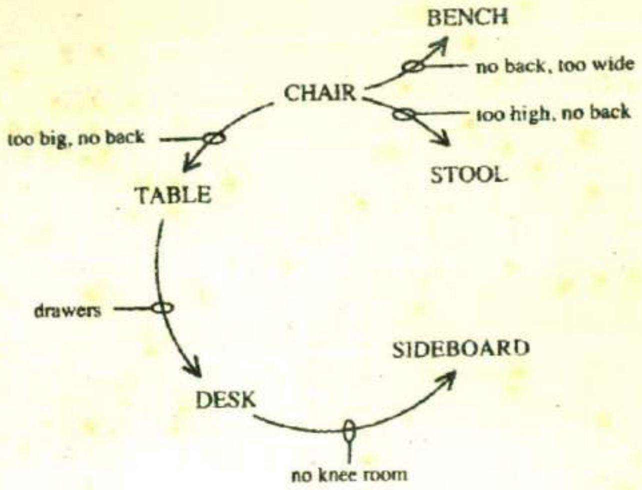


Figure 4.11: A Similarity Net

concerns how to represent efficiently *sequences* of problem states that arise from a search process. For complex ill-structured problems, this can be a serious matter.

Consider the world of a household robot. There are many objects and relationships in the world, and a state description must somehow include facts like *on(Plant12, Table34)*, *under(Table34, Window13)*, and *in(Table34, Room15)*. One strategy is to store each state description as a list of such facts. But what happens during the problem-solving process if each of those descriptions is very long? Most of the facts will not change from one state to another, yet each fact will be represented once at every node, and we will quickly run out of memory. Furthermore, we will spend the majority of our time creating these nodes and copying these facts—most of which do not change often—from one node to another. For example, in the robot world, we could spend a lot of time recording *above(Ceiling, Floor)* at every node. All of this is, of course, in addition to the real problem of figuring out which facts *should* be different at each node.

This whole problem of representing the facts that change as well as those that do not is known as the *frame problem* [McCarthy and Hayes, 1969]. In some domains, the only hard part is representing all the facts. In others, though, figuring out which ones change is nontrivial. For example, in the robot world, there might be a table with a plant on it under the window. Suppose we move the table to the center of the room. We must also infer that the plant is now in the center of the room too but that the window is not.

To support this kind of reasoning, some systems make use of an explicit set of axioms called *frame axioms*, which describe all the things that do not change when a particular operator is applied in state n to produce state $n + 1$. (The things that do change must be mentioned as part of the operator itself.) Thus, in the robot domain, we might write axioms such as

$$color(x, y, s_1) \wedge move(x, s_1, s_2) \rightarrow color(x, y, s_2)$$

which can be read as, "If x has color y in state s_1 and the operation of moving x is applied in state s1 to produce state s_2, then the color of x in s_2 is still y." Unfortunately, in any complex domain, a huge number of these axioms becomes necessary. An alternative approach is to make the assumption that the only things that change are the things that must. By "must" here we mean that the change is either required explicitly by the axioms that describe the operator or that it follows logically from some change that is asserted explicitly. This idea of *circumscribing* the set of unusual things is a very powerful one; it can be used as a partial solution to the frame problem and as a way of reasoning with incomplete knowledge. We return to it in Chapter 7.

But now let's return briefly to the problem of representing a changing problem state. We could do it by simply starting with a description of the initial state and then making changes to that description as indicated by the rules we apply. This solves the problem of the wasted space and time involved in copying the information for each node. And it works fine until the first time the search has to backtrack. Then, unless all the changes that were made can simply be ignored (as they could be if, for example, they were simply additions of new theorems), we are faced with the problem of backing up to some earlier node. But how do we know what changes in the problem state description need to be undone? For example, what do we have to change to undo the effect of moving the table to the center of the room? There are two ways this problem can be solved:

- Do not modify the initial state description at all. At each node, store an indication of the specific changes that should be made at this node. Whenever it is necessary to refer to the description of the current problem state, look at the initial state description and also look back through all the nodes on the path from the start state to the current state. This is what we did in our solution to the cryptarithmetic problem in Section 3.5. This approach makes backtracking very easy, but it makes referring to the state description fairly complex.

- Modify the initial state description as appropriate, but also record at each node an indication of what to do to undo the move should it ever be necessary to backtrack through the node. Then, whenever it is necessary to backtrack, check each node along the way and perform the indicated operations on the state description.

Sometimes, even these solutions are not enough. We might want to remember, for example, in the robot world, that before the table was moved, it was under the window and after being moved, it was in the center of the room. This can be handled by adding to the representation of each fact a specific indication of the time at which that fact was true. This indication is called a *state variable*. But to apply the same technique to a real-world problem, we need, for example, separate facts to indicate all the times at which the Statue of Liberty is in New York.

There is no simple answer either to the question of knowledge representation or to the frame problem. Each of them is discussed in greater depth later in the context of specific problems. But it is important to keep these questions in mind when considering search strategies, since the representation of knowledge and the search process depend heavily on each other.

4.5 Summary

The purpose of this chapter has been to outline the need for knowledge in reasoning programs and to survey issues that must be addressed in the design of a good knowledge-representation structure. Of course, we have not covered everything. In the chapters that follow, we describe some specific representations and look at their relative strengths and weaknesses.

The following collections all contain further discussions of the fundamental issues in knowledge representation, along with specific techniques to address these issues: Bobrow [1975], Winograd [1978], Brachman and Levesque [1985], and Halpern [1986]. For especially clear discussions of specific issues on the topic of knowledge representation and use see Woods [1975] and Brachman [1985].

Chapter 5

Using Predicate Logic

In this chapter, we begin exploring one particular way of representing facts—the language of logic. Other representational formalisms are discussed in later chapters. The logical formalism is appealing because it immediately suggests a powerful way of deriving new knowledge from old—mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known. Thus the idea of a proof, as developed in mathematics as a rigorous way of demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answers to questions and solutions to problems.

One of the early domains in which AI techniques were explored was mechanical theorem proving, by which was meant proving statements in various areas of mathematics. For example, the Logic Theorist [Newell *et al.*, 1963] proved theorems from the first chapter of Whitehead and Russell's *Principia Mathematica* [1950]. Another theorem prover [Gelernter *et al.*, 1963] proved theorems in geometry. Mathematical theorem proving is still an active area of AI research. (See, for example, Wos *et al.* [1984].) But, as we show in this chapter, the usefulness of some mathematical techniques extends well beyond the traditional scope of mathematics. It turns out that mathematics is no different from any other complex intellectual endeavor in requiring both reliable deductive mechanisms and a mass of heuristic knowledge to control what would otherwise be a completely intractable search problem.

At this point, readers who are unfamiliar with propositional and predicate logic may want to consult a good introductory logic text before reading the rest of this chapter. Readers who want a more complete and formal presentation of the material in this chapter should consult Chang and Lee [1973]. Throughout the chapter, we use the following standard logic symbols: "→" (*material implication*), "¬" (*not*), "∨" (*or*), "∧" (*and*), "∀" (*for all*), and "∃" (*there exists*).

5.1 Representing Simple Facts in Logic

Let's first explore the use of propositional logic as a way of representing the sort of world knowledge that an AI system might need. Propositional logic is appealing because it is simple to deal with and a decision procedure for it exists. We can easily

It is raining.
RAINING

It is sunny.
SUNNY

It is windy.
WINDY

If it is raining, then it is not sunny.
$RAINING \rightarrow \neg SUNNY$

Figure 5.1: Some Simple Facts in Propositional Logic

represent real-world facts as logical *propositions* written as *well-formed formulas (wff's)* in propositional logic, as shown in Figure 5.1. Using these propositions, we could, for example, conclude from the fact that it is raining the fact that it is not sunny. But very quickly we run up against the limitations of propositional logic. Suppose we want to represent the obvious fact stated by the classical sentence

Socrates is a man.

We could write:

SOCRATESMAN

But if we also wanted to represent

Plato is a man.

we would have to write something such as:

PLATOMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between Socrates and Plato. It would be much better to represent these facts as:

MAN(SOCRATES)
MAN(PLATO)

since now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use predicates applied to arguments. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal.

We could represent this as:

MORTALMAN

But that fails to capture the relationship between any individual being a man and that individual being a mortal. To do that, we really need variables and quantification unless we are willing to write separate statements about the mortality of every known man.

So we appear to be forced to move to first-order predicate logic (or just predicate logic, since we do not discuss higher order theories in this chapter) as a way of representing knowledge because it permits representations of things that cannot reasonably be represented in propositional logic. In predicate logic, we can represent real-world facts as *statements* written as wff's.

But a major motivation for choosing to use logic at all is that if we use logical statements as a way of representing knowledge, then we have available a good way of reasoning with that knowledge. Determining the validity of a proposition in propositional logic is straightforward, although it may be computationally hard. So before we adopt predicate logic as a good medium for representing knowledge, we need to ask whether it also provides a good way of reasoning with the knowledge. At first glance, the answer is yes. It provides a way of deducing new statements from old ones. Unfortunately, however, unlike propositional logic, it does not possess a decision procedure, even an exponential one. There do exist procedures that will find a proof of a proposed theorem if indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. In other words, although first-order predicate logic is not decidable, it is semidecidable. A simple such procedure is to use the rules of inference to generate theorems from the axioms in some orderly fashion, testing each to see if it is the one for which a proof is sought. This method is not particularly efficient, however, and we will want to try to find a better one.

Although negative results, such as the fact that there can exist no decision procedure for predicate logic, generally have little direct effect on a science such as AI, which seeks positive methods for doing things, this particular negative result is helpful since it tells us that in our search for an efficient proof procedure, we should be content if we find one that will prove theorems, even if it is not guaranteed to halt if given a nontheorem. And the fact that there cannot exist a decision procedure that halts on all possible inputs does not mean that there cannot exist one that will halt on almost all the inputs it would see in the process of trying to solve real problems. So despite the theoretical undecidability of predicate logic, it can still serve as a useful way of representing and manipulating some of the kinds of knowledge that an AI system might need.

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

1. Marcus was a man.

   $man(Marcus)$

   This representation captures the critical fact of Marcus being a man. It fails to capture some of the information in the English sentence, namely the notion of past tense. Whether this omission is acceptable or not depends on the use to which we intend to put the knowledge. For this simple example, it will be all right.

2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. All Pompeians were Romans.

   $\forall x : Pompeian(x) \rightarrow Roman(x)$

4. Caesar was a ruler.

   $ruler(Caesar)$

   Here we ignore the fact that proper names are often not references to unique individuals, since many people share the same name. Sometimes deciding which of several people of the same name is being referred to in a particular statement may require a fair amount of knowledge and reasoning.

5. All Romans were either loyal to Caesar or hated him.

   $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

   In English, the word "or" sometimes means the logical *inclusive-or* and sometimes means the logical *exclusive-or* (XOR). Here we have used the inclusive interpretation. Some people will argue, however, that this English sentence is really stating an exclusive-or. To express that, we would have to write:

   $$\forall x : Roman(x) \rightarrow [(loyalto(x, Caesar) \vee hate(x, Caesar)) \wedge \\ \neg(loyalto(x, Caesar) \wedge hate(x, Caesar))]$$

6. Everyone is loyal to someone.

   $\forall x : \exists y : loyalto(x, y)$

   A major problem that arises when trying to convert English sentences into logical statements is the scope of quantifiers. Does this sentence say, as we have assumed in writing the logical formula above, that for each person there exists someone to whom he or she is loyal, possibly a different someone for everyone? Or does it say that there exists someone to whom everyone is loyal (which would be written as $\exists y : \forall x : loyalto(x, y)$)? Often only one of the two interpretations seems likely, so people tend to favor it.

7. People only try to assassinate rulers they are not loyal to.

   $\forall x : \forall y : person(x) \wedge ruler(y) \wedge tryassassinate(x, y) \rightarrow \neg loyalto(x, y)$

   This sentence, too, is ambiguous. Does it mean that the only rulers that people try to assassinate are those to whom they are not loyal (the interpretation used here), or does it mean that the only thing people try to do is to assassinate rulers to whom they are not loyal?

   In representing this sentence the way we did, we have chosen to write "try to assassinate" as a single predicate. This gives a fairly simple representation with which we can reason about trying to assassinate. But using this representation, the connections between trying to assassinate and trying to do other things and between trying to assassinate and actually assassinating could not be made easily. If such connections were necessary, we would need to choose a different representation.

8. Marcus tried to assassinate Caesar.

   $tryassassinate(Marcus, Caesar)$

From this brief attempt to convert English sentences into logical statements, it should be clear how difficult the task is. For a good description of many issues involved in this process, see Reichenbach [1947].

Now suppose that we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to Caesar (again ignoring the distinction between past and present tense). Now let's try to produce a formal proof, reasoning backward from the desired goal:

$\neg loyalto(Marcus, Caesar)$

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can in turn be transformed, and so on, until there are no unsatisfied goals remaining. This process may require the search of an AND-OR graph (as described in Section 3.4) when there are alternative ways of satisfying individual goals. Here, for simplicity, we show only a single path. Figure 5.2 shows an attempt to produce a proof of the goal by reducing the set of necessary but as yet unattained goals to the empty set. The attempt fails, however, since there is no way to satisfy the goal *person(Marcus)* with the statements we have available.

The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. We need to add the representation of another fact to our system, namely:

$\neg loyalto(Marcus, Caesar)$

$\uparrow$ (7, substitution)

$person(Marcus) \land$
$ruler(Caesar) \land$
$tryassassinate(Marcus, Caesar)$

$\uparrow$ (4)

$person(Marcus)$
$tryassassinate(Marcus, Caesar)$

$\uparrow$ (8)

$person(Marcus)$

Figure 5.2: An Attempt to Prove $\neg loyalto(Marcus, Caesar)$

9. All men are people.

   $\forall x : man(x) \rightarrow person(x)$

Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.

From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:

- Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.

- There is often a choice of how to represent the knowledge (as discussed in connection with 1 and 7 above). Simple representations are desirable, but they may preclude certain kinds of reasoning. The expedient representation for a particular set of sentences depends on the use to which the knowledge contained in the sentences will be put.

- Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively, it is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention. We discuss this issue further in Section 10.3.

An additional problem arises in situations where we do not know in advance which statements to deduce. In the example just presented, the object was to answer the question "Was Marcus loyal to Caesar?" How would a program decide whether it should try to prove

$loyalto(Marcus, Caesar)$

or

$\neg loyalto(Marcus, Caesar)$

There are several things it could do. It could abandon the strategy we have outlined of reasoning backward from a proposed truth to the axioms and instead try to reason forward and see which answer it gets to. The problem with this approach is that, in general, the branching factor going forward from the axioms is so great that it would probably not get to either answer in any reasonable amount of time. A second thing it could do is use some sort of heuristic rules for deciding which answer is more likely and then try to prove that one first. If it fails to find a proof after some reasonable amount of effort, it can try the other answer. This notion of limited effort is important, since any proof procedure we use may not halt if given a nontheorem. Another thing it could do is simply try to prove both answers simultaneously and stop when one effort is successful. Even here, however, if there is not enough information available to answer the question with certainty, the program may never halt. Yet a fourth strategy is to try both to prove one answer and to disprove it, and to use information gained in one of the processes to guide the other.

5.2 Representing Instance and Isa Relationships

In Chapter 4, we discussed the specific attributes *instance* and *isa* and described the important role they play in a particularly useful form of reasoning, property inheritance. But if we look back at the way we just represented our knowledge about Marcus and Caesar, we do not appear to have used these attributes at all. We certainly have not used predicates with those names. Why not? The answer is that although we have not used the predicates *instance* and *isa* explicitly, we have captured the relationships they are used to express, namely class membership and class inclusion.

Figure 5.3 shows the first five sentences of the last section represented in logic in three different ways. The first part of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as *Roman*), each of which corresponds to a class. Asserting that $P(x)$ is true is equivalent to asserting that x is an instance (or element) of P. The second part of the figure contains representations that use the *instance* predicate explicitly. The predicate *instance* is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit *isa* predicate. Instead, subclass relationships, such as that between Pompeians and Romans, are described as shown in sentence 3. The implication rule there states that if an object is an instance of the subclass *Pompeian* then it is an instance of the superclass *Roman*. Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship. The third part contains representations that use both the *instance* and *isa* predicates explicitly. The use of the *isa* predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided. This additional axiom describes how an *instance* relation and an *isa* relation can be combined to derive a new *instance* relation. This one additional axiom is general, though, and does not need to be provided separately for additional *isa* relations.

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\forall x : Pompeian(x) \rightarrow Roman(x)$
4. $ruler(Caesar)$
5. $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $\forall x : instance(x, Pompeian) \rightarrow instance(x, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $isa(Pompeian, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$
6. $\forall x : \forall y : \forall z : instance(x, y) \wedge isa(y, z) \rightarrow instance(x, z)$

Figure 5.3: Three Ways of Representing Class Membership

These examples illustrate two points. The first is fairly specific. It is that, although class and superclass memberships are important facts that need to be represented, those memberships need not be represented with predicates labeled *instance* and *isa*. In fact, in a logical framework it is usually unwieldy to do that, and instead unary predicates corresponding to the classes are often used. The second point is more general. There are usually several different ways of representing a given fact within a particular representational framework, be it logic or anything else. The choice depends partly on which deductions need to be supported most efficiently and partly on taste. The only important thing is that within a particular knowledge base consistency of representation is critical. Since any particular inference rule is designed to work on one particular form of representation, it is necessary that all the knowledge to which that rule is intended to apply be in the form that the rule demands. Many errors in the reasoning performed by knowledge-based programs are the result of inconsistent representation decisions. The moral is simply to be careful.

There is one additional point that needs to be made here on the subject of the use of *isa* hierarchies in logic-based systems. The reason that these hierarchies are so important is not that they permit the inference of superclass membership. It is that by permitting the inference of superclass membership, they permit the inference of other properties associated with membership in that superclass. So, for example, in our sample knowledge base it is important to be able to conclude that Marcus is a Roman because we have some relevant knowledge about Romans, namely that they either hate

Caesar or are loyal to him. But recall that in the baseball example of Chapter 4, we were able to associate knowledge with superclasses that could then be overridden by more specific knowledge associated either with individual instances or with subclasses. In other words, we recorded default values that could be accessed whenever necessary. For example, there was a height associated with adult males and a different height associated with baseball players. Our procedure for manipulating the *isa* hierarchy guaranteed that we always found the correct (i.e., most specific) value for any attribute. Unfortunately, reproducing this result in logic is difficult.

Suppose, for example, that, in addition to the facts we already have, we add the following.[1]

$$Pompeian(Paulus)$$
$$\neg[loyalto(Paulus, Caesar) \vee hate(Paulus, Caesar)]$$

In other words, suppose we want to make Paulus an exception to the general rule about Romans and their feelings toward Caesar. Unfortunately, we cannot simply add these facts to our existing knowledge base the way we could just add new nodes into a semantic net. The difficulty is that if the old assertions are left unchanged, then the addition of the new assertions makes the knowledge base inconsistent. In order to restore consistency, it is necessary to modify the original assertion to which an exception is being made. So our original sentence 5 must become:

$$\forall x : Roman(x) \wedge \neg eq(x, Paulus) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$$

In this framework, every exception to a general rule must be stated twice, once in a particular statement and once in an exception list that forms part of the general rule. This makes the use of general rules in this framework less convenient and less efficient when there are exceptions than is the use of general rules in a semantic net.

A further problem arises when information is incomplete and it is not possible to prove that no exceptions apply in a particular instance. But we defer consideration of this problem until Chapter 7.

5.3 Computable Functions and Predicates

In the example we explored in the last section, all the simple facts were expressed as combinations of individual predicates, such as:

$$tryassassinate(Marcus, Caesar)$$

This is fine if the number of facts is not very large or if the facts themselves are sufficiently unstructured that there is little alternative. But suppose we want to express simple facts, such as the following greater-than and less-than relationships:

[1] For convenience, we now return to our original notation using unary predicates to denote class relations.

$$\begin{array}{ll} gt(1,0) & lt(0,1) \\ gt(2,1) & lt(1,2) \\ gt(3,2) & lt(2,3) \\ \vdots & \vdots \end{array}$$

Clearly we do not want to have to write out the representation of each of these facts individually. For one thing, there are infinitely many of them. But even if we only consider the finite number of them that can be represented, say, using a single machine word per number, it would be extremely inefficient to store explicitly a large set of statements when we could, instead, so easily compute each one as we need it. Thus it becomes useful to augment our representation by these *computable predicates*. Whatever proof procedure we use, when it comes upon one of these predicates, instead of searching for it explicitly in the database or attempting to deduce it by further reasoning, we can simply invoke a procedure, which we will specify in addition to our regular rules, that will evaluate it and return true or false.

It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of

$gt(2+3,1)$

To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to *gt*.

The next example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

Consider the following set of facts, again involving Marcus:

1. Marcus was a man.

   $man(Marcus)$

   Again we ignore the issue of tense.

2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. Marcus was born in 40 A.D.

   $born(Marcus, 40)$

   For simplicity, we will not represent A.D. explicitly, just as we normally omit it in everyday discussions. If we ever need to represent dates B.C., then we will have to decide on a way to do that, such as by using negative numbers. Notice that the representation of a sentence does not have to look like the sentence itself as long as there is a way to convert back and forth between them. This allows us to choose a representation, such as positive and negative numbers, that is easy for a program to work with.

4. All men are mortal.

   $\forall x : man(x) \rightarrow mortal(x)$

5. All Pompeians died when the volcano erupted in 79 A.D.

   $erupted(volcano, 79) \wedge \forall x : [Pompeian(x) \rightarrow died(x, 79)]$

   This sentence clearly asserts the two facts represented above. It may also assert another that we have not shown, namely that the eruption of the volcano caused the death of the Pompeians. People often assume causality between concurrent events if such causality seems plausible.

   Another problem that arises in interpreting this sentence is that of determining the referent of the phrase "the volcano." There is more than one volcano in the world. Clearly the one referred to here is Vesuvius, which is near Pompeii and erupted in 79 A.D. In general, resolving references such as these can require both a lot of reasoning and a lot of additional knowledge.

6. No mortal lives longer than 150 years.

   $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$

   There are several ways that the content of this sentence could be expressed. For example, we could introduce a function *age* and assert that its value is never greater than 150. The representation shown above is simpler, though, and it will suffice for this example.

7. It is now 1991.

   $now = 1991$

   Here we will exploit the idea of equal quantities that can be substituted for each other.

Now suppose we want to answer the question "Is Marcus alive?" A quick glance through the statements we have suggests that there may be two ways of deducing an answer. Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible. As soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking. So we add the following facts:

8. Alive means not dead.

   $\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$

   This is not strictly correct, since $\neg dead$ implies alive only for animate objects. (Chairs can be neither dead nor alive.) Again, we will ignore this for now. This is an example of the fact that rarely do two expressions have truly identical meanings in all circumstances.

9. If someone dies, then he is dead at all later times.

   $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

   This representation says that one is dead in all years after the one in which one died. It ignores the question of whether one is dead in the year in which one died.

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall x : Pompeian(x) \rightarrow died(x, 79)$
6. $erupted(volcano, 79)$
7. $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
8. $now = 1991$
9. $\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
10. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

Figure 5.4: A Set of Facts about Marcus

To answer that requires breaking time up into smaller units than years. If we do that, we can then add rules that say such things as "One is dead at *time*(*year1*, *month1*) if one died during (*year1*, *month2*) and *month2* precedes *month1*." We can extend this to days, hours, etc., as necessary. But we do not want to reduce all time statements to that level of detail, which is unnecessary and often not available.

A summary of all the facts we have now represented is given in Figure 5.4. (The numbering is changed slightly because sentence 5 has been split into two parts.) Now let's attempt to answer the question "Is Marcus alive?" by proving:

$\neg alive(Marcus, now)$

Two such proofs are shown in Figures 5.5 and 5.6. The term *nil* at the end of each proof indicates that the list of conditions remaining to be proved is empty and so the proof has succeeded. Notice in those proofs that whenever a statement of the form:

$a \wedge b \rightarrow c$

was used, a and b were set up as independent subgoals. In one sense they are, but in another sense they are not if they share the same bound variables, since, in that case, consistent substitutions must be made in each of them. For example, in Figure 5.6 look at the step justified by statement 3. We can satisfy the goal

$born(Marcus, t_1)$

using statement 3 by binding t_1 to 40, but then we must also bind t_1 to 40 in

$gt(now - t_1, 150)$

since the two t_1's were the same variable in statement 4, from which the two goals came. A good computational proof procedure has to include both a way of determining

$$
\begin{array}{ll}
\neg alive(Marcus, now) & \\
\quad\uparrow & (9, \text{substitution}) \\
dead(Marcus, now) & \\
\quad\uparrow & (10, \text{substitution}) \\
died(Marcus, t_1) \wedge gt(now, t_1) & \\
\quad\uparrow & (5, \text{substitution}) \\
Pompeian(Marcus) \wedge gt(now, 79) & \\
\quad\uparrow & (2) \\
gt(now, 79) & \\
\quad\uparrow & (8, \text{substitute equals}) \\
gt(1991, 79) & \\
\quad\uparrow & (\text{compute gt}) \\
nil &
\end{array}
$$

Figure 5.5: One Way of Proving That Marcus Is Dead

that a match exists and a way of guaranteeing uniform substitutions throughout a proof. Mechanisms for doing both those things are discussed below.

From looking at the proofs we have just shown, two things should be clear:

- Even very simple conclusions can require many steps to prove.
- A variety of processes, such as matching, substitution, and application of *modus ponens* are involved in the production of a proof. This is true even for the simple statements we are using. It would be worse if we had implications with more than a single term on the right or with complicated expressions involving *ands* and *ors* on the left.

The first of these observations suggests that if we want to be able to do nontrivial reasoning, we are going to need some statements that allow us to take bigger steps along the way. These should represent the facts that people gradually acquire as they become experts. How to get computers to acquire them is a hard problem for which no very good answer is known.

The second observation suggests that actually building a program to do what people do in producing proofs such as these may not be easy. In the next section, we introduce a proof procedure called *resolution* that reduces some of the complexity because it operates on statements that have first been converted to a single canonical form.

5.4 Resolution

As we suggest above, it would be useful from a computational point of view if we had a proof procedure that carried out in a single operation the variety of processes involved

$$\begin{tikzcd}
\neg alive(Marcus, now) \\
dead(Marcus, now) \arrow[u, "\text{(9, substitution)}"'] \\
\begin{array}{l} mortal(Marcus) \wedge \\ born(Marcus, t_1) \wedge \\ gt(now - t_1, 150) \end{array} \arrow[u, "\text{(7, substitution)}"'] \\
\begin{array}{l} man(Marcus) \wedge \\ born(Marcus, t_1) \wedge \\ gt(now - t_1, 150) \end{array} \arrow[u, "\text{(4, substitution)}"'] \\
\begin{array}{l} born(Marcus, t_1) \wedge \\ gt(now - t_1, 150) \end{array} \arrow[u, "\text{(1)}"'] \\
gt(now - 40, 150) \arrow[u, "\text{(3)}"'] \\
gt(1991 - 40, 150) \arrow[u, "\text{(8)}"'] \\
gt(1951, 150) \arrow[u, "\text{(compute minus)}"'] \\
nil \arrow[u, "\text{(compute gt)}"']
\end{tikzcd}$$

Figure 5.6: Another Way of Proving That Marcus Is Dead

in reasoning with statements in predicate logic. Resolution is such a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form, which is described below.

Resolution produces proofs by *refutation*. In other words, to prove a statement (i.e., show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable). This approach contrasts with the technique that we have been using to generate proofs by chaining backward from the theorem to be proved to the axioms. Further discussion of how resolution operates will be much more straightforward after we have discussed the standard form in which statements will be represented, so we defer it until then.

5.4.1 Conversion to Clause Form

Suppose we know that all Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy. We could represent that in the following wff:

$$\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow$$
$$[hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]$$

To use this formula in a proof requires a complex matching process. Then, having matched one piece of it, such as $thinkcrazy(x, y)$, it is necessary to do the right thing with the rest of the formula including the pieces in which the matched part is embedded and those in which it is not. If the formula were in a simpler form, this process would be much easier. The formula would be easier to work with if

- It were flatter, i.e., there was less embedding of components.
- The quantifiers were separated from the rest of the formula so that they did not need to be considered.

Conjunctive normal form [Davis and Putnam, 1960] has both of these properties. For example, the formula given above for the feelings of Romans who know Marcus would be represented in conjunctive normal form as

$$\neg Roman(x) \vee \neg know(x, Marcus) \vee$$
$$hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)$$

Since there exists an algorithm for converting any wff into conjunctive normal form, we lose no generality if we employ a proof procedure (such as resolution) that operates only on wff's in this form. In fact, for resolution to work, we need to go one step further. We need to reduce a set of wff's to a set of *clauses*, where a clause is defined to be a wff in conjunctive normal form but with no instances of the connector $\wedge$. We can do this by first converting each wff into conjunctive normal form and then breaking apart each such expression into clauses, one for each conjunct. All the conjuncts will be considered to be conjoined together as the proof procedure operates. To convert a wff into clause form, perform the following sequence of steps.

Algorithm: Convert to Clause Form

1. Eliminate $\rightarrow$, using the fact that $a \rightarrow b$ is equivalent to $\neg a \vee b$. Performing this transformation on the wff given above yields

$$\forall x : \neg[Roman(x) \wedge know(x, Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\forall y : \neg(\exists z : hate(y, z)) \vee thinkcrazy(x, y))]$$

2. Reduce the scope of each $\neg$ to a single term, using the fact that $\neg(\neg p) = p$, deMorgan's laws [which say that $\neg(a \wedge b) = \neg a \vee \neg b$ and $\neg(a \vee b) = \neg a \wedge \neg b$], and the standard correspondences between quantifiers [$\neg\forall x : P(x) = \exists x : \neg P(x)$ and $\neg\exists x : P(x) = \forall x : \neg P(x)$]. Performing this transformation on the wff from step 1 yields

$$\forall x : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\forall y : \forall z : \neg hate(y, z) \vee thinkcrazy(x, y))]$$

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff. For example, the formula

   $\forall x : P(x) \vee \forall x : Q(x)$

   would be converted to

   $\forall x : P(x) \vee \forall y : Q(y)$

   This step is in preparation for the next.

4. Move all quantifiers to the left of the formula without changing their relative order. This is possible since there is no conflict among variable names. Performing this operation on the formula of step 2, we get

   $\forall x : \forall y : \forall z : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee$
   $[hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x, y))]$

   At this point, the formula is in what is known as *prenex normal form*. It consists of a *prefix* of quantifiers followed by a *matrix*, which is quantifier-free.

5. Eliminate existential quantifiers. A formula that contains an existentially quantified variable asserts that there is a value that can be substituted for the variable that makes the formula true. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. Since we do not necessarily know how to produce the value, we must create a new function name for every such replacement. We make no assertions about these functions except that they must exist. So, for example, the formula

   $\exists y : President(y)$

   can be transformed into the formula

   $President(S1)$

   where $S1$ is a function with no arguments that somehow produces a value that satisfies President.

   If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example, in the formula

   $\forall x : \exists y : father\text{-}of(y, x)$

   the value of y that satisfies *father-of* depends on the particular value of x. Thus we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this example would be transformed into

   $\forall x : father\text{-}of(S2(x), x))$

   These generated functions are called *Skolem functions*. Sometimes ones with no arguments are called *Skolem constants*.

6. Drop the prefix. At this point, all remaining variables are universally quantified, so the prefix can just be dropped and any proof procedure we use can simply assume

that any variable it sees is universally quantified. Now the formula produced in step 4 appears as

$[\neg Roman(x) \lor \neg know(x, Marcus)] \lor$
$\quad [hate(x, Caesar) \lor (\neg hate(y, z) \lor thinkcrazy(x, y))]$

7. Convert the matrix into a conjunction of disjuncts. In the case of our example, since there are no *and*'s, it is only necessary to exploit the associative property of *or* [i.e., $a \lor (b \lor c) = (a \lor b) \lor c$] and simply remove the parentheses, giving

   $\neg Roman(x) \lor \neg know(x, Marcus) \lor$
   $\quad hate(x, Caesar) \lor \neg hate(y, z) \lor thinkcrazy(x, y)$

   However, it is also frequently necessary to exploit the distributive property [i.e., $(a \land b) \lor c = (a \lor c) \land (b \lor c)$]. For example, the formula

   $(winter \land wearingboots) \lor (summer \land wearingsandals)$

   becomes, after one application of the rule

   $[winter \lor (summer \land wearingsandals)]$
   $\quad \land [wearingboots \lor (summer \land wearingsandals)]$

   and then, after a second application, required since there are still conjuncts joined by OR's,

   $(winter \lor summer) \land$
   $(winter \lor wearingsandals) \land$
   $(wearingboots \lor summer) \land$
   $(wearingboots \lor wearingsandals)$

8. Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the same set of facts as were represented by the original wff's.

9. Standardize apart the variables in the set of clauses generated in step 8. By this we mean rename the variables so that no two clauses make reference to the same variable. In making this transformation, we rely on the fact that

   $(\forall x : P(x) \land Q(x)) = \forall x : P(x) \land \forall x : Q(x)$

   Thus since each clause is a separate conjunct and since all the variables are universally quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Performing this final step of standardization is important because during the resolution procedure it is sometimes necessary to instantiate a universally quantified variable (i.e., substitute for it a particular value). But, in general, we want to keep clauses in their most general form as long as possible. So when a variable is instantiated, we want to know the minimum number of substitutions that must be made to preserve the truth value of the system.

After applying this entire procedure to a set of wff's, we will have a set of clauses, each of which is a disjunction of *literals*. These clauses can now be exploited by the resolution procedure to generate proofs.

5.4.2 The Basis of Resolution

The resolution procedure is a simple iterative process: at each step, two clauses, called the *parent clauses*, are compared (*resolved*), yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

$winter \vee summer$
$\neg winter \vee cold$

Recall that this means that both clauses must be true (i.e., the clauses, although they look independent, are really conjoined).

Now we observe that precisely one of *winter* and $\neg winter$ will be true at any point. If *winter* is true, then *cold* must be true to guarantee the truth of the second clause. If $\neg winter$ is true, then *summer* must be true to guarantee the truth of the first clause. Thus we see that from these two clauses we can deduce

$summer \vee cold$

This is the deduction that the resolution procedure will make. Resolution operates by taking two clauses that each contain the same literal, in this example, *winter*. The literal must occur in positive form in one clause and in negative form in the other. The *resolvent* is obtained by combining all of the literals of the two parent clauses except the ones that cancel.

If the clause that is produced is the empty clause, then a contradiction has been found. For example, the two clauses

$winter$
$\neg winter$

will produce the empty clause. If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although as we will see, there are often ways of detecting that no contradiction exists.

So far, we have discussed only resolution in propositional logic. In predicate logic, the situation is more complicated since we must consider all possible ways of substituting values for the variables. The theoretical basis of the resolution procedure in predicate logic is Herbrand's theorem [Chang and Lee, 1973], which tells us the following:

- To show that a set of clauses S is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the *Herbrand universe* of S.
- A set of clauses S is unsatisfiable if and only if a finite subset of ground instances (in which all bound variables have had a value substituted for them) of S is unsatisfiable.

The second part of the theorem is important if there is to exist any computational procedure for proving unsatisfiability, since in a finite amount of time no procedure will be able to examine an infinite set. The first part suggests that one way to go about finding a contradiction is to try systematically the possible substitutions and see if each produces a contradiction. But that is highly inefficient. The resolution principle, first introduced by Robinson [1965], provides a way of finding contradictions by trying a minimum number of substitutions. The idea is to keep clauses in their general form as long as possible and only introduce specific substitutions when they are required. For more details on different kinds of resolution, see Stickel [1988].

5.4.3 Resolution in Propositional Logic

In order to make it clear how resolution works, we first present the resolution procedure for propositional logic. We then expand it to include predicate logic.

In propositional logic, the procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following.

Algorithm: Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
    (a) Select two clauses. Call these the parent clauses.
    (b) Resolve them together. The resulting clause, called the *resolvent*, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
    (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Let's look at a simple example. Suppose we are given the axioms shown in the first column of Figure 5.7 and we want to prove R. First we convert the axioms to clause form, as shown in the second column of the figure. Then we negate R, producing $\neg R$, which is already in clause form. Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of producing the empty clause (shown as a box). We might, for example, generate the sequence of resolvents shown in Figure 5.8. We begin by resolving with the clause $\neg R$ since that is one of the clauses that must be involved in the contradiction we are trying to find.

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and, based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause. To see how this works, let's look again at the example. In order for proposition 2 to be true, one of three things must be true: $\neg P$, $\neg Q$, or R. But we are assuming that $\neg R$ is true. Given

Given Axioms	Converted to Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

Figure 5.7: A Few Facts in Propositional Logic

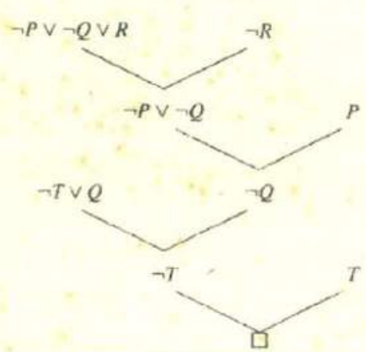


Figure 5.8: Resolution in Propositional Logic

that, the only way for proposition 2 to be true is for one of two things to be true: $\neg P$ or $\neg Q$. That is what the first resolvent clause says. But proposition 1 says that P is true, which means that $\neg P$ cannot be true, which leaves only one way for proposition 2 to be true, namely for $\neg Q$ to be true (as shown in the second resolvent clause). Proposition 4 can be true if either $\neg T$ or Q is true. But since we now know that $\neg Q$ must be true, the only way for proposition 4 to be true is for $\neg T$ to be true (the third resolvent). But proposition 5 says that T is true. Thus there is no way for all of these clauses to be true in a single interpretation. This is indicated by the empty clause (the last resolvent).

5.4.4 The Unification Algorithm

In propositional logic, it is easy to determine that two literals cannot both be true at the same time. Simply look for L and $\neg L$. In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered. For example, *man*(*John*) and $\neg$*man*(*John*) is a contradiction, while *man*(*John*) and $\neg$*man*(*Spot*) is not. Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursive procedure, called the *unification algorithm*, that does just this.

The basic idea of unification is very simple. To attempt to unify two literals, we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can be unified, regardless of their arguments. For example, the two literals

tryassassinate(*Marcus*, *Caesar*)
hate(*Marcus*, *Caesar*)

cannot be unified. If the predicate symbols match, then we must check the arguments one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair, we can simply call the unification procedure recursively. The matching rules are simple. Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression, with the restriction that the predicate expression must not contain any instances of the variable being matched.

The only complication in this procedure is that we must find a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue trying to unify them. For example, suppose we want to unify the expressions

$P(x, x)$
$P(y, z)$

The two instances of P match fine. Next we compare x and y, and decide that if we substitute y for x, they could match. We will write that substitution as

y/x

(We could, of course, have decided instead to substitute x for y, since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions.) But now, if we simply continue and match x and z, we produce the substitution z/x. But we cannot substitute both y and z for x, so we have not produced a consistent substitution.

What we need to do after finding the first substitution y/x is to make that substitution throughout the literals, giving

$P(y, y)$
$P(y, z)$

Now we can attempt to unify arguments y and z, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as

$(z/y)(y/x)$

following standard notation for function composition. In general, the substitution $(a_1/a_2, a_3/a_4, \ldots)(b_1/b_2, b_3/b_4, \ldots)\ldots$ means to apply all the substitutions of the right-most list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.

The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. For example, the literals

$hate(x, y)$
$hate(Marcus, z)$

could be unified with any of the following substitutions:

$(Marcus/x, z/y)$
$(Marcus/x, y/z)$
$(Marcus/x, Caesar/y, Caesar/z)$
$(Marcus/x, Polonius/y, Polonius/z)$

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown below will do that.

Having explained the operation of the unification algorithm, we can now state it concisely. We describe a procedure Unify($L1$, $L2$), which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(L1, L2)

```
1. If $L1$ or $L2$ are both variables or constants, then:
    (a) If $L1$ and $L2$ are identical, then return NIL.
    (b) Else if $L1$ is a variable, then if $L1$ occurs in $L2$ then return {FAIL}, else
        return ($L2$/$L1$).
    (c) Else if $L2$ is a variable then if $L2$ occurs in $L1$ then return {FAIL}, else
        return ($L1$/$L2$).
    (d) Else return {FAIL}.
2. If the initial predicate symbols in $L1$ and $L2$ are not identical, then return {FAIL}.
3. If $L1$ and $L2$ have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the
   substitutions used to unify $L1$ and $L2$.)
5. For $i \leftarrow 1$ to number of arguments in $L1$:
    (a) Call Unify with the ith argument of L1 and the ith argument of L2, putting
        result in S.
    (b) If S contains FAIL then return {FAIL}.
    (c) If S is not equal to NIL then:
        i. Apply S to the remainder of both L1 and L2.
        ii. SUBST := APPEND(S, SUBST).
6. Return SUBST.
```

The only part of this algorithm that we have not yet discussed is the check in steps 1(*b*) and 1(*c*) to make sure that an expression involving a given variable is not unified with that variable. Suppose we were attempting to unify the expressions

$f(x, x)$
$f(g(x), g(x))$

If we accepted $g(x)$ as a substitution for x, then we would have to substitute it for x in the remainder of the expressions. But this leads to infinite recursion since it will never be possible to eliminate x.

Unification has deep mathematical roots and is a useful operation in many AI programs, for example, theorem provers and natural language parsers. As a result, efficient data structures and algorithms for unification have been developed. For an introduction to these techniques and applications, see Knight [1989].

5.4.5 Resolution in Predicate Logic

We now have an easy way of determining that two literals are contradictory—they are if one of them can be unified with the negation of the other. So, for example, $man(x)$ and $\neg man(Spot)$ are contradictory, since $man(x)$ and $man(Spot)$ can be unified. This corresponds to the intuition that says that $man(x)$ cannot be true for all x if there is known to be some x, say Spot, for which $man(x)$ is false. Thus in order to use resolution for expressions in the predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.

We also need to use the unifier produced by the unification algorithm to generate the resolvent clause. For example, suppose we want to resolve two clauses:

1. $man(Marcus)$
2. $\neg man(x_1) \vee mortal(x_1)$

The literal $man(Marcus)$ can be unified with the literal $man(x_1)$ with the substitutio $Marcus/x_1$, telling us that for $x_1 = Marcus$, $\neg man(Marcus)$ is false. But we cannot simply cancel out the two *man* literals as we did in propositional logic and generate the resolvent $mortal(x_1)$. Clause 2 says that for a given x_1, either $\neg man(x_1)$ or $mortal(x_1)$. So for it to be true, we can now conclude only that $mortal(Marcus)$ must be true. It is not necessary that $mortal(x_1)$ be true for all x_1, since for some values of x_1, $\neg man(x_1)$ might be true, making $mortal(x_1)$ irrelevant to the truth of the complete clause. So the resolvent generated by clauses 1 and 2 must be $mortal(Marcus)$, which we get by applying the result of the unification process to the resolvent. The resolution process can then proceed from there to discover whether *mortal*(*Marcus*) leads to a contradiction with other available clauses.

This example illustrates the importance of standardizing variables apart during the process of converting expressions to clause form. Given that that standardization has been done, it is easy to determine how the unifier must be used to perform substitutions to create the resolvent. If two instances of the same variable occur, then they must be given identical substitutions.

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.

2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.

3. Repeat until either a contradiction is found, no progress can be made, or a predetermined amount of effort has been expended.

   (a) Select two clauses. Call these the parent clauses.

   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals $T1$ and $\neg T2$ such that one of the parent clauses contains $T1$ and the other contains $T2$ and if $T1$ and $T2$ are unifiable, then neither $T1$ nor $T2$ should appear in the resolvent. We call $T1$ and $T2$ *complementary literals*. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.

   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably:

- Only resolve pairs of clauses that contain complementary literals, since only such resolutions produce new clauses that are harder to satisfy than their parents. To facilitate this, index clauses by the predicates they contain, combined with an indication of whether the predicate is negated. Then, given a particular clause, possible resolvents that contain a complementary occurrence of one of its predicates can be located directly.

- Eliminate certain clauses as soon as they are generated so that they cannot participate in later resolutions. Two kinds of clauses should be eliminated: tautologies (which can never be unsatisfied) and clauses that are subsumed by other clauses (i.e., they are easier to satisfy. For example, $P \vee Q$ is subsumed by P.)

- Whenever possible, resolve either with one of the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause. This is called the *set-of-support strategy* and corresponds to the intuition that the contradiction we are looking for must involve the statement we are trying to prove. Any other contradiction would say that the previously believed statements were inconsistent.
- Whenever possible, resolve with clauses that have a single literal. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses and thus are probably closer to the goal of a resolvent with zero terms. This method is called the *unit-preference strategy*.

Let's now return to our discussion of Marcus and show how resolution can be used to prove new things about him. Let's first consider the set of statements introduced in Section 5.1. To use them in resolution proofs, we must convert them to clause form as described in Section 5.4.1. Figure 5.9(*a*) shows the results of that conversion. Figure 5.9(*b*) shows a resolution proof of the statement

hate(*Marcus*, *Caesar*)

Of course, many more resolvents could have been generated than we have shown, but we used the heuristics described above to guide the search. Notice that what we have done here essentially is to reason backward from the statement we want to show is a contradiction through a set of intermediate conclusions to the final conclusion of inconsistency.

Suppose our actual goal in proving the assertion

hate(*Marcus*. *Caesar*)

was to answer the question "Did Marcus hate Caesar?" In that case, we might just as easily have attempted to prove the statement

¬*hate*(*Marcus*. *Caesar*)

To do so, we would have added

hate(*Marcus*, *Caesar*)

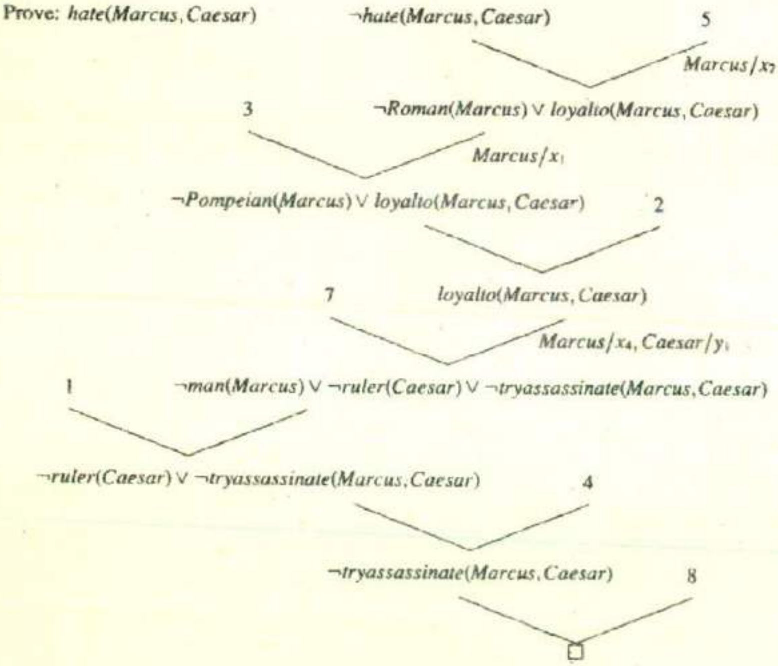
to the set of available clauses and begun the resolution process. But immediately we notice that there are no clauses that contain a literal involving ¬*hate*. Since the resolution process can only generate new clauses that are composed of combinations of literals from already existing clauses, we know that no such clause can be generated and thus we conclude that *hate*(*Marcus*, *Caesar*) will not produce a contradiction with the known statements. This is an example of the kind of situation in which the resolution procedure can detect that no contradiction exists. Sometimes this situation is detected not at the beginning of a proof, but part way through, as shown in the example in Figure 5.10(*a*), based on the axioms given in Figure 5.9.

But suppose our knowledge base contained the two additional statements

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. $ruler(Caesar)$
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f1(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinote(x_4, y_1) \vee loyalto(x_4, y_1)$
8. $tryassassinate(Marcus, Caesar)$

(a)


Prove: hate(Marcus, Caesar)
¬hate(Marcus, Caesar)
5
Marcus/x₂
3
¬Roman(Marcus) ∨ loyalto(Marcus, Caesar)
Marcus/x₁
¬Pompeian(Marcus) ∨ loyalto(Marcus, Caesar)
2
7
loyalto(Marcus, Caesar)
Marcus/x₄, Caesar/y₁
1
¬man(Marcus) ∨ ¬ruler(Caesar) ∨ ¬tryassassinate(Marcus, Caesar)
¬ruler(Caesar) ∨ ¬tryassassinate(Marcus, Caesar)
4
¬tryassassinate(Marcus, Caesar)
8
□

(b)

Figure 5.9: A Resolution Proof

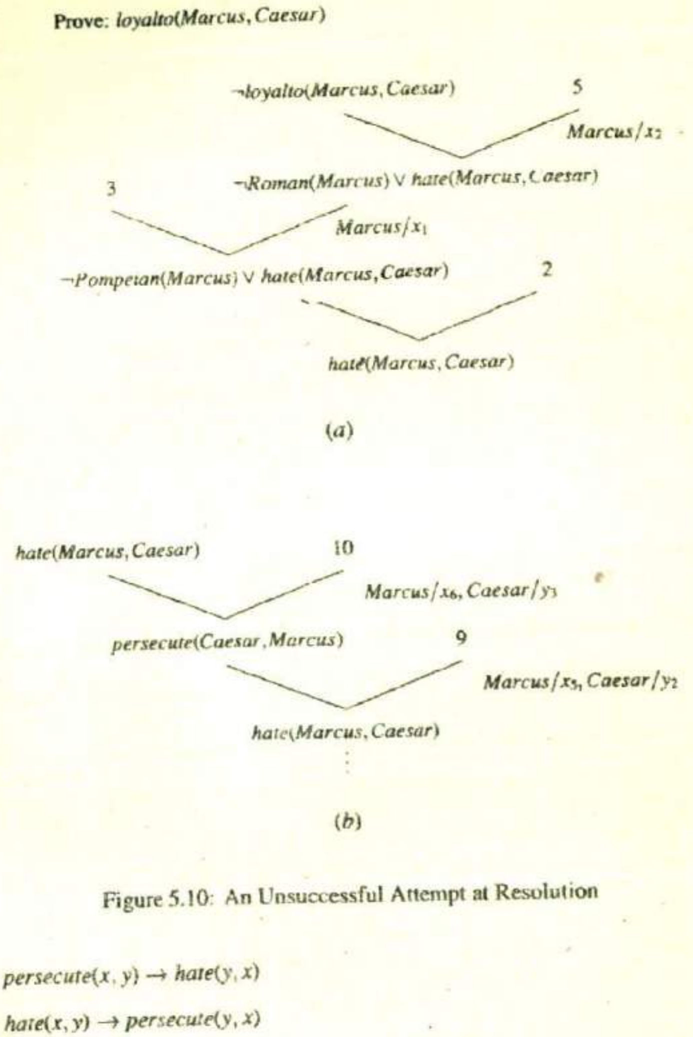


Figure 5.10: An Unsuccessful Attempt at Resolution

9. $persecute(x, y) \rightarrow hate(y, x)$

10. $hate(x, y) \rightarrow persecute(y, x)$

Converting to clause form, we get

9. $\neg persecute(x_5, y_2) \lor hate(y_2, x_5)$

10. $\neg hate(x_6, y_3) \lor persecute(y_3, x_6)$

These statements enable the proof of Figure 5.10(a) to continue as shown in Figure 5.10(b). Now to detect that there is no contradiction we must discover that the only resolvents that can be generated have been generated before. In other words, although we can generate resolvents, we can generate no new ones.

Given:

1. $\neg father(x, y) \vee \neg woman(x)$
   (i.e., $father(x, y) \rightarrow \neg woman(x)$)
2. $\neg mother(x, y) \vee woman(x)$
   (i.e., $mother(x, y) \rightarrow woman(x)$)
3. $mother(Chris, Mary)$
4. $father(Chris, Bill)$

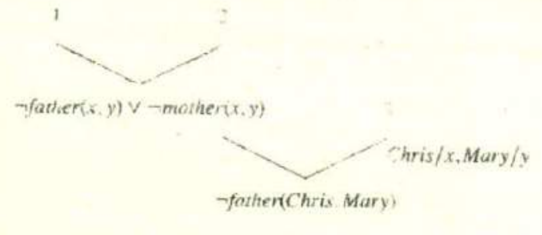


Figure 5.11: The Need to Standardize Variables

Recall that the final step of the process of converting a set of formulas to clause form was to standardize apart the variables that appear in the final clauses. Now that we have discussed the resolution procedure, we can see clearly why this step is so important. Figure 5.11 shows an example of the difficulty that may arise if standardization is not done. Because the variable y occurs in both clause 1 and clause 2, the substitution at the second resolution step produces a clause that is too restricted and so does not lead to the contradiction that is present in the database. If, instead, the clause

$\neg father(Chris, y)$

had been produced, the contradiction with clause 4 would have emerged. This would have happened if clause 2 had been rewritten as

$\neg mother(a, b) \vee woman(a)$

In its pure form, resolution requires all the knowledge it uses to be represented in the form of clauses. But as we pointed out in Section 5.3, it is often more efficient to represent certain kinds of information in the form of computable functions, computable predicates, and equality relationships. It is not hard to augment resolution to handle this sort of knowledge. Figure 5.12 shows a resolution proof of the statement

$\neg alive(Marcus, now)$

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg Pompeian(x_2) \vee died(x_2, 79)$
6. $erupted(volcano, 79)$
7. $\neg mortal(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. $now = 1991$

9a. $\neg alive(x_4, t_3) \vee \neg dead(x_4, t_3)$

9b. $dead(x_5, t_4) \vee alive(x_5, t_4)$

10. $\neg died(x_6, t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

Prove: $\neg alive(Marcus, now)$

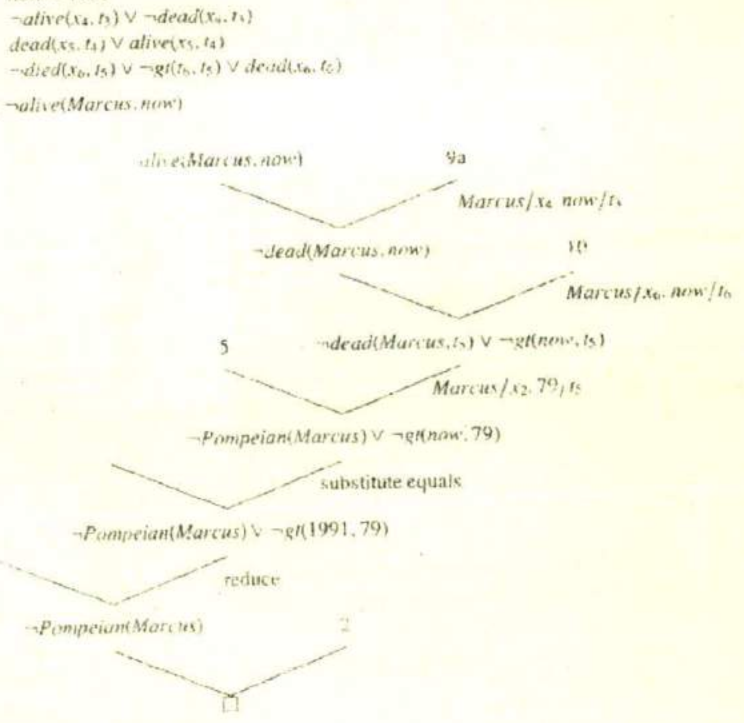


Figure 5.12: Using Resolution with Equality and Reduce

based on the statements given in Section 5.3. We have added two ways of generating new clauses, in addition to the resolution rule:

- Substitution of one value for another to which it is equal.
- Reduction of computable predicates. If the predicate evaluates to FALSE, it can simply be dropped, since adding $\vee$ FALSE to a disjunction cannot change its truth value. If the predicate evaluates to TRUE, then the generated clause is a tautology and cannot lead to a contradiction.

5.4.6 The Need to Try Several Substitutions

Resolution provides a very good way of finding a refutation proof without actually trying all the substitutions that Herbrand's theorem suggests might be necessary. But it does not always eliminate the necessity of trying more than one substitution. For example, suppose we know, in addition to the statements in Section 5.1, that

hate(*Marcus*, *Paulus*)
hate(*Marcus*, *Julian*)

Now if we want to prove that Marcus hates some ruler, we would be likely to try each substitution shown in Figure 5.13(*a*) and (*b*) before finding the contradiction shown in (*c*). Sometimes there is no way short of very good luck to avoid trying several substitutions.

5.4.7 Question Answering

Very early in the history of AI it was realized that theorem-proving techniques could be applied to the problem of answering questions. As we have already suggested, this seems natural since both deriving theorems from axioms and deriving new facts (answers) from old facts employ the process of deduction. We have already shown how resolution can be used to answer yes-no questions, such as "Is Marcus alive?" In this section, we show how resolution can be used to answer fill-in-the-blank questions, such as "When did Marcus die?" or "Who tried to assassinate a ruler?" Answering these questions involves finding a known statement that matches the terms given in the question and then responding with another piece of that same statement that fills the slot demanded by the question. For example, to answer the question "When did Marcus die?" we need a statement of the form

died(*Marcus*, ??)

with ?? actually filled in by some particular year. So, since we can prove the statement

died(*Marcus*, 79)

we can respond with the answer 79.

It turns out that the resolution procedure provides an easy way of locating just the statement we need and finding a proof for it. Let's continue with the example question

Prove: $\exists x : hate(Marcus, x) \wedge ruler(x)$
(negate): $\neg\exists x : hate(Marcus, x) \wedge ruler(x)$
(clausify): $\neg hate(Marcus, x) \vee \neg ruler(x)$

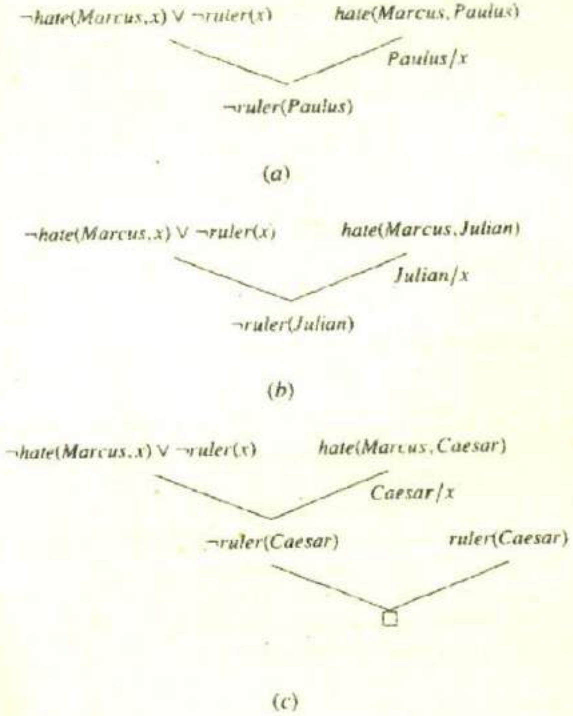


Figure 5.13: Trying Several Substitutions

"When did Marcus die?" In order to be able to answer this question, it must first be true that Marcus died. Thus it must be the case that

$\exists t : died(Marcus, t)$

A reasonable first step then might be to try to prove this. To do so using resolution, we attempt to show that

$\neg\exists t : died(Marcus, t)$

produces a contradiction. What does it mean for that statement to produce a contradiction? Either it conflicts with a statement of the form

$\forall t : died(Marcus, t)$

where t is a variable, in which case we can either answer the question by reporting that there are many times at which Marcus died, or we can simply pick one such time and respond with it. The other possibility is that we produce a contradiction with one or more specific statements of the form

$died(Marcus, date)$

for some specific value of *date*. Whatever value of date we use in producing that contradiction is the answer we want. The value that proves that there is a value (and thus the inconsistency of the statement that there is no such value) is exactly the value we want.

Figure 5.14(a) shows how the resolution process finds the statement for which we are looking. The answer to the question can then be derived from the chain of unifications that lead back to the starting clause. We can eliminate the necessity for this final step by adding an additional expression to the one we are going to use to try to find a contradiction. This new expression will simply be the one we are trying to prove true (i.e., it will be the negation of the expression that is actually used in the resolution). We can tag it with a special marker so that it will not interfere with the resolution process. (In the figure, it is underlined.) It will just get carried along, but each time unification is done, the variables in this dummy expression will be bound just as are the ones in the clauses that are actively being used. Instead of terminating on reaching the nil clause, the resolution procedure will terminate when all that is left is the dummy expression. The bindings of its variables at that point provide the answer to the question. Figure 5.14(b) shows how this process produces an answer to our question.

Unfortunately, given a particular representation of the facts in a system, there will usually be some questions that cannot be answered using this mechanism. For example, suppose that we want to answer the question "What happened in 79 A.D.?" using the statements in Section 5.3. In order to answer the question, we need to prove that something happened in 79. We need to prove

$\exists x : event(x, 79)$

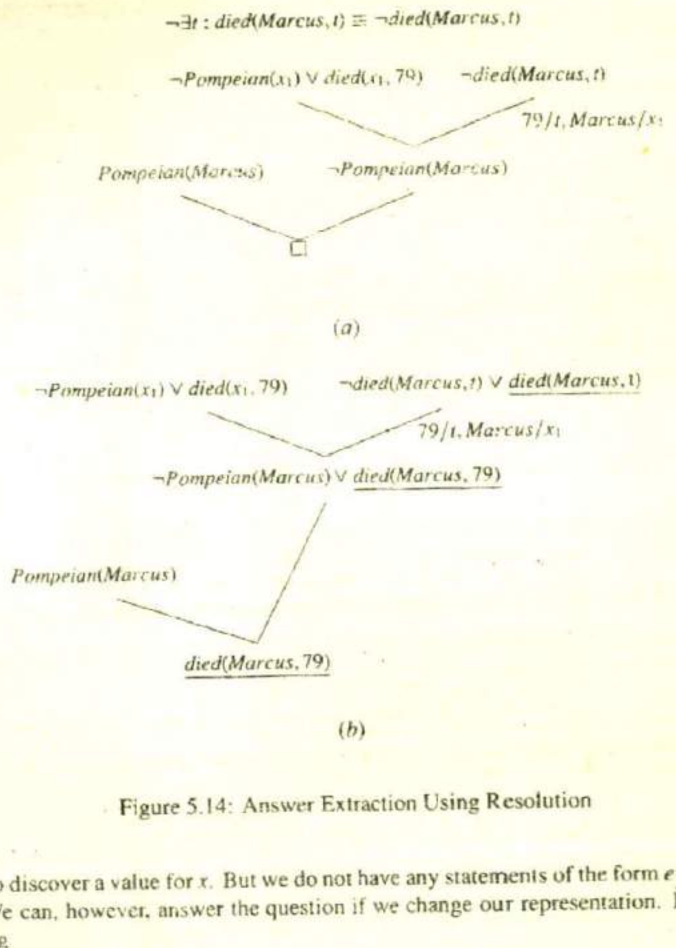

¬∃t : died(Marcus, t) ≡ ¬died(Marcus, t)
¬Pompeian(x₁) ∨ died(x₁, 79)
¬died(Marcus, t)
79/t, Marcus/x₁
Pompeian(Marcus)
¬Pompeian(Marcus)
□
(a)
¬Pompeian(x₁) ∨ died(x₁, 79)
¬died(Marcus, t) ∨ died(Marcus, t)
79/t, Marcus/x₁
¬Pompeian(Marcus) ∨ died(Marcus, 79)
Pompeian(Marcus)
died(Marcus, 79)
(b)

Figure 5.14: Answer Extraction Using Resolution

and to discover a value for x. But we do not have any statements of the form *event*(x, y).

We can, however, answer the question if we change our representation. Instead of saying

erupted(*volcano*, 79)

we can say

event(*erupted*(*volcano*), 79)

Then the simple proof shown in Figure 5.15 enables us to answer the question.

This new representation has the drawback that it is more complex than the old one. And it still does not make it possible to answer all conceivable questions. In general, it is necessary to decide on the kinds of questions that will be asked and to design a representation appropriate for those questions.

$$\begin{tikzcd}
\neg event(x,79) \lor \underline{event(x,79)} \arrow[dr, dash] & & event(erupted(volcano),79) \arrow[dl, dash, "erupted(volcano)/x"] \\
& \underline{event(erupted(volcano,79))} &
\end{tikzcd}$$

Figure 5.15: Using the New Representation

Of course, yes-no and fill-in-the-blank questions are not the only kinds one could ask. For example, we might ask how to do something. So we have not yet completely solved the problem of question answering. In later chapters, we discuss some other methods for answering a variety of questions. Some of them exploit resolution; others do not.

5.5 Natural Deduction

In the last section, we introduced resolution as an easily implementable proof procedure that relies for its simplicity on a uniform representation of the statements it uses. Unfortunately, uniformity has its price—everything looks the same. Since everything looks the same, there is no easy way to select those statements that are the most likely to be useful in solving a particular problem. In converting everything to clause form, we often lose valuable heuristic information that is contained in the original representation of the facts. For example, suppose we believe that all judges who are not crooked are well-educated, which can be represented as

$$\forall x : judge(x) \land \neg crooked(x) \rightarrow educated(x)$$

In this form, the statement suggests a way of deducing that someone is educated. But when the same statement is converted to clause form,

$$\neg judge(x) \lor crooked(x) \lor educated(x)$$

it appears also to be a way of deducing that someone is not a judge by showing that he is not crooked and not educated. Of course, in a logical sense, it is. But it is almost certainly not the best way, or even a very good way, to go about showing that someone is not a judge. The heuristic information contained in the original statement has been lost in the transformation.

Another problem with the use of resolution as the basis of a theorem-proving system is that people do not think in resolution. Thus it is very difficult for a person to interact with a resolution theorem prover, either to give it advice or to be given advice by it. Since proving very hard things is something that computers still do poorly, it is important from a practical standpoint that such interaction be possible. To facilitate it, we are forced to look for a way of doing machine theorem proving that corresponds more closely to the processes used in human theorem proving. We are thus led to what we call, mostly by definition, *natural deduction*.

Natural deduction is not a precise term. Rather it describes a melange of techniques, used in combination to solve problems that are not tractable by any one method alone. One common technique is to arrange knowledge, not by predicates, as we have been doing, but rather by the objects involved in the predicates. Some techniques for doing this are described in Chapter 9. Another technique is to use a set of rewrite rules that not only describe logical implications but also suggest the way that those implications can be exploited in proofs.

For a good survey of the variety of techniques that can be exploited in a natural deduction system, see Bledsoe [1977]. Although the emphasis in that paper is on proving mathematical theorems, many of the ideas in it can be applied to a variety of domains in which it is necessary to deduce new statements from known ones. For another discussion of theorem proving using natural mechanisms, see Boyer and Moore [1988], which describes a system for reasoning about programs. It places particular emphasis on the use of mathematical induction as a proof technique.

5.6 Summary

In this chapter we showed how predicate logic can be used as the basis of a technique for knowledge representation. We also discussed a problem-solving technique, resolution, that can be applied when knowledge is represented in this way. The resolution procedure is not guaranteed to halt if given a nontheorem to prove. But is it guaranteed to halt and find a contradiction if one exists? This is called the *completeness* question. In the form in which we have presented the algorithm, the answer to this question is no. Some small changes, usually not implemented in theorem-proving systems, must be made to guarantee completeness. But, from a computational point of view, completeness is not the only important question. Instead, we must ask whether a proof can be found in the limited amount of time that is available. There are two ways to approach achieving this computational goal. The first is to search for good heuristics that can inform a theorem-proving program. Current theorem-proving research attempts to do this. The other approach is to change not the program but the data given to the program. In this approach, we recognize that a knowledge base that is just a list of logical assertions possesses no structure. Suppose an information-bearing structure could be imposed on such a knowledge base. Then that additional information could be used to guide the program that uses the knowledge. Such a program may not look a lot like a theorem prover, although it will still be a knowledge-based problem solver. We discuss this idea further in Chapter 9.

A second difficulty with the use of theorem proving in AI systems is that there are some kinds of information that are not easily represented in predicate logic. Consider the following examples:

- "It is very hot today." How can relative degrees of heat be represented?
- "Blond-haired people often have blue eyes." How can the amount of certainty be represented?

- "If there is no evidence to the contrary, assume that any adult you meet knows how to read." How can we represent that one fact should be inferred from the absence of another?
- "It's better to have more pieces on the board than the opponent has." How can we represent this kind of heuristic information?
- "I know Bill thinks the Giants will win, but I think they are going to lose." How can several different belief systems be represented at once?

These examples suggest issues in knowledge representation that we have not yet satisfactorily addressed. They deal primarily with the need to make do with a knowledge base that is incomplete, although other problems also exist, such as the difficulty of representing continuous phenomena in a discrete system. Some solutions to these problems are presented in the remaining chapters in this part of the book.

5.7 Exercises

1. Using facts 1-9 of Section 5.1, answer the question, "Did Marcus hate Caesar?"
2. In Section 5.3, we showed that given our facts, there were two ways to prove the statement $\neg alive(Marcus, now)$. In Figure 5.12 a resolution proof corresponding to one of those methods is shown. Use resolution to derive another proof of the statement using the other chain of reasoning.
3. Trace the operation of the unification algorithm on each of the following pairs of literals:
   (a) $f(Marcus)$ and $f(Caesar)$
   (b) $f(x)$ and $f(g(y))$
   (c) $f(Marcus, g(x, y))$ and $f(x, g(Caesar, Marcus))$
4. Consider the following sentences:
   - John likes all kinds of food.
   - Apples are food.
   - Chicken is food.
   - Anything anyone eats and isn't killed by is food.
   - Bill eats peanuts and is still alive.
   - Sue eats everything Bill eats.

   (a) Translate these sentences into formulas in predicate logic.
   (b) Prove that John likes peanuts using backward chaining.
   (c) Convert the formulas of part a into clause form.
   (d) Prove that John likes peanuts using resolution.

(c) Use resolution to answer the question, "What food does Sue eat?"

5. Consider the following facts:

   - The members of the Elm St. Bridge Club are Joe, Sally, Bill, and Ellen.
   - Joe is married to Sally.
   - Bill is Ellen's brother.
   - The spouse of every married person in the club is also in the club.
   - The last meeting of the club was at Joe's house.

   (a) Represent these facts in predicate logic.

   (b) From the facts given above, most people would be able to decide on the truth of the following additional statements:

      - The last meeting of the club was at Sally's house.
      - Ellen is not married.

      Can you construct resolution proofs to demonstrate the truth of each of these statements given the five facts listed above? Do so if possible. Otherwise, add the facts you need and then construct the proofs.

6. Assume the following facts:

   - Steve only likes easy courses.
   - Science courses are hard.
   - All the courses in the basketweaving department are easy.
   - BK301 is a basketweaving course.

   Use resolution to answer the question, "What course would Steve like?"

7. In Section 5.4.7, we answered the question, "When did Marcus die?" by using resolution to show that there was a time when Marcus died. Using the facts given in Figure 5.4, and the additional fact

   $\forall x : \forall t_1 : dead(x, t_1) \rightarrow \exists t_2 : gt(t_1, t_2) \wedge died(x, t_2)$

   there is another way to show that there was a time when Marcus died.

   (a) Do a resolution proof of this other chain of reasoning.

   (b) What answer will this proof give to the question, "When did Marcus die?"

8. Suppose that we are attempting to resolve the following clauses:

   $loves(father(a), a)$
   $\neg\, loves(y, x) \vee loves(x, y)$

   (a) What will be the result of the unification algorithm when applied to clause 1 and the first term of clause 2?

   (b) What must be generated as a result of resolving these two clauses?

(c) What does this example show about the order in which the substitutions determined by the unification procedure must be performed?

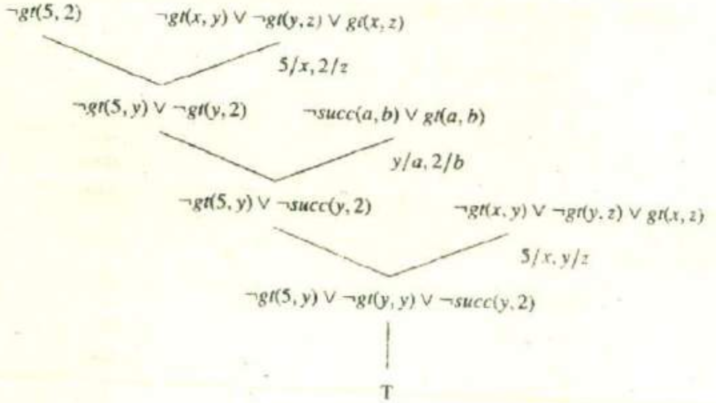
9. Suppose you are given the following facts:

   $\forall x, y, z : gt(x, y) \wedge gt(y, z) \rightarrow gt(x, z)$
   $\forall a, b : succ(a, b) \rightarrow gt(a, b)$
   $\forall x : \neg gt(x, x)$

   You want to prove that

   $gt(5, 2)$

   Consider the following attempt at a resolution proof:


$\neg gt(5, 2)$
$\neg gt(x, y) \vee \neg gt(y, z) \vee gt(x, z)$
$5/x, 2/z$
$\neg gt(5, y) \vee \neg gt(y, 2)$
$\neg succ(a, b) \vee gt(a, b)$
$y/a, 2/b$
$\neg gt(5, y) \vee \neg succ(y, 2)$
$\neg gt(x, y) \vee \neg gt(y, z) \vee gt(x, z)$
$5/x, y/z$
$\neg gt(5, y) \vee \neg gt(y, y) \vee \neg succ(y, 2)$
T

   (a) What went wrong?

   (b) What needs to be added to the resolution procedure to make sure that this does not happen?

10. The answer to the last problem suggests that the unification procedure could be simplified by omitting the check that prevents x and $f(x)$ from being unified together (the *occur check*). This should be possible since no two clauses will ever share variables. If x occurs in one, $f(x)$ cannot occur in another. But suppose the unification procedure is given the following two clauses (in the notation of Section 5.4.4):

    $p(x, f(x))$
    $p(f(a), a)$

    Trace the execution of the procedure. What does this example show about the need for the occur check?

11. What is wrong with the following argument [Henle, 1965]?

    - Men are widely distributed over the earth.
    - Socrates is a man.

- Therefore, Socrates is widely distributed over the earth.

How should the facts represented by these sentences be represented in logic so that this problem does not arise?

12. Consider all the facts about baseball that are represented in the slot-and-filler structure of Figure 4.5. Represent those same facts as a set of assertions in predicate logic. Show how the inferences that were derived from that knowledge in Section 4.2 can be derived using logical deduction.

13. What problems would be encountered in attempting to represent the following statements in predicate logic? It should be possible to deduce the final statement from the others.

    - John only likes to see French movies.
    - It's safe to assume a movie is American unless explicitly told otherwise.
    - The Playhouse rarely shows foreign films.
    - People don't do things that will cause them to be in situations that they don't like.
    - John doesn't go to the Playhouse very often.

Chapter 6

Representing Knowledge Using Rules

In this chapter, we discuss the use of rules to encode knowledge. This is a particularly important issue since rule-based reasoning systems have played a very important role in the evolution of AI from a purely laboratory science into a commercially significant one, as we see later in Chapter 20.

We have already talked about rules as the basis for a search program. But we gave little consideration to the way knowledge about the world was represented in the rules (although we can see a simple example of this in Section 4.2). In particular, we have been assuming that search control knowledge was maintained completely separately from the rules themselves. We will now relax that assumption and consider a set of rules to represent both knowledge about relationships in the world, as well as knowledge about how to solve problems using the content of the rules.

6.1 Procedural versus Declarative Knowledge

Since our discussion of knowledge representation has concentrated so far on the use of logical assertions, we use logic as a starting point in our discussion of rule-based systems.

In the previous chapter, we viewed logical assertions as declarative representations of knowledge. A *declarative representation* is one in which knowledge is specified, but the use to which that knowledge is to be put is not given. To use a declarative representation, we must augment it with a program that specifies what is to be done to the knowledge and how. For example, a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems. There is a different way, though, in which logical assertions can be viewed, namely as a *program*, rather than as *data* to a program. In this view, the implication statements define the legitimate reasoning paths and the atomic assertions provide the starting points (or, if we reason backward, the ending points) of those paths. These reasoning paths define the possible execution paths of the program in much the same way that traditional control constructs, such as *if-then-else*, define the execution paths through traditional programs. In other words, we could view logical assertions as procedural representations of knowledge. A *procedural representation* is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself. To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.

Actually, viewing logical assertions as code is not a very radical idea, given that all programs are really data to other programs that interpret (or compile) and execute them. The real difference between the declarative and the procedural views of knowledge lies in where control information resides. For example, consider the knowledge base:

$man(Marcus)$
$man(Caesar)$
$person(Cleopatra)$
$\forall x : man(x) \rightarrow person(x)$

Now consider trying to extract from this knowledge base the answer to the question

$\exists y : person(y)$

We want to bind y to a particular value for which *person* is true. Our knowledge base justifies any of the following answers:

$y = Marcus$
$y = Caesar$
$y = Cleopatra$

Because there is more than one value that satisfies the predicate, but only one value is needed, the answer to the question will depend on the order in which the assertions are examined during the search for a response. If we view the assertions as declarative, then they do not themselves say anything about how they will be examined. If we view them as procedural, then they do. Of course, nondeterministic programs are possible—for example, the concurrent and parallel programming constructs described in Dijkstra [1976], Hoare [1985], and Chandy and Misra [1989]. So, we could view these assertions as a nondeterministic program whose output is simply not defined. If we do this, then we have a "procedural" representation that actually contains no more information than does the "declarative" form. But most systems that view knowledge as procedural do not do this. The reason for this is that, at least if the procedure is to execute on any sequential or on most existing parallel machines, some decision must be made about the order in which the assertions will be examined. There is no hardware support for randomness. So if the interpreter must have a way of deciding, there is no real reason not to specify it as part of the definition of the language and thus to define the meaning of any particular program in the language. For example, we might specify that assertions will be examined in the order in which they appear in the program and that search will proceed depth-first, by which we mean that if a new subgoal is established then it will be pursued immediately and other paths will only be examined if the new one fails. If we do that, then the assertions we gave above describe a program that will answer our question with

$y = Cleopatra$

To see clearly the difference between declarative and procedural representations, consider the following assertions:

$man(Marcus)$
$man(Caesar)$
$\forall x : man(x) \rightarrow person(x)$
$person(Cleopatra)$

Viewed declaratively, this is the same knowledge base that we had before. All the same answers are supported by the system and no one of them is explicitly selected. But viewed procedurally, and using the control model we used to get *Cleopatra* as our answer before, this is a different knowledge base since now the answer to our question is *Marcus*. This happens because the first statement that can achieve the *person* goal is the inference rule $\forall x : man(x) \rightarrow person(x)$. This rule sets up a subgoal to find a man. Again the statements are examined from the beginning, and now *Marcus* is found to satisfy the subgoal and thus also the goal. So *Marcus* is reported as the answer.

It is important to keep in mind that although we have said that a procedural representation encodes control information in the knowledge base, it does so only to the extent that the interpreter for the knowledge base recognizes that control information. So we could have gotten a different answer to the *person* question by leaving our original knowledge base intact and changing the interpreter so that it examines statements from last to first (but still pursuing depth-first search). Following this control regime, we report *Caesar* as our answer.

There has been a great deal of controversy in AI over whether declarative or procedural knowledge representation frameworks are better. There is no clearcut answer to the question. As you can see from this discussion, the distinction between the two forms is often very fuzzy. Rather than try to answer the question of which approach is better, what we do in the rest of this chapter is to describe ways in which rule formalisms and interpreters can be combined to solve problems. We begin with a mechanism called *logic programming*, and then we consider more flexible structures for rule-based systems.

6.2 Logic Programming

Logic programming is a programming language paradigm in which logical assertions are viewed as programs, as described in the previous section. There are several logic programming systems in use today, the most popular of which is PROLOG [Clocksin and Mellish, 1984; Bratko, 1986]. A PROLOG program is described as a series of logical assertions, each of which is a *Horn clause*.[1] A Horn clause is a clause (as defined in Section 5.4.1) that has at most one positive literal. Thus p, $\neg p \vee q$, and $p \rightarrow q$ are all Horn clauses. The last of these does not look like a clause and it appears to have two positive literals. But recall from Section 5.4.1 that any logical expression can be converted to clause form. If we do that for this example, the resulting clause is $\neg p \vee q$,

[1]Programs written in pure PROLOG are composed only of Horn clauses. PROLOG, as an actual programming language, however, allows departures from Horn clauses. In the rest of this section, we limit our discussion to pure PROLOG.

$\forall x : pet(x) \land small(x) \rightarrow apartmentpet(x)$
$\forall x : cat(x) \lor dog(x) \rightarrow pet(x)$
$\forall x : poodle(x) \rightarrow dog(x) \land small(x)$
$poodle(fluffy)$

A Representation in Logic

```
apartmentpet(X) :- pet(X), small(X).
pet(X) :- cat(X).
pet(X) :- dog(X).
dog(X) :- poodle(X).
small(X) :- poodle(X).
poodle(fluffy).
```

A Representation in PROLOG

Figure 6.1: A Declarative and a Procedural Representation

which is a well-formed Horn clause. As we will see below, when Horn clauses are written in PROLOG programs, they actually look more like the form we started with (an implication with at most one literal on the right of the implication sign) than the clause form we just produced. Some examples of PROLOG Horn clauses appear below.

The fact that PROLOG programs are composed only of Horn clauses and not of arbitrary logical expressions has two important consequences. The first is that because of the uniform representation a simple and efficient interpreter can be written. The second consequence is even more important. The logic of Horn clause systems is decidable (unlike that of full first-order predicate logic).

The control structure that is imposed on a PROLOG program by the PROLOG interpreter is the same one we used at the beginning of this chapter to find the answers *Cleopatra* and *Marcus*. The input to a program is a goal to be proved. Backward reasoning is applied to try to prove the goal given the assertions in the program. The program is read top to bottom, left to right and search is performed depth-first with backtracking.

Figure 6.1 shows an example of a simple knowledge base represented in standard logical notation and then in PROLOG. Both of these representations contain two types of statements, *facts*, which contain only constants (i.e., no variables) and *rules*, which do contain variables. Facts represent statements about specific objects. Rules represent statements about classes of objects.

Notice that there are several superficial, syntactic differences between the logic and the PROLOG representations, including:

1. In logic, variables are explicitly quantified. In PROLOG, quantification is provided implicitly by the way the variables are interpreted (see below). The distinction between variables and constants is made in PROLOG by having all variables

begin with upper case letters and all constants begin with lower case letters or numbers.

2. In logic, there are explicit symbols for *and* ($\wedge$) and *or* ($\vee$). In PROLOG, there is an explicit symbol for *and* (,), but there is none for *or*. Instead, disjunction must be represented as a list of alternative statements, any one of which may provide the basis for a conclusion.

3. In logic, implications of the form "p implies q" are written as $p \rightarrow q$. In PROLOG, the same implication is written "backward," as `q :- p`. This form is natural in PROLOG because the interpreter always works backwards from a goal, and this form causes every rule to begin with the component that must therefore be matched first. This first component is called the *head* of the rule.

The first two of these differences arise naturally from the fact that PROLOG programs are actually sets of Horn clauses that have been transformed as follows:

1. If the Horn clause contains no negative literals (i.e., it contains a single literal which is positive), then leave it as it is.

2. Otherwise, rewrite the Horn clause as an implication, combining all of the negative literals into the antecedent of the implication and leaving the single positive literal (if there is one) as the consequent.

This procedure causes a clause, which originally consisted of a disjunction of literals (all but one of which were negative), to be transformed into a single implication whose antecedent is a conjunction of (what are now positive) literals. Further, recall that in a clause, all variables are implicitly universally quantified. But, when we apply this transformation (which essentially inverts several steps of the procedure we gave in Section 5.4.1 for converting to clause form), any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent (the head) are still universally quantified. For example, the PROLOG clause

```
P(x) :- Q(x, y)
```

is equivalent to the logical expression

$$\forall x : \exists y : Q(x, y) \rightarrow P(x)$$

A key difference between logic and the PROLOG representation is that the PROLOG interpreter has a fixed control strategy, and so the assertions in the PROLOG program define a particular search path to an answer to any question. In contrast, the logical assertions define only the set of answers that they justify; they themselves say nothing about how to choose among those answers if there are more than one.

The basic PROLOG control strategy outlined above is simple. Begin with a problem statement, which is viewed as a goal to be proved. Look for assertions that can prove the goal. Consider facts, which prove the goal directly, and also consider any rule whose head matches the goal. To decide whether a fact or a rule can be applied to the current problem, invoke a standard unification procedure (recall Section 5.4.4). Reason backward from that goal until a path is found that terminates with assertions in the program. Consider paths using a depth-first search strategy and using backtracking. At each choice point, consider options in the order in which they appear in the program. If a goal has more than one conjunctive part, prove the parts in the order in which they appear, propagating variable bindings as they are determined during unification. We can illustrate this strategy with a simple example.

Suppose the problem we are given is to find a value of X that satisfies the predicate `apartmentpet(X)`. We state this goal to PROLOG as

```
?- apartmentpet(X).
```

Think of this as the input to the program. The PROLOG interpreter begins looking for a fact with the predicate `apartmentpet` or a rule with that predicate as its head. Usually PROLOG programs are written with the facts containing a given predicate coming before the rules for that predicate so that the facts can be used immediately if they are appropriate and the rules will only be used when the desired fact is not immediately available. In this example, there are no facts with this predicate, though, so the one rule there is must be used. Since the rule will succeed if both of the clauses on its right-hand side can be satisfied, the next thing the interpreter does is to try to prove each of them. They will be tried in the order in which they appear. There are no facts with the predicate `pet` but again there are rules with it on the right-hand side. But this time there are two such rules, rather than one. All that is necessary for a proof though is that one of them succeed. They will be tried in the order in which they occur. The first will fail because there are no assertions about the predicate `cat` in the program. The second will eventually lead to success, using the rule about dogs and poodles and using the fact `poodle(fluffy)`. This results in the variable X being bound to `fluffy`. Now the second clause `small(X)` of the initial rule must be checked. Since X is now bound to fluffy, the more specific goal, `small(fluffy)`, must be proved. This too can be done by reasoning backward to the assertion `poodle(fluffy)`. The program then halts with the result `apartmentpet(fluffy)`.

Logical negation ($\neg$) cannot be represented explicitly in pure PROLOG. So, for example, it is not possible to encode directly the logical assertion

$$\forall x : dog(x) \rightarrow \neg cat(x)$$

Instead, negation is represented implicitly by the lack of an assertion. This leads to the problem-solving strategy called *negation as failure* [Clark, 1978]. If the PROLOG program of Figure 6.1 were given the goal

```
?- cat(fluffy).
```

it would return FALSE because it is unable to prove that Fluffy is a cat. Unfortunately, this program returns the same answer when given the goal even though the program knows nothing about Mittens and specifically knows nothing that might prevent Mittens from being a cat. Negation by failure requires that we make what is called the *closed world assumption*, which states that all relevant, true assertions are contained in our knowledge base or are derivable from assertions that are so contained. Any assertion that is not present can therefore be assumed to be false. This assumption, while often justified, can cause serious problems when knowledge bases are incomplete. We discuss this issue further in Chapter 7.

There is much to say on the topic of PROLOG-style versus LISP-style programming. A great advantage of logic programming is that the programmer need only specify rules and facts since a search engine is built directly into the language. The disadvantage is that the search control is fixed. Although it is possible to write PROLOG code that uses search strategies other than depth-first with backtracking, it is difficult to do so. It is even more difficult to apply domain knowledge to constrain a search. PROLOG does allow for rudimentary control of search through a non-logical operator called *cut*. A cut can be inserted into a rule to specify a point that may not be backtracked over.

More generally, the fact that PROLOG programs must be composed of a restricted set of logical operators can be viewed as a limitation of the expressiveness of the language. But the other side of the coin is that it is possible to build PROLOG compilers that produce very efficient code.

In the rest of this chapter, we retain the rule-based nature of PROLOG, but we relax a number of PROLOG's design constraints, leading to more flexible rule-based architectures.

6.3 Forward versus Backward Reasoning

The object of a search procedure is to discover a path through a problem space from an initial configuration to a goal state. While PROLOG only searches from a goal state, there are actually two directions in which such a search could proceed:

- Forward, from the start states
- Backward, from the goal states

The production system model of the search process provides an easy way of viewing forward and backward reasoning as symmetric processes. Consider the problem of solving a particular instance of the 8-puzzle. The rules to be used for solving the puzzle can be written as shown in Figure 6.2. Using those rules we could attempt to solve the puzzle shown back in Figure 2.12 in one of two ways:

- *Reason forward from the initial states.* Begin building a tree of move sequences that might be solutions by starting with the initial configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *left* sides match the root node and using their right sides to create the new configurations. Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue until a configuration that matches the goal state is generated.

Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty and Square 2 contains tile n $\rightarrow$
    Square 2 empty and Square 1 contains tile n
Square 1 empty and Square 4 contains tile n $\rightarrow$
    Square 4 empty and Square 1 contains tile n
Square 2 empty and Square 1 contains tile n $\rightarrow$
    Square 1 empty and Square 2 contains tile n

⋮

Figure 6.2: A Sample of the Rules for Solving the 8-Puzzle

- *Reason backward from the goal states.* Begin building a tree of move sequences that might be solutions by starting with the goal configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *right* sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree. Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes. Continue until a node that matches the initial state is generated. This method of reasoning backward from the desired final state is often called *goal-directed reasoning*.

Notice that the same rules can be used both to reason forward from the initial state and to reason backward from the goal state. To reason forward, the left sides (the preconditions) are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved. This continues until one of these goal states is matched by an initial state.

In the case of the 8-puzzle, it does not make much difference whether we reason forward or backward; about the same number of paths will be explored in either case. But this is not always true. Depending on the topology of the problem space, it may be significantly more efficient to search in one direction rather than the other.

Four factors influence the question of whether it is better to reason forward or backward:

- Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.

- In which direction is the branching factor (i.e., the average number of nodes that can be reached directly from a single node) greater? We would like to proceed in the direction with the lower branching factor.
- Will the program be asked to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
- What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

A few examples make these issues clearer. It seems easier to drive from an unfamiliar place home than from home to an unfamiliar place. Why is this? The branching factor is roughly the same in both directions (unless one-way streets are laid out very strangely). But for the purpose of finding our way around, there are many more locations that count as being home than there are locations that count as the unfamiliar target place. Any place from which we know how to get home can be considered as equivalent to home. If we can get to any such place, we can get home easily. But in order to find a route from where we are to an unfamiliar place, we pretty much have to be already at the unfamiliar place. So in going toward the unfamiliar place, we are aiming at a much smaller target than in going home. This suggests that if our starting position is home and our goal position is the unfamiliar place, we should plan our route by reasoning backward from the unfamiliar place.

On the other hand, consider the problem of symbolic integration. The problem space is the set of formulas, some of which contain integral expressions. The start state is a particular formula containing some integral expression. The desired goal state is a formula that is equivalent to the initial one and that does not contain any integral expressions. So we begin with a single easily identified start state and a huge number of possible goal states. Thus to solve this problem, it is better to reason forward using the rules for integration to try to generate an integral-free expression than to start with arbitrary integral-free expressions, use the rules for differentiation, and try to generate the particular integral we are trying to solve. Again we want to head toward the largest target; this time that means chaining forward.

These two examples have illustrated the importance of the relative number of start states to goal states in determining the optimal direction in which to search when the branching factor is approximately the same in both directions. When the branching factor is not the same, however, it must also be taken into account.

Consider again the problem of proving theorems in some particular domain of mathematics. Our goal state is the particular theorem to be proved. Our initial states are normally a small set of axioms. Neither of these sets is significantly bigger than the other. But consider the branching factor in each of the two directions. From a small set of axioms we can derive a very large number of theorems. On the other hand, this large number of theorems must go back to the small set of axioms. So the branching factor is significantly greater going forward from the axioms to the theorems than it is going backward from theorems to axioms. This suggests that it would be much better to reason backward when trying to prove theorems. Mathematicians have long realized this [Polya, 1957], as have the designers of theorem-proving programs.

The third factor that determines the direction in which search should proceed is the need to generate coherent justifications of the reasoning process as it proceeds. This is often crucial for the acceptance of programs for the performance of very important tasks. For example, doctors are unwilling to accept the advice of a diagnostic program that cannot explain its reasoning to the doctors' satisfaction. This issue was of concern to the designers of MYCIN [Shortliffe, 1976], a program that diagnoses infectious diseases. It reasons backward from its goal of determining the cause of a patient's illness. To do that, it uses rules that tell it such things as "If the organism has the following set of characteristics as determined by the lab results, then it is likely that it is organism x." By reasoning backward using such rules, the program can answer questions like "Why should I perform that test you just asked for?" with such answers as "Because it would help to determine whether organism x is present." (For a discussion of the explanation capabilities of MYCIN, see Chapter 20.)

Most of the search techniques described in Chapter 3 can be used to search either forward or backward. By describing the search process as the application of a set of production rules, it is easy to describe the specific search algorithms without reference to the direction of the search.[2]

We can also search both forward from the start state and backward from the goal simultaneously until two paths meet somewhere in between. This strategy is called *bidirectional search*. It seems appealing if the number of nodes at each step grows exponentially with the number of steps that have been taken. Empirical results [Pohl, 1971] suggest that for blind search, this divide-and-conquer strategy is indeed effective. Unfortunately, other results [Pohl, 1971; de Champeaux and Sint, 1977] suggest that for informed, heuristic search it is much less likely to be so. Figure 6.3 shows why bidirectional search may be ineffective. The two searches may pass each other, resulting in more work than it would have taken for one of them, on its own, to have finished. However, if individual forward and backward steps are performed as specified by a program that has been carefully constructed to exploit each in exactly those situations where it can be the most profitable, the results can be more encouraging. In fact, many successful AI applications have been written using a combination of forward and backward reasoning, and most AI programming environments provide explicit support for such hybrid reasoning.

Although in principle the same set of rules can be used for both forward and backward reasoning, in practice it has proved useful to define two classes of rules, each of which encodes a particular kind of knowledge.

- Forward rules, which encode knowledge about how to respond to certain input configurations.
- Backward rules, which encode knowledge about how to achieve particular goals.

By separating rules into these two classes, we essentially add to each rule an additional piece of information, namely how it should be used in problem solving. In the next three sections, we describe in more detail the two kinds of rule systems and how they can be combined.

[2]One exception to this is the means-ends analysis technique, described in Section 3.6, which proceeds not by making successive steps in a single direction but by reducing differences between the current and the goal states, and, as a result, sometimes reasoning backward and sometimes forward.

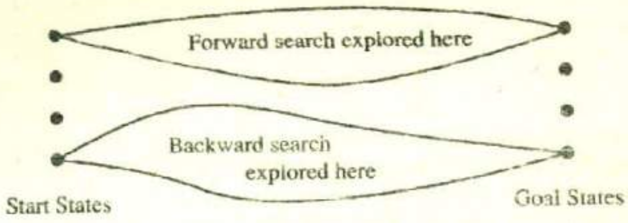


Figure 6.3: A Bad Use of Heuristic Bidirectional Search

6.3.1 Backward-Chaining Rule Systems

Backward-chaining rule systems, of which PROLOG is an example, are good for goal-directed problem solving. For example, a query system would probably use backward chaining to reason about and answer user questions.

In PROLOG, rules are restricted to Horn clauses. This allows for rapid indexing because all of the rules for deducing a given fact share the same rule head. Rules are matched with the unification procedure. Unification tries to find a set of bindings for variables to equate a (sub)goal with the head of some rule. Rules in a PROLOG program are matched in the order in which they appear.

Other backward-chaining systems allow for more complex rules. In MYCIN, for example, rules can be augmented with probabilistic certainty factors to reflect the fact that some rules are more reliable than others. We discuss this in more detail in Chapter 8.

6.3.2 Forward-Chaining Rule Systems

Instead of being directed by goals, we sometimes want to be directed by incoming data. For example, suppose you sense searing heat near your hand. You are likely to jerk your hand away. While this could be construed as goal-directed behavior, it is modeled more naturally by the recognize-act cycle characteristic of forward-chaining rule systems. In forward-chaining systems, left sides of rules are matched against the state description. Rules that match dump their right-hand side assertions into the state, and the process repeats.

Matching is typically more complex for forward-chaining systems than backward ones. For example, consider a rule that checks for some condition in the state description and then adds an assertion. After the rule fires, its conditions are probably still valid, so it could fire again immediately. However, we will need some mechanism to prevent repeated firings, especially if the state remains unchanged

While simple matching and control strategies are possible, most forward-chaining systems (e.g., OPS5 [Brownston *et al.*, 1985]) implement highly efficient matchers and supply several mechanisms for preferring one rule over another. We discuss matching in more detail in the next section.

6.3.3 Combining Forward and Backward Reasoning

Sometimes certain aspects of a problem are best handled via forward chaining and other aspects by backward chaining. Consider a forward-chaining medical diagnosis program. It might accept twenty or so facts about a patient's condition, then forward chain on those facts to try to deduce the nature and/or cause of the disease. Now suppose that at some point, the left side of a rule was *nearly* satisfied—say, nine out of ten of its preconditions were met. It might be efficient to apply backward reasoning to satisfy the tenth precondition in a directed manner, rather than wait for forward chaining to supply the fact by accident. Or perhaps the tenth condition requires further medical tests. In that case, backward chaining can be used to query the user.

Whether it is possible to use the same rules for both forward and backward reasoning also depends on the form of the rules themselves. If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left side of a rule and add to the state description the assertions on the right side. But if arbitrary procedures are allowed as the right sides of rules, then the rules will not be reversible. Some production languages allow only reversible rules; others do not. When irreversible rules are used, then a commitment to the direction of the search must be made at the time the rules are written. But, as we suggested above, this is often a useful thing to do anyway because it allows the rule writer to add control knowledge to the rules themselves.

6.4 Matching

So far, we have described the process of using search to solve problems as the application of appropriate rules to individual problem states to generate new states to which the rules can then be applied, and so forth, until a solution is found. We have suggested that clever search involves choosing from among the rules that can be applied at a particular point, the ones that are most likely to lead to a solution. But we have said little about how we extract from the entire collection of rules those that can be applied at a given point. To do so requires some kind of *matching* between the current state and the preconditions of the rules. How should this be done? The answer to this question can be critical to the success of a rule-based system. We discuss a few proposals below.

6.4.1 Indexing

One way to select applicable rules is to do a simple search through all the rules, comparing each one's preconditions to the current state and extracting all the ones that match. But there are two problems with this simple solution:

- In order to solve very interesting problems, it will be necessary to use a large number of rules. Scanning through all of them at every step of the search would be hopelessly inefficient.
- It is not always immediately obvious whether a rule's preconditions are satisfied by a particular state.

Sometimes there are easy ways to deal with the first of these problems. Instead of searching through the rules, use the current state as an index into the rules and select the

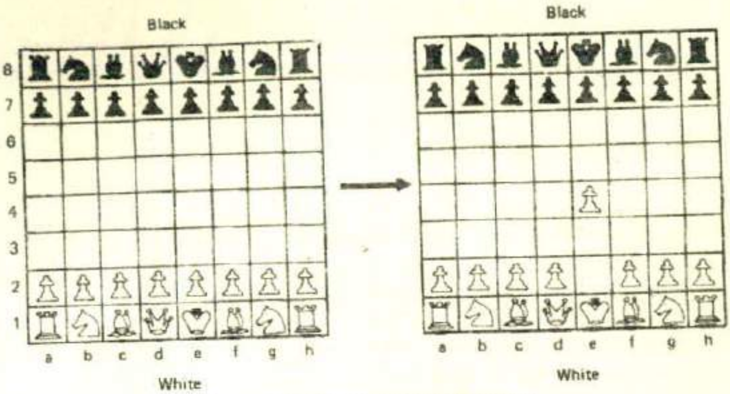


Figure 6.4: One Legal Chess Move

```
White pawn at
    Square(file e, rank 2)
        AND                          move pawn from
Square(file e, rank 3)        →      Square(file e, rank 2)
    is empty                         to Square(file e, rank 4)
        AND
Square(file e, rank 4)
    is empty
```

Figure 6.5: Another Way to Describe Chess Moves

matching ones immediately. For example, consider the legal-move generation rule for chess shown in in Figure 6.4. To be able to access the appropriate rules immediately, all we need do is assign an index to each board position. This can be done simply by treating the board description as a large number. Any reasonable hashing function can then be used to treat that number as an index into the rules. All the rules that describe a given board position will be stored under the same key and so will be found together. Unfortunately, this simple indexing scheme only works because preconditions of rules match exact board configurations. Thus the matching process is easy but at the price of complete lack of generality in the statement of the rules. As discussed in Section 2.1, it is often better to write rules in a more general form, such as that shown in Figure 6.5. When this is done, such simple indexing is not possible. In fact, there is often a trade-off between the ease of writing rules (which is increased by the use of high-level descriptions) and the simplicity of the matching process (which is decreased by such descriptions).

All of this does not mean that indexing cannot be helpful even when the preconditions of rules are stated as fairly high-level predicates. In PROLOG and many theorem-proving systems, for example, rules are indexed by the predicates they contain, so all the rules that could be applicable to proving a particular fact can be accessed fairly quickly.

In the chess example, rules can be indexed by pieces and their positions. Despite some limitations of this approach, indexing in some form is very important in the efficient operation of rule-based systems.

6.4.2 Matching with Variables

The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties (of varying complexity) that the situations must have. It often turns out that discovering whether there is a match between a particular situation and the preconditions of a given rule must itself involve a significant search process.

If we want to match a single condition against a single element in a state description, then the unification procedure of Section 5.4.4 will suffice. However, in many rule-based systems, we need to compute the whole set of rules that match the current state description. Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated *conflict resolution strategies* to choose among the applicable rules.[3] While it is possible to apply unification repeatedly over the cross product of preconditions and state description elements, it is more efficient to consider the *many-many* match problem, in which many rules are matched against many elements in the state description simultaneously.

One efficient many-many match algorithm is RETE, which gains efficiency from three major sources:

- The temporal nature of data. Rules usually do not alter the state description radically. Instead, a rule will typically add one or two elements, or perhaps delete one or two, but most of the state description remains the same. (Recall our discussion of this as part of our treatment of the frame problem in Section 4.4.) If a rule did not match in the previous cycle, it will most likely fail to apply in the current cycle. RETE maintains a network of rule conditions, and it uses changes in the state description to determine which new rules might apply (and which rules might no longer apply). Full matching is only pursued for candidates that could be affected by incoming or outgoing data.

- Structural similarity in rules. Different rules may share a large number of preconditions. For example, consider rules for identifying wild animals. One rule concludes *jaguar*(x) if *mammal*(x), *feline*(x), *carnivorous*(x), and *has-spots*(x). Another rule concludes *tiger*(x) and is identical to the first rule except that it replaces *has-spots* with *has-stripes*. If we match the two rules independently, we will repeat a lot of work unnecessarily. RETE stores the rules so that they share structures in memory; sets of conditions that appear in several rules are matched (at most) once per cycle.

- Persistence of variable binding consistency. While all the individual preconditions of a rule might be met, there may be variable binding conflicts that prevent the rule from firing. For example, suppose we know the facts *son*(*Mary*, *Joe*) and *son*(*Bill*, *Bob*). The individual preconditions of the rule

[3]Conflict resolution is discussed in the next section

> $son(x,y) \wedge son(y,z) \rightarrow grandparent(x,z)$
>
> can be matched, but not in a manner that satisfies the constraint imposed by the variable y. Fortunately, it is not necessary to compute binding consistency from scratch every time a new condition is satisfied. RETE remembers its previous calculations and is able to merge new binding information efficiently.

For more details about the RETE match algorithm, see Forgy [1982]. Other matching algorithms (e.g., Miranker [1987] and Oflazer [1987]) take different stands on how much time to spend on saving state information between cycles. They can be more or less efficient than RETE, depending on the types of rules written for the domain and on the degree of hardware parallelism available.

6.4.3 Complex and Approximate Matching

A more complex matching process is required when the preconditions of a rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others.

An even more complex matching process is required if rules should be applied if their preconditions *approximately* match the current situation. This is often the case in situations involving physical descriptions of the world. For example, a speech-understanding program must contain rules that map from a description of a physical waveform to phones (instances of English phonemes, such as *p* or *d*). There is so much variability in the physical signal, as a result of background noise, differences in the way individuals speak, and so forth, that one can hope to find only an approximate match between the rule that describes an ideal sound and the input that describes an unideal world. Approximate matching is particularly difficult to deal with because as we increase the tolerance allowed in the match, we also increase the number of rules that will match, thus increasing the size of the main search process. But approximate matching is nevertheless superior to exact matching in situations such as speech understanding, where exact matching may often result in no rules being matched and the search process coming to a grinding halt. Although symbolic techniques for approximate matching exist, there is another, very different approach that can be used to solve this problem. We discuss it in detail in Chapter 18 where we describe connectionist systems (also called neural nets).

For some problems, almost all the action is in the matching of the rules to the problem state. Once that is done, so few rules apply that the remaining search is trivial. This was the case, for example, in ELIZA [Weizenbaum, 1966], an early AI program that simulated the behavior of a Rogerian therapist. A fragment of a dialogue between ELIZA and a user is shown in Figure 6.6. ELIZA's knowledge about both English and psychology was coded in a set of simple rules. Figure 6.7 shows some ELIZA like rules.

ELIZA operated by matching the left sides of the rules against the user's last sentence and using the appropriate right side to generate a response. For example, if the user typed "My brother is mean to me," ELIZA might respond, "Who else in your family is mean to you?" or "Tell me more about your family." The rules were indexed by keywords so only a few had actually to be matched against a particular sentence. Some

Person:	Men are all alike.
ELIZA:	In what way?
Person:	They're always bugging us about something or other.
ELIZA:	Can you think of a specific example?
Person:	Well, my boyfriend made me come here.
ELIZA:	Your boyfriend made you come here.
Person:	He says I'm depressed much of the time.
ELIZA:	I am sorry to hear you are depressed.
Person:	It's true. I am unhappy.
ELIZA:	Do you think coming here will help you not to be unhappy?
Person:	I need some help, that much seems certain.
ELIZA:	What would it mean to you if you got some help?
Person:	Perhaps I could learn to get along with my mother.
ELIZA:	Tell me more about your family.

Figure 6.6: A Bit of a Dialogue with ELIZA

(X me Y)	→	(X you Y)
(I remember X)	→	(Why do remember X just now?)
(My {family-member} is Y)	→	(Who else in your family is Y?)
(X {family-member} Y)	→	(Tell me more about your family)

Figure 6.7: Some ELIZA-like rules

of the rules had no left side, so the rule could apply anywhere. These rules were used if no other rules matched and they generated replies such as "Tell me more about that." Notice that the rules themselves cause a form of approximate matching to occur. The patterns ask about specific words in the user's sentence. They do not need to match entire sentences. Thus a great variety of sentences can be matched by a single rule, and the grammatical complexity of English is pretty much ignored. This accounts both for ELIZA's major strength, its ability to say something fairly reasonable almost all of the time, and its major weakness, the superficiality of its understanding and its ability to be led completely astray. Approximate matching can easily lead to both these results.

As if the matching process were not already complicated enough, recall the frame problem mentioned in Chapter 4. One way of dealing with the frame problem is to avoid storing entire state descriptions at each node but instead to store only the changes from the previous node. If this is done, the matching process will have to be modified to scan backward from a node through its predecessors, looking for the required objects.

6.4.4 Conflict Resolution

The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable bindings were generated by the matching process. It is the job of the search method to decide on the order in which rules will be applied. But sometimes it is useful to incorporate some of that decision making into the matching process. This phase of the matching process is then called *conflict resolution*.

There are three basic approaches to the problem of conflict resolution in a production system:

- Assign a preference based on the rule that matched.
- Assign a preference based on the objects that matched.
- Assign a preference based on the action that the matched rule would perform.

Preferences Based on Rules

There are two common ways of assigning a preference based on the rules themselves. The first, and simplest, is to consider the rules to have been specified in a particular order, such as the physical order in which they are presented to the system. Then priority is given to the rules in the order in which they appear. This is the scheme used in PROLOG.

The other common rule-directed preference scheme is to give priority to special case rules over rules that are more general. We ran across this in Chapter 2, in the case of the water jug problem of Figure 2.3. Recall that rules 11 and 12 were special cases of rules 9 and 5, respectively. The purpose of such specific rules is to allow for the kind of knowledge that expert problem solvers use when they solve problems directly, without search. If we consider all rules that match, then the addition of such special-purpose rules will increase the size of the search rather than decrease it. In order to prevent that, we build the matcher so that it rejects rules that are more general than other rules that also match. How can the matcher decide that one rule is more general than another? There are a few easy ways:

- If the set of preconditions of one rule contains all the preconditions of another (plus some others), then the second rule is more general than the first.
- If the preconditions of one rule are the same as those of another except that in the first case variables are specified where in the second there are constants, then the first rule is more general than the second.

Preferences Based on Objects

Another way in which the matching process can ease the burden on the search mechanism is to order the matches it finds based on the importance of the objects that are matched. There are a variety of ways this can happen. Consider again ELIZA, which matched patterns against a user's sentence in order to find a rule to generate a reply. The patterns looked for specific combinations of important keywords. Often an input sentence contained several of the keywords that ELIZA knew. If that happened, then ELIZA made use of the fact that some keywords had been marked as being more significant than others. The pattern matcher returned the match involving the highest priority keyword. For example, ELIZA knew the word "I" as a keyword. Matching the input sentence "I know everybody laughed at me" by the keyword "I" would have enabled it to respond, "You say you know everybody laughed at you." But ELIZA also knew the word "everybody" as a keyword. Because "everybody" occurs more rarely than "I," ELIZA knows it to be more semantically significant and thus to be the clue to which it should respond. So it will produce a response such as "Who in particular are you thinking of?" Notice that priority matching such as this is particularly important if only one of the choices will ever be tried. This was true for ELIZA and would also be true, say, for a person who, when leaving a fast-burning room, must choose between turning off the lights (normally a good thing to do) and grabbing the baby (a more important thing to do).

Another form of priority matching can occur as a function of the position of the matchable objects in the current state description. For example, suppose we want to model the behavior of human short-term memory (STM). Rules can be matched against the current contents of STM and then used to generate actions, such as producing output to the environment or storing something in long-term memory. In this situation, we might like to have the matcher first try to match against the objects that have most recently entered STM and only compare against older elements if the newer elements do not trigger a match. For a discussion of this method as a conflict resolution strategy in a production system, see Newell [1973].

Preferences Based on States

Suppose that there are several rules waiting to fire. One way of selecting among them is to fire all of them temporarily and to examine the results of each. Then, using a heuristic function that can evaluate each of the resulting states, compare the merits of the results, and select the preferred one. Throw away (or maybe keep for later if necessary) the remaining ones.

This approach should look familiar—it is identical to the best-first search procedure we saw in Chapter 3. Although conceptually this approach can be thought of as a conflict resolution strategy, it is usually implemented as a search control technique that operates on top of the states generated by rule applications. The drawback to this design is that LISP-coded search control knowledge is procedural and therefore difficult to modify. Many AI search programs, especially ones that learn from their experience, represent their control strategies declaratively. The next section describes some methods for capturing knowledge about control using rules.

6.5 Control Knowledge

A major theme of this book is that while intelligent programs require search, search is computationally intractable unless it is constrained by knowledge about the world. In large knowledge bases that contain thousands of rules, the intractability of search is an overriding concern. When there are many possible paths of reasoning, it is critical that

```
Under conditions A and B,
Rules that do {not} mention X
    {at all,
    in their left-hand side
    in their right-hand side}
will
    {definitely be useless,
    probably be useless
     ...
    probably be especially useful
    definitely be especially useful}
```

Figure 6.8: Syntax for a Control Rule [Davis, 1980]

fruitless ones not be pursued. Knowledge about which paths are most likely to lead quickly to a goal state is often called *search control knowledge*. It can take many forms:

1. Knowledge about which states are more preferable to others.
2. Knowledge about which rule to apply in a given situation.
3. Knowledge about the order in which to pursue subgoals.
4. Knowledge about useful sequences of rules to apply.

In Chapter 3, we saw how the first type of knowledge could be represented with heuristic evaluation functions. There are many ways of representing the other types of control knowledge. For example, rules can be labeled and partitioned. A medical diagnosis system might have one set of rules for reasoning about bacteriological diseases and another set for immunological diseases. If the system is trying to prove a particular fact by backward chaining, it can probably eliminate one of the two rule sets, depending on what the fact is. Another method [Etzioni, 1989] is to assign cost and probability-of-success measures to rules. The problem solver can then use probabilistic decision analysis to choose a cost-effective alternative at each point in the search.

By now it should be clear that we are discussing how to represent knowledge about knowledge. For this reason, search control knowledge is sometimes called *meta-knowledge*. Davis [1980] first pointed out the need for meta-knowledge, and suggested that it be represented declaratively using rules. The syntax for one type of control rule is shown in Figure 6.8.

A number of AI systems represent their control knowledge with rules. We look briefly at two such systems, SOAR and PRODIGY.

SOAR [Laird *et al.*, 1987] is a general architecture for building intelligent systems. SOAR is based on a set of specific, cognitively motivated hypotheses about the structure of human problem solving. These hypotheses are derived from what we know about short-term memory, practice effects, etc. In SOAR:

1. Long-term memory is stored as a set of productions (or, rules).

2. Short-term memory (also called *working memory*) is a buffer that is affected by perceptions and serves as a storage area for facts deduced by rules in long-term memory. Working memory is analogous to the state description in problem solving.

3. All problem-solving activity takes place as state space traversal. There are several classes of problem-solving activities, including reasoning about which states to explore, which rules to apply in a given situation, and what effects those rules will have.

4. All intermediate and final results of problem solving are remembered (or, *chunked*) for future reference.[4]

The third feature is of most interest to us here. When SOAR is given a start state and a goal state, it sets up an initial problem space. In order to take the first step in that space, it must choose a rule from the set of applicable ones. Instead of employing a fixed conflict resolution strategy, SOAR considers that choice of rules to be a substantial problem in its own right, and it actually sets up another, auxiliary problem space. The rules that apply in this space look something like the rule shown in Figure 6.8. Operator preference rules may be very general, such as the ones described in the previous section on conflict resolution, or they may contain domain-specific knowledge.

SOAR also has rules for expressing a preference for applying a whole sequence of rules in a given situation. In learning mode, SOAR can take useful sequences and build from them more complex productions that it can apply in the future.

We can also write rules based on preferences for some states over others. Such rules can be used to implement the basic search strategies we studied in Chapters 2 and 3. For example, if we always prefer to work from the state we generated last, we will get depth-first behavior. On the other hand, if we prefer states that were generated earlier in time, we will get breadth-first behavior. If we prefer any state that looks better than the current state (according to some heuristic function), we will get hill climbing. Best-first search results when state preference rules prefer the state with the highest heuristic score. Thus we see that all of the weak methods are subsumed by an architecture that reasons with explicit search control knowledge. Different methods may be employed for different problems, and specific domain knowledge can override the more general strategies.

PRODIGY [Minton *et al.*, 1989] is a general-purpose problem-solving system that incorporates several different learning mechanisms. A good deal of the learning in PRODIGY is directed at automatically constructing a set of control rules to improve search in a particular domain. We return to PRODIGY's learning methods in Chapter 17, but we mention here a few facts that bear on the issue of search control rules. PRODIGY can acquire control rules in a number of ways:

- Through hand coding by programmers.
- Through a static analysis of the domain's operators.
- Through looking at traces of its own problem-solving behavior.

[4]We return to chunking in Chapter 17.

PRODIGY learns control rules from its experience, but unlike SOAR it also learns from its failures. If PRODIGY pursues an unfruitful path, it will try to come up with an explanation of why that path failed. It will then use that explanation to build control knowledge that will help it avoid fruitless search paths in the future.

One reason why a path may lead to difficulties is that subgoals can interact with one another. In the process of solving one subgoal, we may undo our solution of a previous subgoal. Search control knowledge can tell us something about the order in which we should pursue our subgoals. Suppose we are faced with the problem of building a piece of wooden furniture. The problem specifies that the wood must be sanded, sealed, and painted. Which of the three goals do we pursue first? To humans who have knowledge about this sort of thing, the answer is clear. An AI program, however, might decide to try painting first, since any physical object can be painted, regardless of whether it has been sanded. However, as the program plans further, it will realize that one of the effects of the sanding process is to remove the paint. The program will then be forced to plan a repainting step or else backtrack and try working on another subgoal first. Proper search control knowledge can prevent this wasted computational effort. Rules we might consider include:

- If a problem's subgoals include sanding and painting, then we should solve the sanding subgoal first.
- If subgoals include sealing and painting, then consider what the object is made of. If the object is made of wood, then we should seal it before painting it.

Before closing this section, we should touch on a couple of seemingly paradoxical issues concerning control rules. The first issue is called the *utility problem* [Minton, 1988]. As we add more and more control knowledge to a system, the system is able to search more judiciously. This cuts down on the number of nodes it expands. However, in deliberating about which step to take next in the search space, the system must consider all the control rules. If there are many control rules, simply matching them all can be very time-consuming. It is easy to reach a situation (especially in systems that generate control knowledge automatically) in which the system's problem-solving efficiency, as measured in CPU cycles, is worse with the control rules than without them. Different systems handle this problem in different ways, as demonstrated in Section 17.4.4.

The second issue concerns the complexity of the production system interpreter. As this chapter has progressed, we have seen a trend toward explicitly representing more and more knowledge about how search should proceed. We have found it useful to create meta-rules that talk about when to apply other rules. Now, a production system interpreter must know how to apply various rules and meta-rules, so we should expect that our interpreters will have to become more complex as we progress away from simple backward-chaining systems like PROLOG. And yet, moving to a declarative representation for control knowledge means that previously hand coded LISP functions can be eliminated from the interpreter. In this sense, the interpreter becomes more streamlined.

6.6 Summary

In this chapter, we have seen how to represent knowledge declaratively in rule-based systems and how to reason with that knowledge. We began with a simple mechanism, logic programming, and progressed to more complex production system models that can reason both forward and backward, apply sophisticated and efficient matching techniques, and represent their search control knowledge in rules.

In later chapters, we expand further on rule-based systems. In Chapter 7, we describe the use of rules that allow default reasoning to occur in the absence of specific counter evidence. In Chapter 8, we introduce the idea of attaching probabilistic measures to rules. And, in Chapter 20, we look at how rule-based systems are being used to solve complex, real-world problems.

The book *Pattern-Directed Inference Systems* [Waterman and Hayes-Roth, 1978] is a collection of papers describing the wide variety of uses to which production systems have been put in AI. Its introduction provides a good overview of the subject. Brownston *et al.* [1985] is an introduction to programming in production rules, with an emphasis on the OPS5 programming language.

6.7 Exercises

1. Consider the following knowledge base:

   $\forall x : \forall y : cat(x) \land fish(y) \rightarrow likes - to - eat(x, y)$
   $\forall x : calico(x) \rightarrow cat(x)$
   $\forall x : tuna(x) \rightarrow fish(x)$
   $tuna(Charlie)$
   $tuna(Herb)$
   $calico(Puss)$

   (a) Convert these wff's into Horn clauses.

   (b) Convert the Horn clauses into a PROLOG program.

   (c) Write a PROLOG query corresponding to the question, "What does Puss like to eat?" and show how it will be answered by your program.

   (d) Write another PROLOG program that corresponds to the same set of wff's but returns a different answer to the same query.

2. A problem-solving search can proceed either forward (from a known start state to a desired goal state) or backward (from a goal state to a start state). What factors determine the choice of direction for a particular problem?

3. If a problem-solving search program were to be written to solve each of the following types of problems, determine whether the search should proceed forward or backward:

   (a) water jug problem

   (b) blocks world

   (c) natural language understanding

4. Program the interpreter for a production system. You will need to build a table that holds the rules and a matcher that compares the current state to the left sides of the rules. You will also need to provide an appropriate control strategy to select among competing rules. Use your interpreter as the basis of a program that solves water jug problems.

Chapter 7

Symbolic Reasoning under Uncertainty

So far, we have described techniques for reasoning with a complete, consistent, and unchanging model of the world. Unfortunately, in many problem domains it is not possible to create such models. In this chapter and the next, we explore techniques for solving problems with incomplete and uncertain models.

7.1 Introduction to Nonmonotonic Reasoning

In their book, *The Web of Belief*, Quine and Ullian [1978] provide an excellent discussion of techniques that can be used to reason effectively even when a complete, consistent, and constant model of the world is not available. One of their examples, which we call the ABC Murder story, clearly illustrates many of the main issues that such techniques must deal with. Quoting Quine and Ullian [1978]:

> Let Abbott, Babbitt, and Cabot be suspects in a murder case. Abbott has an alibi, in the register of a respectable hotel in Albany. Babbitt also has an alibi, for his brother-in-law testified that Babbitt was visiting him in Brooklyn at the time. Cabot pleads alibi too, claiming to have been watching a ski meet in the Catskills, but we have only his word for that. So we believe
>
> (1) That Abbott did not commit the crime,
>
> (2) That Babbitt did not,
>
> (3) That Abbott or Babbitt or Cabot did.
>
> But presently Cabot documents his alibi—he had the good luck to have been caught by television in the sidelines at the ski meet. A new belief is thus thrust upon us:
>
> (4) That Cabot did not.

Our beliefs (1) through (4) are inconsistent, so we must choose one for rejection. Which has the weakest evidence? The basis for (1) in the hotel register is good, since it is a fine old hotel. The basis for (2) is weaker, since Babbitt's brother-in-law might be lying. The basis for (3) is perhaps twofold: that there is no sign of burglary and that only Abbott, Babbitt, and Cabot seem to have stood to gain from the murder apart from burglary. This exclusion of burglary seems conclusive, but the other consideration does not; there could be some fourth beneficiary. For (4), finally, the basis is conclusive: the evidence from television. Thus (2) and (3) are the weak points. To resolve the inconsistency of (1) through (4) we should reject (2) or (3), thus either incriminating Babbitt or widening our net for some new suspect.

See also how the revision progresses downward. If we reject (2), we also revise our previous underlying belief, however tentative, that the brother-in-law was telling the truth and Babbitt was in Brooklyn. If instead we reject (3), we also revise our previous underlying belief that none but Abbott, Babbitt, and Cabot stood to gain from the murder apart from burglary.

Finally a certain arbitrariness should be noted in the organization of this analysis. The inconsistent beliefs (1) through (4) were singled out, and then various further beliefs were accorded a subordinate status as underlying evidence: a belief about a hotel register, a belief about the prestige of the hotel, a belief about the television, a perhaps unwarranted belief about the veracity of the brother-in-law, and so on. We could instead have listed this full dozen of beliefs on an equal footing, appreciated that they were in contradiction, and proceeded to restore consistency by weeding them out in various ways. But the organization lightened our task. It focused our attention on four prominent beliefs among which to drop one, and then it ranged the other beliefs under these four as mere aids to choosing which of the four to drop.

The strategy illustrated would seem in general to be a good one: divide and conquer. When a set of beliefs has accumulated to the point of contradiction, find the smallest selection of them you can that still involves contradiction; for instance, (1) through (4). For we can be sure that we are going to have to drop some of the beliefs in that subset, whatever else we do. In reviewing and comparing the evidence for the beliefs in the subset, then, we will find ourselves led down in a rather systematic way to other beliefs of the set. Eventually we find ourselves dropping some of them too.

In probing the evidence, where do we stop? In probing the evidence for (1) through (4) we dredged up various underlying beliefs, but we could have probed further, seeking evidence in turn for them. In practice, the probing stops when we are satisfied how best to restore consistency: which ones to discard among the beliefs we have canvassed.

This story illustrates some of the problems posed by uncertain, fuzzy, and often changing knowledge. A variety of logical frameworks and computational methods have been proposed for handling such problems. In this chapter and the next, we discuss two approaches:

- Nonmonotonic reasoning, in which the axioms and/or the rules of inference are extended to make it possible to reason with incomplete information. These systems preserve, however, the property that, at any given moment, a statement is either believed to be true, believed to be false, or not believed to be either.

- Statistical reasoning, in which the representation is extended to allow some kind of numeric measure of certainty (rather than simply TRUE or FALSE) to be associated with each statement.

Other approaches to these issues have also been proposed and used in systems. For example, it is sometimes the case that there is not a single knowledge base that captures the beliefs of all the agents involved in solving a problem. This would happen in our murder scenario if we were to attempt to model the reasoning of Abbott, Babbitt, and Cabot, as well as that of the police investigator. To be able to do this reasoning, we would require a technique for maintaining several parallel *belief spaces*, each of which would correspond to the beliefs of one agent. Such techniques are complicated by the fact that the belief spaces of the various agents, although not identical, are sufficiently similar that it is unacceptably inefficient to represent them as completely separate knowledge bases. In Section 15.4.2 we return briefly to this issue. Meanwhile, in the rest of this chapter, we describe techniques for nonmonotonic reasoning.

Conventional reasoning systems, such first-order predicate logic, are designed to work with information that has three important properties:

- It is complete with respect to the domain of interest. In other words, all the facts that are necessary to solve a problem are present in the system or can be derived from those that are by the conventional rules of first-order logic.

- It is consistent.

- The only way it can change is that new facts can be added as they become available. If these new facts are consistent with all the other facts that have already been asserted, then nothing will ever be retracted from the set of facts that are known to be true. This property is called *monotonicity*.

Unfortunately, if any of these properties is not satisfied, conventional logic-based reasoning systems become inadequate. Nonmonotonic reasoning systems, on the other hand, are designed to be able to solve problems in which all of these properties may be missing.

In order to do this, we must address several key issues, including the following.

1. *How can the knowledge base be extended to allow inferences to be made on the basis of lack of knowledge as well as on the presence of it?* For example, we would like to be able to say things like, "If you have no reason to suspect that a particular person committed a crime, then assume he didn't," or "If you have no reason to believe that someone is not getting along with her relatives, then assume that the relatives will try to protect her." Specifically, we need to make clear the distinction between:

    - It is known that $\neg P$.

- It is not known whether P.

First-order predicate logic allows reasoning to be based on the first of these. We need an extended system that allows reasoning to be based on the second as well. In our new system, we call any inference that depends on the lack of some piece of knowledge a *nonmonotonic inference*.[1]

Allowing such reasoning has a significant impact on a knowledge base. Nonmonotonic reasoning systems derive their name from the fact that because of inferences that depend on lack of knowledge, knowledge bases may not grow monotonically as new assertions are made. Adding a new assertion may invalidate an inference that depended on the absence of that assertion. First-order predicate logic systems, on the other hand, are monotonic in this respect. As new axioms are asserted, new wff's may become provable, but no old proofs ever become invalid.

In other words, if some set of axioms T entails the truth of some statement w, then T combined with another set of axioms N also entails w. Because nonmonotonic reasoning does not share this property, it is also called *defeasible*: a nonmonotonic inference may be defeated (rendered invalid) by the addition of new information that violates assumptions that were made during the original reasoning process. It turns out, as we show below, that making this one change has a dramatic impact on the structure of the logical system itself. In particular, most of our ideas of what it means to find a proof will have to be reevaluated.

2. *How can the knowledge base be updated properly when a new fact is added to the system (or when an old one is removed)?* In particular, in nonmonotonic systems, since the addition of a fact can cause previously discovered proofs to be become invalid, how can those proofs, and all the conclusions that depend on them be found? The usual solution to this problem is to keep track of proofs, which are often called *justifications*. This makes it possible to find all the justifications that depended on the absence of the new fact, and those proofs can be marked as invalid. Interestingly, such a recording mechanism also makes it possible to support conventional, monotonic reasoning in the case where axioms must occasionally be retracted to reflect changes in the world that is being modeled. For example, it may be the case that Abbott is in town this week and so is available to testify, but if we wait until next week, he may be out of town. As a result, when we discuss techniques for maintaining valid sets of justifications, we talk both about nonmonotonic reasoning and about monotonic reasoning in a changing world.

3. *How can knowledge be used to help resolve conflicts when there are several in consistent nonmonotonic inferences that could be drawn?* It turns out that when inferences can be based on the lack of knowledge as well as on its presence, contradictions are much more likely to occur than they were in conventional logical systems in which the only possible contradictions were those that depended

[1]Recall that in Section 2.4, we also made a monotonic/nonmonotonic distinction. There the issue was classes of production systems. Although we are applying the distinction to different entities here, it is essentially the same distinction in both cases, since it distinguishes between systems that never shrink as a result of an action (monotonic ones) and ones that can (nonmonotonic ones).

on facts that were explicitly asserted to be true. In particular, in nonmonotonic systems, there are often portions of the knowledge base that are locally consistent but mutually (globally) inconsistent. As we show below, many techniques for reasoning nonmonotonically are able to define the alternatives that could be believed, but most of them provide no way to choose among the options when not all of them can be believed at once.

To do this, we require additional methods for resolving such conflicts in ways that are most appropriate for the particular problem that is being solved. For example, as soon as we conclude that Abbott, Babbitt, and Cabot all claim that they didn't commit a crime, yet we conclude that one of them must have since there's no one else who is believed to have had a motive, we have a contradiction, which we want to resolve in some particular way based on other knowledge that we have. In this case, for example, we choose to resolve the conflict by finding the person with the weakest alibi and believing that he committed the crime (which involves believing other things, such as that the chosen suspect lied).

The rest of this chapter is divided into five parts. In the first, we present several logical formalisms that provide mechanisms for performing nonmonotonic reasoning. In the last four, we discuss approaches to the implementation of such reasoning in problem-solving programs. For more detailed descriptions of many of these systems, see the papers in Ginsberg [1987].

7.2 Logics for Nonmonotonic Reasoning

Because monotonicity is fundamental to the definition of first-order predicate logic, we are forced to find some alternative to support nonmonotonic reasoning. In this section, we look at several formal approaches to doing this. We examine several because no single formalism with all the desired properties has yet emerged (although there are some attempts, e.g., Shoham [1987] and Konolige [1987], to present a unifying framework for these several theories). In particular, we would like to find a formalism that does all of the following things:

- Defines the set of possible worlds that could exist given the facts that we do have. More precisely, we will define an *interpretation* of a set of wff's to be a domain (a set of objects) D, together with a function that assigns: to each predicate, a relation (of corresponding arity); to each n-ary function, an operator that maps from D^n into D; and to each constant, an element of D. A *model* of a set of wff's is an interpretation that satisfies them. Now we can be more precise about this requirement. We require a mechanism for defining the set of models of any set of wff's we are given.

- Provides a way to say that we prefer to believe in some models rather than others.

- Provides the basis for a practical implementation of this kind of reasoning.

- Corresponds to our intuitions about how this kind of reasoning works. In other words, we do not want vagaries of syntax to have a significant impact on the conclusions that can be drawn within our system.

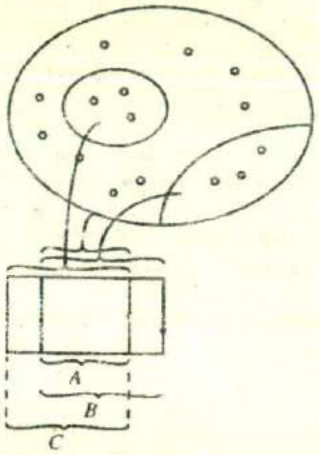


Figure 7.1: Models, Wff's, and Nonmonotonic Reasoning

As we examine each of the theories below, we need to evaluate how well they perform each of these tasks. For a more detailed discussion of these theories and some comparisons among them, see Reiter [1987a], Etherington [1988], and Genesereth and Nilsson [1987].

Before we go into specific theories in detail, let's consider Figure 7.1, which shows one way of visualizing how nonmonotonic reasoning works in all of them. The box labeled A corresponds to an original set of wff's. The large circle contains all the models of A. When we add some nonmonotonic reasoning capabilities to A, we get a new set of wff's, which we've labeled B.[2] B (usually) contains more information than A does. As a result, fewer models satisfy B than A. The set of models corresponding to B is shown at the lower right of the large circle. Now suppose we add some new wff's (representing new information) to A. We represent A with these additions as the box C. A difficulty may arise, however, if the set of models corresponding to C is as shown in the smaller, interior circle, since it is disjoint with the models for B. In order to find a new set of models that satisfy C, we need to accept models that had previously been rejected. To do that, we need to eliminate the wff's that were responsible for those models being thrown away. This is the essence of nonmonotonic reasoning.

7.2.1 Default Reasoning

We want to use nonmonotonic reasoning to perform what is commonly called *default reasoning*. We want to draw conclusions based on what is most likely to be true. In this section, we discuss two approaches to doing this.

[2] As we will see below, some techniques add inference rules, which then generate wff's, while others add wff's directly. We'll ignore that difference for the moment.

- Nonmonotonic Logic[3]
- Default Logic

We then describe two common kinds of nonmonotonic reasoning that can be defined in those logics:

- Abduction
- Inheritance

Nonmonotonic Logic

One system that provides a basis for default reasoning is *Nonmonotonic Logic* (NML) [McDermott and Doyle, 1980], in which the language of first-order predicate logic is augmented with a modal operator M, which can be read as "is consistent." For example, the formula

$$\forall x, y : Related(x, y) \land \mathrm{M}\ GetAlong(x, y) \rightarrow WillDefend(x,y)$$

should be read as, "For all x and y, if x and y are related and if the fact that x gets along with y is consistent with everything else that is believed, then conclude that x will defend y."

Once we augment our theory to allow statements of this form, one important issue must be resolved if we want our theory to be even semidecidable. (Recall that even in a standard first-order theory, the question of theoremhood is undecidable, so semidecidability is the best we can hope for.) We must define what "is consistent" means. Because consistency in this system, as in first-order predicate logic, is undecidable, we need some approximation. The one that is usually used is the PROLOG notion of negation as failure, or some variant of it. In other words, to show that P is consistent, we attempt to prove $\neg P$. If we fail, then we assume $\neg P$ to be false and we call P consistent. Unfortunately, this definition does not completely solve our problem. Negation as failure works in pure PROLOG because, if we restrict the rest of our language to Horn clauses, we have a decidable theory. So failure to prove something means that it is not entailed by our theory. If, on the other hand, we start with full first-order predicate logic as our base language, we have no such guarantee. So, as a practical matter, it may be necessary to define consistency on some heuristic basis, such as failure to prove inconsistency within some fixed level of effort.

A second problem that arises in this approach (and others, as we explain below) is what to do when multiple nonmonotonic statements, taken alone, suggest ways of augmenting our knowledge that if taken together would be inconsistent. For example, consider the following set of assertions:

$$\forall x : Republican(x) \land \mathrm{M}\ \neg Pacifist(x) \rightarrow \neg Pacifist(x)$$
$$\forall x : Quaker(x) \land \mathrm{M}\ Pacifist(x) \rightarrow Pacifist(x)$$
$$Republican(Dick)$$
$$Quaker(Dick)$$

[3]Try not to get confused about names here. We are using the terms "nonmonotonic reasoning" and "default reasoning" generically to describe a kind of reasoning. The terms "Nonmonotonic Logic" and "Default Logic" are, on the other hand, being used to refer to specific formal theories.

The definition of NML that we have given supports two distinct ways of augmenting this knowledge base. In one, we first apply the first assertion, which allows us to conclude $\neg Pacifist(Dick)$. Having done that, the second assertion cannot apply, since it is not consistent to assume $Pacifist(Dick)$. The other thing we could do, however, is apply the second assertion first. This results in the conclusion $Pacifist(Dick)$, which prevents the first one from applying. So what conclusion does the theory actually support?

The answer is that NML defines the set of theorems that can be derived from a set of wff's A to be the intersection of the sets of theorems that result from the various ways in which the wff's of A might be combined. So, in our example, no conclusion about Dick's pacifism can be derived. This theory thus takes a very conservative approach to theoremhood.

It is worth pointing out here that although assertions such as the ones we used to reason about Dick's pacifism look like rules, they are, in this theory, just ordinary wff's which can be manipulated by the standard rules for combining logical expressions. So, for example, given

$$A \wedge \mathrm{M}\,B \rightarrow B$$
$$\neg A \wedge \mathrm{M}\,B \rightarrow B$$

we can derive the expression

$$\mathrm{M}\,B \rightarrow B$$

In the original formulation of NML, the semantics of the modal operator M, which is self-referential, were unclear. A more recent system, *Autoepistemic Logic* [Moore, 1985] is very similar, but solves some of these problems.

Default Logic

An alternative logic for performing default-based reasoning is Reiter's *Default Logic* (DL) [Reiter, 1980], in which a new class of inference rules is introduced. In this approach, we allow inference rules of the form[4]

$$\frac{A : B}{C}$$

Such a rule should be read as, "If A is provable and it is consistent to assume B then conclude C." As you can see, this is very similar in intent to the nonmonotonic expressions that we used in NML. There are some important differences between the two theories, however. The first is that in DL the new inference rules are used as a basis for computing a set of plausible *extensions* to the knowledge base. Each extension corresponds to one maximal consistent augmentation of the knowledge base.[5] The logic

[4]Reiter's original notation had ":M" in place of ":", but since it conveys no additional information, the M is usually omitted.

[5]What we mean by the expression "maximal consistent augmentation" is that no additional default rules can be applied without violating consistency. But it is important to note that only expressions generated by the application of the stated inference rules to the original knowledge are allowed in an extension. Gratuitous additions are not permitted.

then admits as a theorem any expression that is valid in any extension. If a decision among the extensions is necessary to support problem solving, some other mechanism must be provided. So, for example, if we return to the case of Dick the Republican, we can compute two extensions, one corresponding to his being a pacifist and one corresponding to his not being a pacifist. The theory of DL does not say anything about how to choose between the two. But see Reiter and Criscuolo [1981], Touretzky [1986], and Rich [1983] for discussions of this issue.

A second important difference between these two theories is that, in DL, the non-monotonic expressions are rules of inference rather than expressions in the language. Thus they cannot be manipulated by the other rules of inference. This leads to some unexpected results. For example, given the two rules

$$\frac{A : B}{B} \qquad \frac{\neg A : B}{B}$$

and no assertion about A, no conclusion about B will be drawn, since neither inference rule applies.

Abduction

Standard logic performs deduction. Given two axioms:

$\forall x : A(x) \rightarrow B(x)$
$A(C)$

we can conclude $B(C)$ using deduction. But what about applying the implication in reverse? For example, suppose the axiom we have is

$\forall x : Measles(x) \rightarrow Spots(x)$

The axiom says that having measles implies having spots. But suppose we notice spots. We might like to conclude measles. Such a conclusion is not licensed by the rules of standard logic and it may be wrong, but it may be the best guess we can make about what is going on. Deriving conclusions in this way is thus another form of default reasoning. We call this specific form *abductive reasoning*. More precisely, the process of abductive reasoning can be described as, "Given two wff's $(A \rightarrow B)$ and (B), for any expressions A and B, if it is consistent to assume A, do so."

In many domains, abductive reasoning is particularly useful if some measure of certainty is attached to the resulting expressions. These certainty measures quantify the risk that the abductive reasoning process is wrong, which it will be whenever there were other antecedents besides A that could have produced B. We discuss ways of doing this in Chapter 8.

Abductive reasoning is not a kind of logic in the sense that DL and NML are. In fact, it can be described in either of them. But it is a very useful kind of nonmonotonic reasoning, and so we mentioned it explicitly here.

Inheritance

One very common use of nonmonotonic reasoning is as a basis for inheriting attribute values from a prototype description of a class to the individual entities that belong to the class. We considered one example of this kind of reasoning in Chapter 4, when we discussed the baseball knowledge base. Recall that we presented there an algorithm for implementing inheritance. We can describe informally what that algorithm does by saying, "An object inherits attribute values from all the classes of which it is a member unless doing so leads to a contradiction, in which case a value from a more restricted class has precedence over a value from a broader class." Can the logical ideas we have just been discussing provide a basis for describing this idea more formally? The answer is yes. To see how, let's return to the baseball example (as shown in Figure 4.5) and try to write its inheritable knowledge as rules in DL.

We can write a rule to account for the inheritance of a default value for the height of a baseball player as:

$$\frac{\mathit{Baseball\text{-}Player}(x) : \mathit{height}(x, 6\text{-}1)}{\mathit{height}(x, 6\text{-}1)}$$

Now suppose we assert *Pitcher(Three-Finger-Brown)*. Since this enables us to conclude that *Three-Finger-Brown* is a baseball player, our rule allows us to conclude that his height is 6-1. If, on the other hand, we had asserted a conflicting value for Three Finger's height, and if we had an axiom like

$$\forall x, y, z : \mathit{height}(x, y) \land \mathit{height}(x, z) \rightarrow y = z,$$

which prohibits someone from having more than one height, then we would not be able to apply the default rule. Thus an explicitly stated value will block the inheritance of a default value, which is exactly what we want. (We'll ignore here the order in which the assertions and the rules occur. As a logical framework, default logic does not care. We'll just assume that somehow it settles out to a consistent state in which no defaults that conflict with explicit assertions have been asserted. In Section 7.5.1 we look at issues that arise in creating an implementation that assures that.)

But now, let's encode the default rule for the height of adult males in general. If we pattern it after the one for baseball players, we get

$$\frac{\mathit{Adult\text{-}Male}(x) : \mathit{height}(x, 5\text{-}10)}{\mathit{height}(x, 5\text{-}10)}$$

Unfortunately, this rule does not work as we would like. In particular, if we again assert *Pitcher(Three-Finger-Brown)*, then the resulting theory contains two extensions: one in which our first rule fires and Brown's height is 6-1 and one in which this new rule applies and Brown's height is 5-10. Neither of these extensions is preferred. In order to state that we prefer to get a value from the more specific category, baseball player, we could rewrite the default rule for adult males in general as:

$$\frac{\mathit{Adult\text{-}Male}(x) : \neg \mathit{Baseball\text{-}Player}(x) \land \mathit{height}(x, 5\text{-}10)}{\mathit{height}(x, 5\text{-}10)}$$

This effectively blocks the application of the default knowledge about adult males in the case that more specific information from the class of baseball players is available.

Unfortunately, this approach can become unwieldy as the set of exceptions to the general rule increases. For example, we could end up with a rule like:

$$\frac{Adult\text{-}Male(x) : \neg Baseball\text{-}Player(x) \wedge \neg Midget(x) \wedge \neg Jockey(x) \wedge height(x, 5\text{-}10)}{height(x, 5\text{-}10)}$$

What we have done here is to clutter our knowledge about the general class of adult males with a list of all the known exceptions with respect to height. A clearer approach is to say something like, "Adult males typically have a height of 5-10 unless they are abnormal in some way." We can then associate with other classes the information that they are abnormal in one or another way. So we could write, for example:

$\forall x : Adult\text{-}Male(x) \wedge \neg AB(x, aspect1) \rightarrow height(x, 5\text{-}10)$
$\forall x : Baseball\text{-}Player(x) \rightarrow AB(x, aspect1)$
$\forall x : Midget(x) \rightarrow AB(x, aspect1)$
$\forall x : Jockey(x) \rightarrow AB(x, aspect1)$

Then, if we add the single default rule:

$$\frac{: \neg AB(x, y)}{\neg AB(x, y)}$$

we get the desired result.

7.2.2 Minimalist Reasoning

So far, we have talked about general methods that provide ways of describing things that are generally true. In this section we describe methods for saying a very specific and highly useful class of things that are generally true. These methods are based on some variant of the idea of a *minimal model*. Recall from the beginning of this section that a model of a set of formulas is an interpretation that satisfies them. Although there are several distinct definitions of what constitutes a minimal model, for our purposes, we will define a model to be minimal if there are no other models in which fewer things are true. (As you can probably imagine, there are technical difficulties in making this precise, many of which involve the treatment of sentences with negation.) The idea behind using minimal models as a basis for nonmonotonic reasoning about the world is the following: "There are many fewer true statements than false ones. If something is true and relevant it makes sense to assume that it has been entered into our knowledge base. Therefore, assume that the only true statements are those that necessarily must be true in order to maintain the consistency of the knowledge base." We have already mentioned (in Section 6.2) one kind of reasoning based on this idea, the PROLOG concept of negation as failure, which provides an implementation of the idea for Horn clause-based systems. In the rest of this section we look at some logical issues that arise when we remove the Horn clause limitation.

The Closed World Assumption

A simple kind of minimalist reasoning is suggested by the *Closed World Assumption* or CWA [Reiter, 1978]. The CWA says that the only objects that satisfy any predicate P are those that must. The CWA is particularly powerful as a basis for reasoning with databases, which are assumed to be complete with respect to the properties they describe. For example, a personnel database can safely be assumed to list all of the company's employees. If someone asks whether Smith works for the company, we should reply "no" unless he is explicitly listed as an employee. Similarly, an airline database can be assumed to contain a complete list of all the routes flown by that airline. So if I ask if there is a direct flight from Oshkosh to El Paso, the answer should be "no" if none can be found in the database. The CWA is also useful as a way to deal with *AB* predicates, of the sort we introduced in Section 7.2.1, since we want to take as abnormal only those things that are asserted to be so.

Although the CWA is both simple and powerful, it can fail to produce an appropriate answer for either of two reasons. The first is that its assumptions are not always true in the world; some parts of the world are not realistically "closable." We saw this problem in the murder story example. There were facts that were relevant to the investigation that had not yet been uncovered and so were not present in the knowledge base. The CWA will yield appropriate results exactly to the extent that the assumption that all the relevant positive facts are present in the knowledge base is true.

The second kind of problem that plagues the CWA arises from the fact that it is a purely syntactic reasoning process. Thus, as you would expect, its results depend on the form of the assertions that are provided. Let's look at two specific examples of this problem.

Consider a knowledge base that consists of just a single statement:

$A(Joe) \vee B(Joe)$

The CWA allows us to conclude both $\neg A(Joe)$ and $\neg B(Joe)$, since neither *A* nor *B* must necessarily be true of Joe. Unfortunately, the resulting extended knowledge base

$A(Joe) \vee B(Joe)$
$\neg A(Joe)$
$\neg B(Joe)$

is inconsistent.

The problem is that we have assigned a special status to positive instances of predicates, as opposed to negative ones. Specifically, the CWA forces completion of a knowledge base by adding the negative assertion $\neg P$ whenever it is consistent to do so. But the assignment of a real world property to some predicate *P* and its complement to the negation of *P* may be arbitrary. For example, suppose we define a predicate *Single* and create the following knowledge base:

Single(*John*)
Single(*Mary*)

Then, if we ask about Jane, the CWA will yield the answer $\neg$*Single*(*Jane*). But now suppose we had chosen instead to use the predicate *Married* rather than *Single*. Then the corresponding knowledge base would be

$\neg$*Married*(*John*)
$\neg$*Married*(*Mary*)

If we now ask about Jane, the CWA will yield the result $\neg Married(Jane)$.

Circumscription

Although the CWA captures part of the idea that anything that must not necessarily be true should be assumed to be false, it does not capture all of it. It has two essential limitations:

- It operates on individual predicates without considering the interactions among predicates that are defined in the knowledge base. We saw an example of this above when we considered the statement $A(Joe) \vee B(Joe)$.
- It assumes that all predicates have all of their instances listed. Although in many database applications this is true, in many knowledge-based systems it is not. Some predicates can reasonably be assumed to be completely defined (i.e., the part of the world they describe is closed), but others cannot (i.e., the part of the world they describe is open). For example, the predicate *has-a-green-shirt* should probably be considered open since in most situations it would not be safe to assume that one has been told all the details of everyone else's wardrobe.

Several theories of *circumscription* (e.g., McCarthy [1980], McCarthy [1986], and Lifschitz [1985]) have been proposed to deal with these problems. In all of these theories, new axioms are added to the existing knowledge base. The effect of these axioms is to force a minimal interpretation on a selected portion of the knowledge base. In particular, each specific axiom describes a way that the set of values for which a particular axiom of the original theory is true is to be delimited (i.e., circumscribed).

As an example, suppose we have the simple assertion

$$\forall x : Adult(x) \wedge \neg AB(x, aspect1) \rightarrow Literate(x)$$

We would like to circumscribe *AB*, since we would like it to apply only to those individuals to which it applies. In essence, what we want to do is to say something about what the predicate *AB* must be (since at this point we have no idea what it is; all we know is its name). To know what it is, we need to know for what values it is true. Even though we may know a few values for which it is true (if any individuals have been asserted to be abnormal in this way), there are many different predicates that would be consistent with what we know so far. Imagine this universe of possible binary predicates. We might ask, which of these predicates could be *AB*? We want to say that *AB* can only be one of the predicates that is true only for those objects that we know it must be true for. We can do this by adding a (second order) axiom that says that *AB* is the smallest predicate that is consistent with our existing knowledge base.

In this simple example, circumscription yields the same result as does the CWA since there are no other assertions in the knowledge base with which a minimization of *AB* must be consistent. In both cases, the only models that are admitted are ones in which there are no individuals who are abnormal in *aspect1*. In other words, *AB* must be the predicate FALSE.

But, now let's return to the example knowledge base

$A(Joe) \vee B(Joe)$

If we circumscribe only A, then this assertion describes exactly those models in which A is true of no one and B is true of at least *Joe*. Similarly, if we circumscribe only B, then we will accept exactly those models in which B is true of no one and A is true of at least *Joe*. If we circumscribe A and B together, then we will admit only those models in which A is true of only *Joe* and B is true of no one or those in which B is true of only *Joe* and A is true of no one. Thus, unlike the CWA, circumscription allows us to describe the logical relationship between A and B.

7.3 Implementation Issues

Although the logical frameworks that we have just discussed take us part of the way toward a basis for implementing nonmonotonic reasoning in problem-solving programs, they are not enough. As we have seen, they all have some weaknesses as logical systems. In addition, they fail to deal with four important problems that arise in real systems.

The first is how to derive exactly those nonmonotonic conclusions that are relevant to solving the problem at hand while not wasting time on those that, while they may be licensed by the logic, are not necessary and are not worth spending time on.

The second problem is how to update our knowledge incrementally as problem-solving progresses. The definitions of the logical systems tell us how to decide on the truth status of a proposition with respect to a given truth status of the rest of the knowledge base. Since the procedure for doing this is a global one (relying on some form of consistency or minimality), any change to the knowledge base may have far-reaching consequences. It would be computationally intractable to handle this problem by starting over with just the facts that are explicitly stated and reapplying the various nonmonotonic reasoning steps that were used before, this time deriving possibly different results.

The third problem is that in nonmonotonic reasoning systems, it often happens that more than one interpretation of the known facts is licensed by the available inference rules. In Reiter's terminology, a given nonmonotonic system may (and often does) have several extensions at the moment, even though many of them will eventually be eliminated as new knowledge becomes available. Thus some kind of search process is necessary. How should it be managed?

The final problem is that, in general, these theories are not computationally effective. None of them is decidable. Some are semidecidable, but only in their propositional forms. And none is efficient.

In the rest of this chapter, we discuss several computational solutions to these problems. In all of these systems, the reasoning process is separated into two parts: a problem solver that uses whatever mechanism it happens to have to draw conclusions as necessary and a truth maintenance system whose job is just to do the bookkeeping required to provide a solution to our second problem. The various logical issues we have been discussing, as well as the heuristic ones we have raised here are issues in the design of the problem solver. We discuss these issues in Section 7.4. Then in the following sections, we describe techniques for tracking nonmonotonic inferences so that changes to the knowledge base are handled properly. Techniques for doing this can be divided into two classes, determined by their approach to the search control problem:

- Depth-first, in which we follow a single, most likely path until some new piece of information comes in that forces us to give up this path and find another.
- Breadth-first, in which we consider all the possibilities as equally likely. We consider them as a group, eliminating some of them as new facts become available. Eventually, it may happen that only one (or a small number) turn out to be consistent with everything we come to know

It is important to keep in mind throughout the rest of this discussion that there is no exact correspondence between any of the logics that we have described and any of the implementations that we will present. Unfortunately, the details of how the two can be brought together are still unknown.

7.4 Augmenting a Problem Solver

So far, we have described a variety of logical formalisms, all of which describe the theorems that can be derived from a set of axioms. We have said nothing about how we might write a program that solves problems using those axioms. In this section, we do that.

As we have already discussed several times, problem solving can be done using either forward or backward reasoning. Problem solving using uncertain knowledge is no exception. As a result, there are two basic approaches to this kind of problem solving (as well as a variety of hybrids):

- Reason forward from what is known. Treat nonmonotonically derivable conclusions the same way monotonically derivable ones are handled. Nonmonotonic reasoning systems that support this kind of reasoning allow standard forward-chaining rules to be augmented with *unless* clauses, which introduce a basis for reasoning by default. Control (including deciding which default interpretation to choose) is handled in the same way that all other control decisions in the systen. are made (whatever that may be, for example, via rule ordering or the use of metarules).
- Reason backward to determine whether some expression *P* is true (or perhaps to find a set of bindings for its variables that make it true). Nonmonotonic reasoning systems that support this kind of reasoning may do either or both of the following two things:
  - Allow default (unless) clauses in backward rules. Resolve conflicts among defaults using the same control strategy that is used for other kinds of reasoning (usually rule ordering).
  - Support a kind of debate in which an attempt is made to construct arguments both in favor of *P* and opposed to it. Then some additional knowledge is applied to the arguments to determine which side has the stronger case.

Let's look at backward reasoning first. We will begin with the simple case of backward reasoning in which we attempt to prove (and possibly to find bindings for)

$Suspect(x) \leftarrow Beneficiary(x)$
UNLESS $Alibi(x)$

$Alibi(x) \leftarrow SomewhereElse(x)$

$SomewhereElse(x) \leftarrow RegisteredHotel(x, y)$ and $FarAway(y)$
UNLESS $ForgedRegister(y)$

$Alibi(x) \leftarrow Defends(x, y)$
UNLESS $Lies(y)$

$SomewhereElse(x) \leftarrow PictureOf(x, y)$ and $FarAway(y)$

$Contradiction() \leftarrow$ TRUE
UNLESS $\exists x$: $Suspect(x)$

$Beneficiary(Abbott)$
$Beneficiary(Babbitt)$
$Beneficiary(Cabot)$

Figure 7.2: Backward Rules Using UNLESS

an expression P. Suppose that we have a knowledge base that consists of the backward rules shown in Figure 7.2.

Assume that the problem solver that is using this knowledge base uses the usual PROLOG-style control structure in which rules are matched top to bottom, left to right. Then if we ask the question ?$Suspect(x)$, the program will first try Abbott, who is a fine suspect given what we know now, so it will return Abbott as its answer. If we had also included the facts

$RegisteredHotel(Abbott, Albany)$
$FarAway(Albany)$

then, the program would have failed to conclude that Abbott was a suspect and it would instead have located Babbitt.

As an alternative to this approach, consider the idea of a debate. In debating systems, an attempt is made to find multiple answers. In the ABC Murder story case, for example, all three possible suspects would be considered. Then some attempt to choose among the arguments would be made. In this case, for example, we might want to have a choice rule that says that it is more likely that people will lie to defend themselves than to defend others. We might have a second rule that says that we prefer to believe hotel registers rather than people. Using these two rules, a problem solver would conclude that the most likely suspect is Cabot.

Backward rules work exactly as we have described if all of the required facts are present when the rules are invoked. But what if we begin with the situation shown in Figure 7.2 and conclude that Abbott is our suspect. Later, we are told that he was

If: *Beneficiary*(*x*),
  UNLESS *Alibi*(*x*),
then *Suspect*(*x*)

If: *SomewhereElse*(*x*),
then *Alibi*(*x*)

If: *RegisteredHotel*(*x*, *y*), and
  FarAway(*y*),
  UNLESS *ForgedRegister*(*y*),
then *SomewhereElse*(*x*)

If *Defends*(*x*, *y*),
  UNLESS *Lies*(*y*),
then *Alibi*(*x*)

If *PictureOf*(*x*, *y*), and
  FarAway(*y*),
then *SomewhereElse*(*x*)

If TRUE,
  UNLESS $\exists x$: ¬*Suspect*(*x*)
then *Contradiction*()

Beneficiary(*Abbott*)
Beneficiary(*Babbitt*)
Beneficiary(*Cabot*)

Figure 7.3: Forward Rules Using UNLESS

registered at a hotel in Albany. Backward rules will never notice that anything has changed. To make our system data-driven, we need to use forward rules. Figure 7.3 shows how the same knowledge could be represented as forward rules. Of course, what we probably want is a system that can exploit both. In such a system, we could use a backward rule whose goal is to find a suspect, coupled with forward rules that fire as new facts that are relevant to finding a suspect appear.

7.5 Implementation: Depth-First Search

7.5.1 Dependency-Directed Backtracking

If we take a depth-first approach to nonmonotonic reasoning, then the following scenario is likely to occur often: We need to know a fact, F, which cannot be derived monotonically from what we already know, but which can be derived by making some assumption A which seems plausible. So we make assumption A, derive F, and then derive some additional facts G and H from F. We later derive some other facts M and N, but they are completely independent of A and F. A little while later, a new fact comes in that invalidates A. We need to rescind our proof of F, and also our proofs of G and H since they depended on F. But what about M and N? They didn't depend on F, so there is no logical need to invalidate them. But if we use a conventional backtracking scheme, we have to back up past conclusions in the order in which we derived them. So we have to backup past M and N, thus undoing them, in order to get back to F, G, H and A. To get around this problem, we need a slightly different notion of backtracking, one that is based on logical dependencies rather than the chronological order in which decisions were made. We call this new method *dependency-directed backtracking* [Stallman and Sussman, 1977], in contrast to *chronological backtracking*, which we have been using up until now.

Before we go into detail on how dependency-directed backtracking works, it is worth pointing out that although one of the big motivations for it is in handling nonmonotonic reasoning, it turns out to be useful for conventional search programs as well. This is not too surprising when you consider that what any depth-first search program does is to "make a guess" at something, thus creating a branch in the search space. If that branch eventually dies out, then we know that at least one guess that led to it must be wrong. It could be any guess along the branch. In chronological backtracking we have to assume it was the most recent guess and back up there to try an alternative. Sometimes, though, we have additional information that tells us which guess caused the problem. We'd like to retract only that guess and the work that explicitly depended on it, leaving everything else that has happened in the meantime intact. This is exactly what dependency-directed backtracking does.

As an example, suppose we want to build a program that generates a solution to a fairly simple problem, such as finding a time at which three busy people can all attend a meeting. One way to solve such a problem is first to make an assumption that the meeting will be held on some particular day, say Wednesday, add to the database an assertion to that effect, suitably tagged as an assumption, and then proceed to find a time, checking along the way for any inconsistencies in people's schedules. If a conflict arises, the statement representing the assumption must be discarded and replaced by another, hopefully noncontradictory, one. But, of course, any statements that have been generated along the way that depend on the now-discarded assumption must also be discarded.

Of course, this kind of situation can be handled by a straightforward tree search with chronological backtracking. All assumptions, as well as the inferences drawn from them, are recorded at the search node that created them. When a node is determined to represent a contradiction, simply backtrack to the next node from which there remain unexplored paths. The assumptions and their inferences will disappear automatically. The drawback to this approach is illustrated in Figure 7.4, which shows part of the search tree of a program that is trying to schedule a meeting. To do so, the program must solve a constraint satisfaction problem to find a day and time at which none of the participants is busy and at which there is a sufficiently large room available.

In order to solve the problem, the system must try to satisfy one constraint at a time. Initially, there is little reason to choose one alternative over another, so it decides to schedule the meeting on Wednesday. That creates a new constraint that must be met by the rest of the solution. The assumption that the meeting will be held on Wednesday

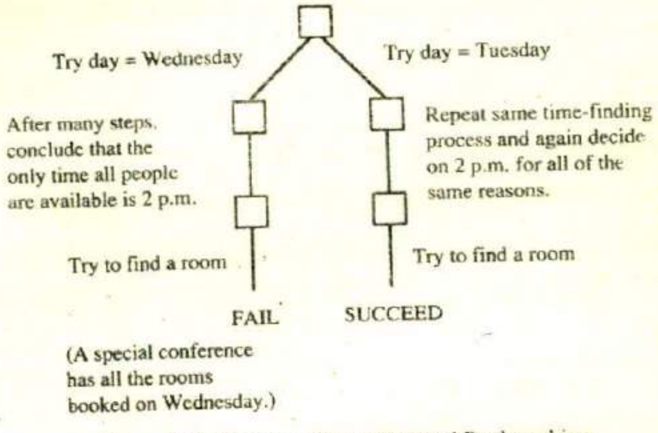

Try day = Wednesday
Try day = Tuesday
After many steps, conclude that the only time all people are available is 2 p.m.
Repeat same time-finding process and again decide on 2 p.m. for all of the same reasons.
Try to find a room
Try to find a room
FAIL
SUCCEED
(A special conference has all the rooms booked on Wednesday.)

Figure 7.4: Nondependency-Directed Backtracking

is stored at the node it generated. Next the program tries to select a time at which all participants are available. Among them, they have regularly scheduled daily meetings at all times except 2:00. So 2:00 is chosen as the meeting time. But it would not have mattered which day was chosen. Then the program discovers that on Wednesday there are no rooms available. So it backtracks past the assumption that the day would be Wednesday and tries another day, Tuesday. Now it must duplicate the chain of reasoning that led it to choose 2:00 as the time because that reasoning was lost when it backtracked to redo the choice of day. This occurred even though that reasoning did not depend in any way on the assumption that the day would be Wednesday. By withdrawing statements based on the order in which they were generated by the search process rather than on the basis of responsibility for inconsistency, we may waste a great deal of effort.

If we want to use dependency-directed backtracking instead, so that we do not waste this effort, then we need to do the following things:

- Associate with each node one or more justifications. Each justification corresponds to a derivation process that led to the node. (Since it is possible to derive the same node in several different ways, we want to allow for the possibility of multiple justifications.) Each justification must contain a list of all the nodes (facts, rules, assumptions) on which its derivation depended.

- Provide a mechanism that, when given a contradiction node and its justification, computes the "no-good" set of assumptions that underlie the justification. The no-good set is defined to be the minimal set of assumptions such that if you remove any element from the set, the justification will no longer be valid and the inconsistent node will no longer be believed.

- Provide a mechanism for considering a no-good set and choosing an assumption to retract.
- Provide a mechanism for propagating the result of retracting an assumption. This mechanism must cause all of the justifications that depended, however indirectly, on the retracted assumption to become invalid.

In the next two sections, we will describe two approaches to providing such a system.

7.5.2 Justification-Based Truth Maintenance Systems

The idea of a truth maintenance system or TMS [Doyle, 1979] arose as a way of providing the ability to do dependency-directed backtracking and so to support nonmonotonic reasoning. There was a later attempt to rename it to Reason Maintenance System (a bit less pretentious), but since the old name has stuck, we use it here.

A TMS allows assertions to be connected via a spreadsheet-like network of dependencies. In this section, we describe a simple form of truth maintenance system, a justification-based truth maintenance system (or JTMS). In a JTMS (or just TMS for the rest of this section), the TMS itself does not know anything about the structure of the assertions themselves. (As a result, in our examples, we use an English-like shorthand for representing the contents of nodes.) The TMS's only role is to serve as a bookkeeper for a separate problem-solving system, which in turn provides it with both assertions and dependencies among assertions.

To see how a TMS works, let's return to the ABC Murder story. Initially, we might believe that Abbott is the primary suspect because he was a beneficiary of the deceased and he had no alibi. There are three assertions here, a specific combination of which we now believe, although we may change our beliefs later. We can represent these assertions in shorthand as follows:

- *Suspect Abbott* (Abbott is the primary murder suspect.)
- *Beneficiary Abbott* (Abbott is a beneficiary of the victim.)
- *Alibi Abbott* (Abbott was at an Albany hotel at the time.)

Our reason for possible belief that Abbott is the murderer is nonmonotonic. In the notation of Default Logic, we can state the rule that produced it as

$$\frac{Beneficiary(x) : \neg Alibi(x)}{Suspect(x)}$$

or we can write it as a backward rule as we did in Section 7.4.

If we currently believe that he is a beneficiary and we have no reason to believe he has a valid alibi, then we will believe that he is our suspect. But if later we come to believe that he does have a valid alibi, we will no longer believe Abbott is a suspect.

But how should belief be represented and how should this change in belief be enforced? There are various *ad hoc* ways we might do this in a rule-based system. But they would all require a developer to construct rules carefully for each possible change in belief. For instance, we would have to have a rule that said that if Abbott ever gets

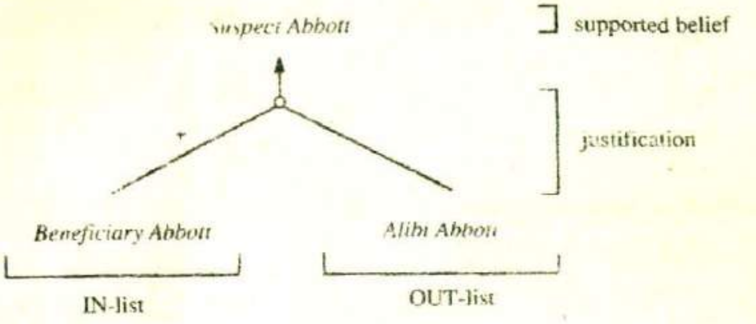


Figure 7.5: A Justification

an alibi, then we should erase from the database the belief that Abbott is a suspect. But suppose that we later fire a rule that erases belief in Abbott's alibi. Then we need another rule that would reconclude that Abbott is a suspect. The task of creating a rule set that consistently maintains beliefs when new assertions are added to the database quickly becomes unmanageable. In contrast, a TMS dependency network offers a purely syntactic, domain-independent way to represent belief and change it consistently.

Figure 7.5 shows how these three facts would be represented in a dependency network, which can be created as a result of applying the first rule of either Figure 7.2 or Figure 7.3. The assertion *Suspect Abbott* has an associated TMS *justification*. Each justification consists of two parts: an *IN-list* and an *OUT-list*. In the figure, the assertions on the IN-list are connected to the justification by "+" links, those on the OUT-list by "–" links. The justification is connected by an arrow to the assertion that it supports. In the justification shown, there is exactly one assertion in each list. *Beneficiary Abbott* is in the IN-list and *Alibi Abbott* is in the OUT-list. Such a justification says that Abbott should be a suspect just when it is believed that he is a beneficiary and it is not believed that he has an alibi.

More generally, assertions (usually called nodes) in a TMS dependency network are believed when they have a valid justification. A justification is *valid* if every assertion in the IN-list is believed and none of those in the OUT-list is. A justification is nonmonotonic if its OUT-list is not empty, or, recursively, if any assertion in its IN-list has a nonmonotonic justification. Otherwise, it is monotonic. In a TMS network, nodes are labeled with a *belief status*. If the assertion corresponding to the node should be believed, then in the TMS it is labeled IN. If there is no good reason to believe the assertion, then it is labeled OUT. What does it mean that an assertion "should be believed" or has no "good" reason for belief?

A TMS answers these questions for a dependency network in a way that is independent of any interpretation of the assertions associated with the nodes. The *labeling* task of a TMS is to label each node so that two criteria about the dependency network

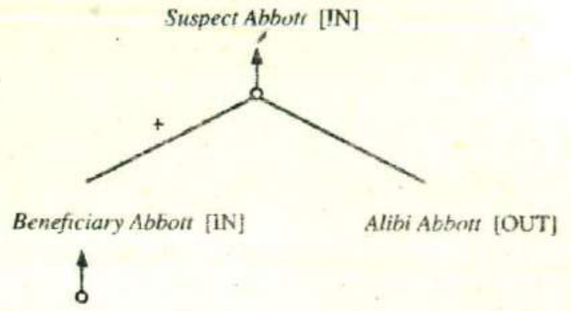


Figure 7.6: Labeled Nodes with Premise Justification

structure are met. The first criterion is *consistency*: every node labeled IN is supported by at least one valid justification and all other nodes are labeled OUT. More specifically than before a justification is valid if every node in its IN-list is labeled IN and every node in its OUT-list is labeled OUT. Notice that in Figure 7.5, all of the assertions would have to be labeled OUT to be consistent. *Alibi Abbott* has no justification at all, much less a valid one, and so must be labeled OUT. But the same is true for *Beneficiary Abbott*, so it must be OUT as well. Then the justification for *Suspect Abbott* is invalid because an element of its IN-list is labeled OUT. *Suspect Abbott* would then be labeled OUT as well. Thus status labels correspond to our belief or lack of it in assertions, and justifications correspond to our reasons for such belief, with valid justifications being our "good" reasons. Notice that the label OUT may indicate that we have specific reason to believe that a node represents an assertion that is not true, or it may mean simply that we have no information one way or the other.

But the state of affairs in Figure 7.5 is incomplete. We are told that Abbott is a beneficiary. We have no further justification for this fact; we must simply accept it. For such facts, we give a *premise* justification: a justification with empty IN- and OUT-lists. Premise justifications are always valid. Figure 7.6 shows such a justification added to the network and a consistent labeling for that network, which shows *Suspect Abbott* labeled IN.

That Abbot is the primary suspect represents an initial state of the murder investigation. Subsequently, the detective establishes that Abbott is listed on the register of a good Albany hotel on the day of the murder. This provides a valid reason to believe Abbott's alibi. Figure 7.7 shows the effect of adding such a justification to the network, assuming that we have used forward (data-driven) rules as shown in Figure 7.3 for all of our reasoning except possibly establishing the top-level goal. That Abbott was registered at the hotel, *Registered Abbott*, was told to us and has a premise justification and so is labeled IN. That the hotel is far away is also asserted as a premise. The register might have been forged, but we have no good reason to believe it was. Thus

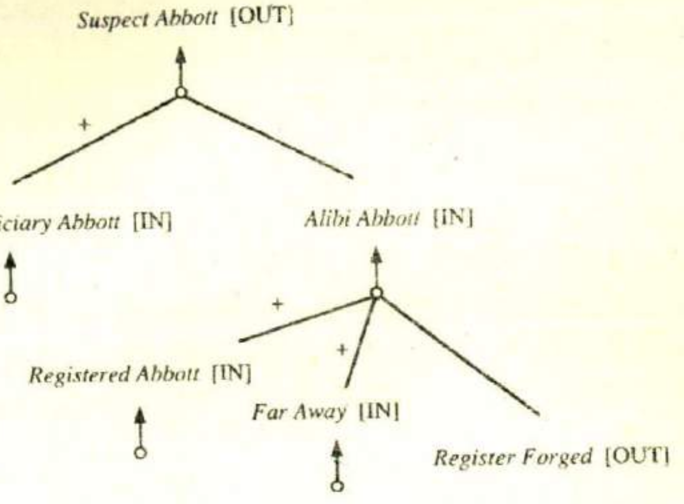


Figure 7.7: Changed Labeling

Register Forged lacks any justification and is labeled OUT. That Abbott was on the register of a far away hotel and the lack of belief that the register was forged will cause the appropriate forward rule to fire and create a justification for *Alibi Abbott*, which is thus labeled IN. This means that *Suspect Abbott* no longer has a valid justification and must be labeled OUT. Abbott is no longer a suspect.

Notice that such a TMS labeling carefully avoids saying that the register definitely was *not* forged. It only says that there is currently no good reason to believe that it was. Just like our original reason for believing that Abbott was a suspect, this is a nonmonotonic justification. Later, if we find that Abbott was secretly married to the desk clerk, we might add to this network a justification that would reverse some of the labeling. Babbitt will have a similar justification based upon lack of belief that his brother-in-law lied as shown in Figure 7.8 (where *B-I-L* stands for "Brother-In-Law").

Abbott's changing state showed how consistency was maintained. There is another criterion that the TMS must meet in labeling a dependency network: *well-foundedness* (i.e., the proper grounding of a chain of justifications on a set of nodes that do not themselves depend on the nodes they support). To illustrate this, consider poor Cabot. Not only does he have fewer *b*s and *t*s in his name, he also lacks a valid justification for his alibi that he was at a ski show. We have only his word that he was. Ignoring the more complicated representation of lying, the simple dependency network in Figure 7.9 illustrates the fact that the only support for the alibi of attending the ski show is that Cabot is telling the truth about being there. The only support for his telling the truth would be if we knew he was at the ski show. But this is a circular argument. Part of the task of a TMS is to disallow such arguments. In particular, if the support for a node only depends on an unbroken chain of positive links (IN-list links) leading back to itself

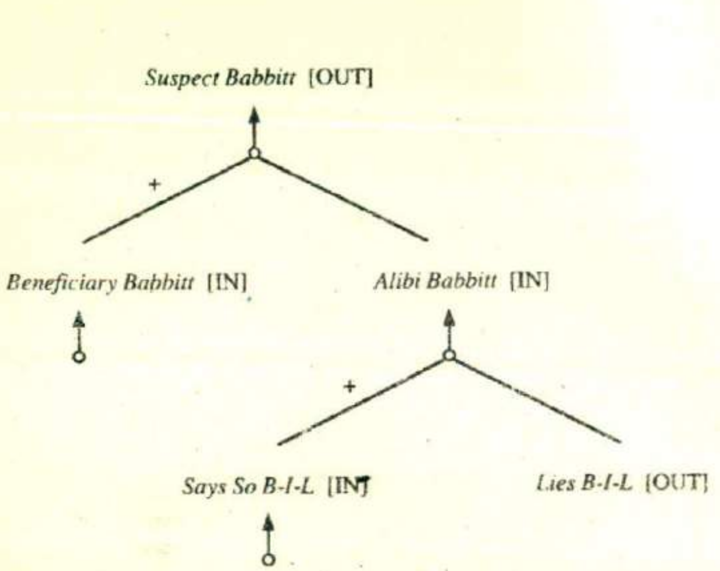


Figure 7.8: Babbitt's Justification

then that node must be labeled OUT if the labeling is to be well-founded.

The TMS task of ensuring a consistent, well-founded labeling has now been outlined. The other major task of a TMS is resolving contradictions. In a TMS, a contradiction node does not represent a logical contradiction but rather a state of the database explicitly declared to be undesirable. (In the next section, we describe a slightly different kind of TMS in which this is not the case.) In our example, we have a contradiction if we do not have at least one murder suspect. Thus a contradiction might have the justification shown in Figure 7.10, where the node *Other Suspects* means that there are suspects other than Abbott, Babbitt, and Cabot. This is one way of explicitly representing an instance of the closed world assumption. Later, if we discover a long-lost relative, this will provide a valid justification for *Other Suspects*. But for now, it has none and must be labeled OUT. Fortunately, even though Abbott and Babbitt are not suspects, *Suspect Cabot* is labeled IN, invalidating the justification for the contradiction. While the contradiction is labeled OUT, there is no contradiction to resolve.

Now we learn that Cabot was seen on television attending the ski tournament. Adding this to the dependency network first illustrates the fact that nodes can have more than one justification as shown in Figure 7.11. Not only does Cabot say he was at the ski slopes, but he was seen there on television, and we have no reason to believe that this was an elaborate forgery. This new valid justification of *Alibi Cabot* causes it to be labeled IN (which also causes *Tells Truth Cabot* to come IN). This change in state propagates to *Suspect Cabot*, which goes OUT. Now we have a problem.

The justification for the contradiction is now valid and the contradiction is IN. The job of the TMS at this point is to determine how the contradiction can be made OUT again. In a TMS network, a node can be made OUT by causing all of its justifications

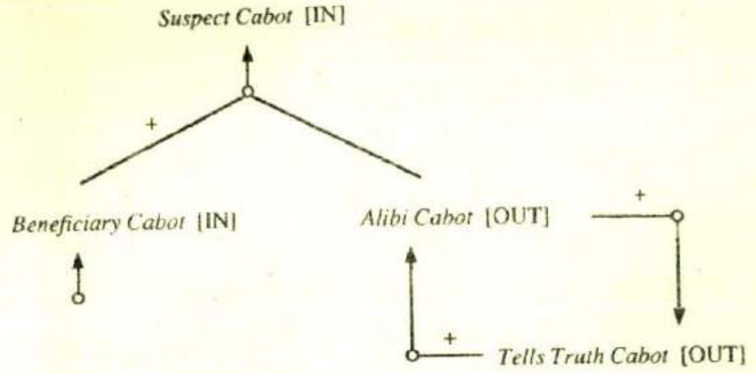


Figure 7.9: Cabot's Justification

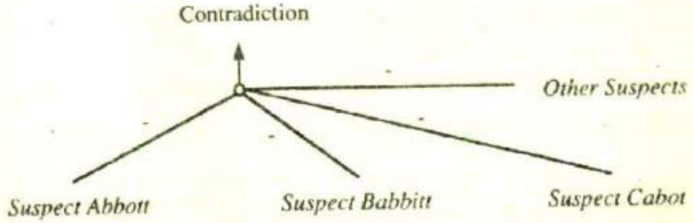


Figure 7.10: A Contradiction

to become invalid. Monotonic justifications cannot be made invalid without retracting explicit assertions that have been made to the network. Nonmonotonic justifications can, however, be invalidated by asserting some fact whose absence is required by the justification. We call assertions with nonmonotonic justifications *assumptions*. An assumption can be retracted by making IN some element of its justification's OUT-list (or recursively in some element of the OUT-list of the justification of some element in its IN-list). Unfortunately, there may be many such assumptions in a large dependency network. Fortunately, the network gives us a way to identify those that are relevant to the contradiction at hand. Dependency-directed backtracking algorithms, of the sort we described in Section 7.5.1, can use the dependency links to determine an AND/OR tree of assumptions that might be retracted and ways to retract them by justifying other beliefs.

In Figure 7.10, we see that the contradiction itself is an assumption whenever its justification is valid. We might retract it by believing there were other suspects or by finding a way to believe again that either Abbott, Babbitt, or Cabot was a suspect. Each of the last three could be believed if we disbelieved their alibis, which in turn

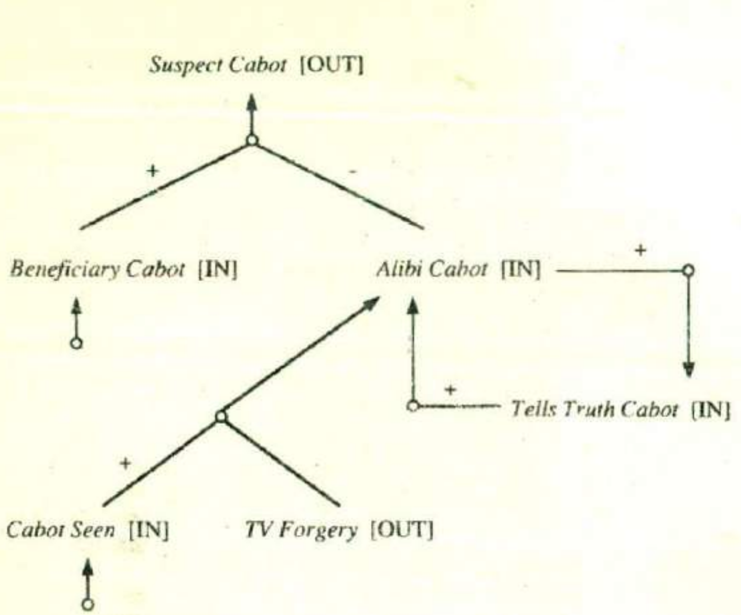


Figure 7.11: A Second Justification

are assumptions. So if we believed that the hotel register was a forgery, that Babbitt's brother-in-law lied, or that the television pictures were faked, we would have a suspect again and the contradiction would go back OUT. So there are four things we might believe to resolve the contradiction. That is as far as DDB will take us. It reports there is an OR tree with four nodes. What should we do?

A TMS has no answer for this question. Early TMSs picked an answer at random. More recent architectures take the more reasonable position that this choice was a problem for the same problem-solving agent that created the dependencies in the first place. But suppose we do pick one. Suppose, in particular, that we choose to believe that Babbitt's brother-in-law lied. What should be the justification for that belief? If we believe it just because not believing it leads to a contradiction, then we should install a justification that should be valid only as long as it needs to be. If later we find another way that the contradiction can be labeled OUT, we will not want to continue in our abductive belief.

For instance, suppose that we believe that the brother-in-law lied, but later we discover that a long-lost relative, jilted by the family, was in town the day of the murder. We would no longer have to believe the brother-in-law lied just to avoid a contradiction. A TMS may also have algorithms to create such justifications, which we call abductive since they are created using abductive reasoning. If they have the property that they are not unnecessarily valid, they are said to be *complete*. Figure 7.12 shows a complete abductive justification for the belief that Babbitt's brother-in-law lied. If we come to

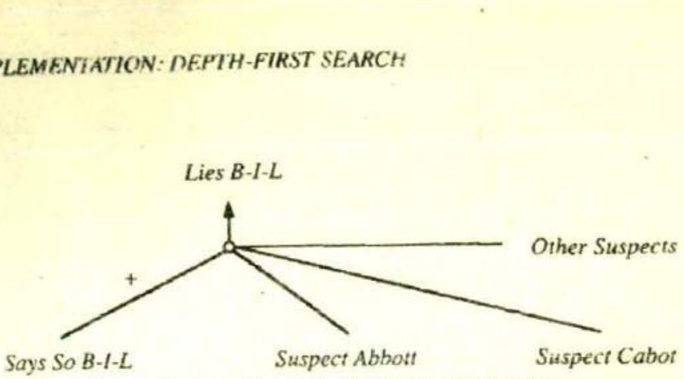


Figure 7.12: A Complete Abductive Justification

believe that Abbott or Cabot is a suspect, or we find a long-lost relative, or we somehow come to believe that Babbitt's brother-in-law didn't really say Babbitt was at his house, then this justification for lying will become invalid.

At this point, we have described the key reasoning operations that are performed by a JTMS:

- consistent labeling
- contradiction resolution

We have also described a set of important reasoning operations that a JTMS does not perform, including:

- applying rules to derive conclusions
- creating justifications for the results of applying rules (although justifications are created as part of contradiction resolution)
- choosing among alternative ways of resolving a contradiction
- detecting contradictions

All of these operations must be performed by the problem-solving program that is using the JTMS. In the next section, we describe a slightly different kind of TMS, in which, although the first three of these operations must still be performed by the problem-solving system, the last can be performed by the TMS.

7.5.3 Logic-Based Truth Maintenance Systems

A *logic-based truth maintenance system* (LTMS) [McAllester, 1980] is very similar to a JTMS. It differs in one important way. In a JTMS, the nodes in the network are treated as atoms by the TMS, which assumes no relationships among them except the ones that are explicitly stated in the justifications. In particular, a JTMS has no problem simultaneously labeling both P and $\neg P$ IN. For example, we could have represented explicitly both *Lies B-I-L* and *Not Lies B-I-L* and labeled both of them IN. No contradiction will be detected automatically. In an LTMS, on the other hand, a contradiction would be asserted automatically in such a case. If we had constructed the ABC example in an

LTMS system, we would not have created an explicit contradiction corresponding to the assertion that there was no suspect. Instead we would replace the contradiction node by one that asserted something like *No Suspect*. Then we would assert *Suspect*. When *No Suspect* came IN, it would cause a contradiction to be asserted automatically.

7.6 Implementation: Breadth-First Search

The *assumption-based truth maintenance system* (ATMS) [de Kleer, 1986] is an alternative way of implementing nonmonotonic reasoning. In both JTMS and LTMS systems, a single line of reasoning is pursued at a time, and dependency-directed backtracking occurs whenever it is necessary to change the system's assumptions. In an ATMS, alternative paths are maintained in parallel. Backtracking is avoided at the expense of maintaining multiple contexts, each of which corresponds to a set of consistent assumptions. As reasoning proceeds in an ATMS-based system, the universe of consistent contexts is pruned as contradictions are discovered. The remaining consistent contexts are used to label assertions, thus indicating the contexts in which each assertion has a valid justification. Assertions that do not have a valid justification in any consistent context can be pruned from consideration by the problem solver. As the set of consistent contexts gets smaller, so too does the set of assertions that can consistently be believed by the problem solver. Essentially, an ATMS system works breadth-first, considering all possible contexts at once, while both JTMS and LTMS systems operate depth-first.

The ATMS, like the JTMS, is designed to be used in conjunction with a separate problem solver. The problem solver's job is to:

- Create nodes that correspond to assertions (both those that are given as axioms and those that are derived by the problem solver).
- Associate with each such node one or more justifications, each of which describes a reasoning chain that led to the node.
- Inform the ATMS of inconsistent contexts.

Notice that this is identical to the role of the problem solver that uses a JTMS, except that no explicit choices among paths to follow need be made as reasoning proceeds. Some decision may be necessary at the end, though, if more than one possible solution still has a consistent context.

The role of the ATMS system is then to:

- Propagate inconsistencies, thus ruling out contexts that include subcontexts (sets of assertions) that are known to be inconsistent.
- Label each problem solver node with the contexts in which it has a valid justification. This is done by combining contexts that correspond to the components of a justification. In particular, given a justification of the form

  $$A1 \wedge A2 \wedge \cdots \wedge An \rightarrow C$$

  assign as a context for the node corresponding to C the intersection of the contexts corresponding to the nodes $A1$ through An.

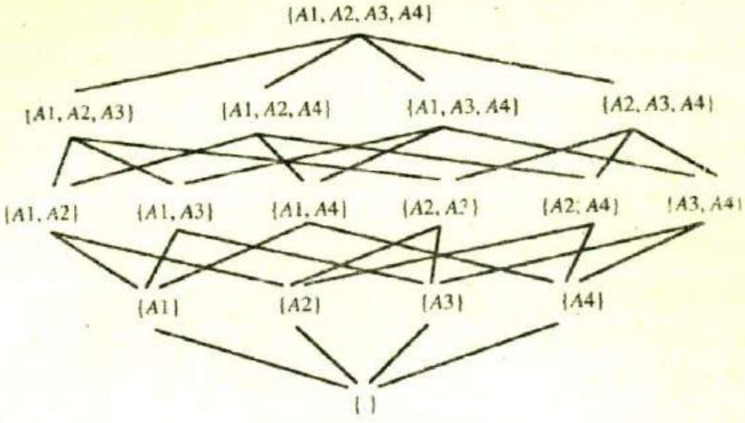


Figure 7.13: A Context Lattice

Contexts get eliminated as a result of the problem solver asserting inconsistencies and the ATMS propagating them. Nodes get created by the problem solver to represent possible components of a problem solution. They may then get pruned from consideration if all their context labels get pruned. Thus a choice among possible solution components gradually evolves in a process very much like the constraint satisfaction procedure that we examined in Section 3.5.

One problem with this approach is that given a set of n assumptions, the number of possible contexts that may have to be considered is 2^n. Fortunately, in many problem-solving scenarios, most of them can be pruned without ever looking at them. Further, the ATMS exploits an efficient labeling system that makes it possible to encode a set of contexts as a single context that delimits the set. To see how both of these things work, it is necessary to think of the set of contexts that are defined by a set of assumptions as forming a lattice, as shown for a simple example with four assumptions in Figure 7.13. Lines going upward indicate a subset relationship.

The first thing this lattice does for us is to illustrate a simple mechanism by which contradictions (inconsistent contexts) can be propagated so that large parts of the space of 2^n contexts can be eliminated. Suppose that the context labeled $\{A2, A3\}$ is asserted to be inconsistent. Then all contexts that include it (i.e., those that are above it) must also be inconsistent.

Now consider how a node can be labeled with all the contexts in which it has a valid justification. Suppose its justification depends on assumption $A1$. Then the context labeled $\{A1\}$ and all the contexts that include it are acceptable. But this can be indicated just by saying $\{A1\}$. It is not necessary to enumerate its supersets. In general, each node will be labeled with the greatest lower bounds of the contexts in which it should be believed.

Clearly, it is important that this lattice not be built explicitly but only used as an implicit structure as the ATMS proceeds.

As an example of how an ATMS-based problem solver works, let's return to the ABC Murder story. Again, our goal is to find a primary suspect. We need (at least) the following assumptions:

- $A1$. Hotel register was forged.
- $A2$. Hotel register was not forged.
- $A3$. Babbitt's brother-in-law lied.
- $A4$. Babbitt's brother-in-law did not lie.
- $A5$. Cabot lied.
- $A6$. Cabot did not lie.
- $A7$. Abbott, Babbitt, and Cabot are the only possible suspects.
- $A8$. Abbott, Babbitt, and Cabot are not the only suspects.

The problem solver could then generate the nodes and associated justifications shown in the first two columns of Figure 7.14. In the figure, the justification for a node that corresponds to a decision to make assumption N is shown as $\{N\}$. Justifications for nodes that correspond to the result of applying reasoning rules are shown as the rule involved. Then the ATMS can assign labels to the nodes as shown in the second two columns. The first shows the label that would be generated for each justification taken by itself. The second shows the label (possibly containing multiple contexts) that is actually assigned to the node given all its current justifications. These columns are identical in simple cases, but they may differ in more complex situations as we see for nodes 12, 13, and 14 of our example.

There are several things to notice about this example:

- Nodes may have several justifications if there are several possible reasons for believing them. This is the case for nodes 12, 13, and 14..
- Recall that when we were using a JTMS, a node was labeled IN if it had at least one valid justification. Using an ATMS, a node will end up being labeled with a consistent context if it has at least one justification that can occur in a consistent context.
- The label assignment process is sometimes complicated. We describe it in more detail below.

Suppose that a problem-solving program first created nodes 1 through 14, representing the various dependencies among them without committing to which of them it currently believes. It can indicate known contradictions by marking as no good the context:

- A, B, C are the only suspects; A, B, C are not the only suspects: $\{A7, A8\}$

	Nodes	Justifications		Node Labels
[1]	Register was not forged	$\{A2\}$	$\{A2\}$	$\{A2\}$
[2]	Abbott at hotel	$[1] \rightarrow [2]$	$\{A2\}$	$\{A2\}$
[3]	B-I-L didn't lie	$\{4\}$	$\{A4\}$	$\{A4\}$
[4]	Babbitt at B-I-L	$[3] \rightarrow [4]$	$\{A4\}$	$\{A4\}$
[5]	Cabot didn't lie	$\{6\}$	$\{A6\}$	$\{A6\}$
[6]	Cabot at ski show	$[5] \rightarrow [6]$	$\{A6\}$	$\{A6\}$
[7]	A, B, C only suspects	$\{A7\}$	$\{A7\}$	$\{A7\}$
[8]	Prime Suspect Abbott	$[7] \wedge [13] \wedge [14] \rightarrow [8]$	$\{A7, A4, A6\}$	$\{A7, A4, A6\}$
[9]	Prime Suspect Babbitt	$[7] \wedge [12] \wedge [14] \rightarrow [9]$	$\{A7, A2, A6\}$	$\{A7, A2, A6\}$
[10]	Prime Suspect Cabot	$[7] \wedge [12] \wedge [13] \rightarrow [10]$	$\{A7, A2, A4\}$	$\{A7, A2, A4\}$
[11]	A, B, C not only suspects	$\{A8\}$	$\{A8\}$	$\{A8\}$
[12]	Not prime suspect Abbott	$[2] \rightarrow [12]$	$\{A2\}$	$\{A2\}, \{A8\}$
		$[11] \rightarrow [12]$	$\{A8\}$	
		$[9] \rightarrow [12]$	$\{A7, A2, A6\}$	
		$[10] \rightarrow [12]$	$\{A7, A2, A4\}$	
[13]	Not prime suspect Babbitt	$[4] \rightarrow [13]$	$\{A4\}$	$\{A4\}, \{A8\}$
		$[11] \rightarrow [13]$	$\{A8\}$	
		$[8] \rightarrow [13]$	$\{A7, A4, A6\}$	
		$[10] \rightarrow [13]$	$\{A7, A4, A2\}$	
[14]	Not prime suspect Cabot	$[6] \rightarrow [14]$	$\{A6\}$	$\{A6\}, \{A8\}$
		$[11] \rightarrow [14]$	$\{A8\}$	
		$[8] \rightarrow [14]$	$\{A7, A4, A6\}$	
		$[9] \rightarrow [14]$	$\{A7, A2, A6\}$	

Figure 7.14: Nodes and Their Justifications and Labels

The ATMS would then assign the labels shown in the figure. Let's consider the case of node 12. We generate four possible labels, one for each justification. But we want to assign to the node a label that contains just the greatest lower bounds of all the contexts in which it can occur, since they implicitly encode the superset contexts. The label $\{A2\}$ is the greatest lower bound of the first, third, and fourth label, and $\{A8\}$ is the same for the second label. Thus those two contexts are all that are required as the label for the node. Now let's consider labeling node 8. Its label must be the union of the labels of nodes 7, 13, and 14. But nodes 13 and 14 have complex labels representing alternative justifications. So we must consider all ways of combining the labels of all three nodes. Fortunately, some of these combinations, namely those that contain both $A7$ and $A8$, can be eliminated because they are already known to be contradictory. Thus we are left with a single label as shown.

Now suppose the problem-solving program labels the context $\{A2\}$ as no good, meaning that the assumption it contains (namely that the hotel register was not forged) conflicts with what it knows. Then many of the labels that we had disappear since they are now inconsistent. In particular, the labels for nodes 1, 2, 9, 10, and 12 disappear. At this point, the only suspect node that has a label is node 8. But node 12 (Not prime suspect Abbott) also still has a label that corresponds to the assumption that Abbott, Babbitt, and Cabot are not the only suspects. If this assumption is made, then Abbott would not be a clear suspect even if the hotel register were forged. Further information or some choice process is still necessary to choose between these remaining nodes.

7.7 Summary

In this chapter we have discussed several logical systems that provide a basis for nonmonotonic reasoning, including nonmonotonic logic, default logic, abduction, inheritance, the closed world assumption, and circumscription. We have also described a way in which the kind of rules that we discussed in Chapter 6 could be augmented to support nonmonotonic reasoning.

We then presented three kinds of TMS systems, all of which provide a basis for implementing nonmonotonic reasoning. We have considered two dimensions along which TMS systems can vary: whether they automatically detect logical contradictions and whether they maintain single or multiple contexts. The following table summarizes this discussion:

TMS Kinds	single context	multiple context
nonlogical	JTMS	ATMS
logical	LTMS	?

As can be seen in this table, there is currently no TMS with logical contradictions and multiple contexts.

These various TMS systems each have advantages and disadvantages with respect to each other. The major issues that distinguish JTMS and ATMS systems are:

- The JTMS is often better when only a single solution is desired since it does not need to consider alternatives; the ATMS is usually more efficient if all solutions are eventually going to be needed.
- To create the context lattice, the ATMS performs a global operation in which it considers all possible combinations of assumptions. As a result, either all assumptions must be known at the outset of problem solving or an expensive, recompilation process must occur whenever an assumption is added. In the JTMS, on the other hand, the gradual addition of new assumptions poses no problem.
- The JTMS may spend a lot of time switching contexts when backtracking is necessary. Context switching does not happen in the ATMS.
- In an ATMS, inconsistent contexts disappear from consideration. If the initial problem description was overconstrained, then all nodes will end up with empty labels and there will be no problem-solving trace that can serve as a basis for relaxing one or more of the constraints. In a JTMS, on the other hand, the justification that is attached to a contradiction node provides exactly such a trace.
- The ATMS provides a natural way to answer questions of the form, "In what contexts is A true?" The only way to answer such questions using a JTMS is to try all the alternatives and record the ones in which A is labeled IN.

One way to get the best of both of these worlds is to combine an ATMS and a JTMS (or LTMS), letting each handle the part of the problem-solving process to which it is best suited.

The various nonmonotonic systems that we have described in this chapter have served as a basis for a variety of applications. One area of particular significance is diagnosis (for example, of faults in a physical device) [Reiter, 1987b; de Kleer and Williams, 1987]. Diagnosis is a natural application area for minimalist reasoning in particular, since one way to describe the diagnostic task is, "Find the smallest set of abnormally behaving components that would account for the observed behavior." A second application area is reasoning about action, with a particular emphasis on addressing the frame problem [Hanks and McDermott, 1986]. The frame problem is also natural for this kind of reasoning since it can be described as, "Assume that everything stays the same after an action except the things that necessarily change." A third application area is design [Steele *et al.*, 1989]. Here, nonmonotonic reasoning provides a basis for using common design principles to find a promising path quickly even in a huge design space while preserving the option to consider alternatives later if necessary. And yet another application area is in extracting intent from English expressions (see Chapter 15.)

In all the systems that we have discussed, we have assumed that belief status is a binary function. An assertion must eventually be either believed or not. Sometimes, this is too strong an assumption. In the next chapter, we present techniques for dealing with uncertainty without making that assumption. Instead, we allow for varying degrees of belief.

7.8 Exercises

1. Try to formulate the ABC Murder story in predicate logic and see how far you can get.

2. The classic example of nonmonotonic reasoning involves birds and flying. In particular, consider the following facts:

   - Most things do not fly.
   - Most birds do fly, unless they are too young or dead or have a broken wing.
   - Penguins and ostriches do not fly.
   - Magical ostriches fly.
   - Tweety is a bird.
   - Chirpy is either a penguin or an ostrich.
   - Feathers is a magical ostrich.

   Use one or more of the nonmonotonic reasoning systems we have discussed to answer the following questions:

   - Does Tweety fly?
   - Does Chirpy fly?
   - Does Feathers fly?
   - Does Paul fly?

3. Consider the missionaries and cannibals problem of Section 2.6. When you solved that problem, you used the CWA several times (probably without thinking about it). List some of the ways in which you used it.

4. A big technical problem that arises in defining circumscription precisely is the definition of a minimal model. Consider again the problem of Dick, the Quaker and Republican, which we can rewrite using a slightly different kind of AB predicate as:

$$\forall x : Republican(x) \land \neg AB1(x) \rightarrow \neg Pacifist(x)$$
$$\forall x : Quaker(x) \land \neg AB2(x) \rightarrow Pacifist(x)$$
$$Republican(x)$$
$$Quaker(x)$$

   (a) Write down the smallest models you can that describe the two extensions that we computed for that knowledge base.
   (b) Does it make sense to say that either is smaller than the other?
   (c) Prioritized circumscription [McCarthy, 1986] attempts to solve this problem by ranking predicates by the order in which they should be minimized. How could you use this idea to indicate a preference as to which extension to prefer?

5. Consider the problem of finding clothes to wear in the morning. To solve this problem, it is necessary to use knowledge such as:

   - Wear jeans unless either they are dirty or you have a job interview today.
   - Wear a sweater if it's cold.
   - It's usually cold in the winter.
   - Wear sandals if it's warm.
   - It's usually warm in the summer.

   (a) Build a JTMS-style database of the necessary facts to solve this problem.
   (b) Show how the problem can be solved and how the solution changes as the relevant facts (such as time of year and dirtiness of jeans) change.

6. Show how a JTMS could be used in medical diagnosis. Consider rules such as, "If you have a runny nose, assume you have a cold unless it is allergy season."

7. Solve the same medical reasoning problem with an ATMS.

8. Show how a JTMS could be used to select a TV program to watch. Consider rules such as, "If it is 6:00, then watch the news on channel 2 unless there is a football game still going on."

9. TMSs are useful tools in solving constraint satisfaction problems since they facilitate the nonmonotonic reasoning that occurs during the search for a complete solution.

(a) Show how a JTMS could be used to solve the cryptarithmetic problems of Chapter 2.

(b) Show how an ATMS would solve the same problem.

10. We described informally the JTMS labeling process. Write a formal description of that algorithm.

11. Work through the details of the ATMS node labeling process whose results are shown in Figure 7.14.

Chapter 8

Statistical Reasoning

So far, we have described several representation techniques that can be used to model belief systems in which, at any given point, a particular fact is believed to be true, believed to be false, or not considered one way or the other. For some kinds of problem solving, though, it is useful to be able to describe beliefs that are not certain but for which there is some supporting evidence. Let's consider two classes of such problems.

The first class contains problems in which there is genuine randomness in the world. Playing card games such as bridge and blackjack is a good example of this class Although in these problems it is not possible to predict the world with certainty, some knowledge about the likelihood of various outcomes is available, and we would like to be able to exploit it.

The second class contains problems that could, in principle, be modeled using the techniques we described in the last chapter. In these problems, the relevant world is not random; it behaves "normally" unless there is some kind of exception. The difficulty is that there are many more possible exceptions than we care to enumerate explicitly (using techniques such as *AB* and UNLESS). Many common sense tasks fall into this category, as do many expert reasoning tasks such as medical diagnosis. For problems like this statistical measures may serve a very useful function as summaries of the world; rather than enumerating all the possible exceptions, we can use a numerical summary that tells us how often an exception of some sort can be expected to occur.

In this chapter we explore several techniques that can be used to augment knowledge representation techniques with statistical measures that describe levels of evidence and belief.

8.1 Probability and Bayes' Theorem

An important goal for many problem-solving systems is to collect evidence as the system goes along and to modify its behavior on the basis of the evidence. To model this behavior, we need a statistical theory of evidence. Bayesian statistics is such a theory The fundamental notion of Bayesian statistics is that of conditional probability:

$$P(H|E)$$

Read this expression as the probability of hypothesis H given that we have observed evidence E. To compute this, we need to take into account the prior probability of H (the probability that we would assign to H if we had no evidence) and the extent to which E provides evidence of H. To do this, we need to define a universe that contains an exhaustive, mutually exclusive set of H_i's, among which we are trying to discriminate. Then, let

$P(H_i\|E)$ =	the probability that hypothesis H_i is true given evidence E
$P(E\|H_i)$ =	the probability that we will observe evidence E given that hypothesis i is true
$P(H_i)$ =	the *a priori* probability that hypothesis i is true in the absence of any specific evidence. These probabilities are called prior probabilities or *priors*.
k =	the number of possible hypotheses

Bayes' theorem then states that

$$P(H_i|E) = \frac{P(E|H_i) \cdot P(H_i)}{\sum_{n=1}^{k} P(E|H_n) \cdot P(H_n)}$$

Suppose, for example, that we are interested in examining the geological evidence at a particular location to determine whether that would be a good place to dig to find a desired mineral. If we know the prior probabilities of finding each of the various minerals and we know the probabilities that if a mineral is present then certain physical characteristics will be observed, then we can use Bayes' formula to compute, from the evidence we collect, how likely it is that the various minerals are present. This is, in fact, what is done by the PROSPECTOR program [Duda *et al.*, 1979], which has been used successfully to help locate deposits of several minerals, including copper and uranium.

The key to using Bayes' theorem as a basis for uncertain reasoning is to recognize exactly what it says. Specifically, when we say $P(A|B)$, we are describing the conditional probability of A given that the only evidence we have is B. If there is also other relevant evidence, then it too must be considered. Suppose, for example, that we are solving a medical diagnosis problem. Consider the following assertions:

S: patient has spots
M: patient has measles
F: patient has high fever

Without any additional evidence, the presence of spots serves as evidence in favor of measles. It also serves as evidence of fever since measles would cause fever. But suppose we already know that the patient has measles. Then the additional evidence that he has spots actually tells us nothing about the likelihood of fever. Alternatively, either spots alone or fever alone would constitute evidence in favor of measles. If both are present, we need to take both into account in determining the total weight of evidence. But, since spots and fever are not independent events, we cannot just sum their effects. Instead, we need to represent explicitly the conditional probability that arises from their conjunction. In general, given a prior body of evidence e and some new observation E, we need to compute

$$P(H|E,e) = P(H|E) \cdot \frac{P(e|E,H)}{P(e|E)}$$

Unfortunately, in an arbitrarily complex world, the size of the set of joint probabilities that we require in order to compute this function grows as 2^n if there are n different propositions being considered. This makes using Bayes' theorem intractable for several reasons:

- The knowledge acquisition problem is insurmountable; too many probabilities have to be provided. In addition, there is substantial empirical evidence (e.g., Tversky and Kahneman [1974] and Kahneman *et al.* [1982]) that people are very poor probability estimators.
- The space that would be required to store all the probabilities is too large.
- The time required to compute the probabilities is too large.

Despite these problems, though, Bayesian statistics provide an attractive basis for an uncertain reasoning system. As a result, several mechanisms for exploiting its power while at the same time making it tractable have been developed. In the rest of this chapter, we explore three of these:

- Attaching certainty factors to rules
- Bayesian networks
- Dempster-Shafer theory

We also mention one very different numerical approach to uncertainty, fuzzy logic.

There has been an active, strident debate for many years on the question of whether pure Bayesian statistics are adequate as a basis for the development of reasoning programs. (See, for example, Cheeseman [1985] for arguments that it is and Buchanan and Shortliffe [1984] for arguments that it is not.) On the one hand, non-Bayesian approaches have been shown to work well for some kinds of applications (as we see below). On the other hand, there are clear limitations to all known techniques. In essence, the jury is still out. So we sidestep the issue as much as possible and simply describe a set of methods and their characteristics.

8.2 Certainty Factors and Rule-Based Systems

In this section we describe one practical way of compromising on a pure Bayesian system. The approach we discuss was pioneered in the MYCIN system [Shortliffe, 1976; Buchanan and Shortliffe, 1984; Shortliffe and Buchanan, 1975], which attempts to recommend appropriate therapies for patients with bacterial infections. It interacts with the physician to acquire the clinical data it needs. MYCIN is an example of an *expert system*, since it performs a task normally done by a human expert. Here we concentrate on the use of probabilistic reasoning; Chapter 20 provides a broader view of expert systems.

MYCIN represents most of its diagnostic knowledge as a set of rules. Each rule has associated with it a *certainty factor*, which is a measure of the extent to which the evidence that is described by the antecedent of the rule supports the conclusion that is given in the rule's consequent. A typical MYCIN rule looks like:

```
If: (1) the stain of the organism is gram-positive, and
    (2) the morphology of the organism is coccus, and
    (3) the growth conformation of the organism is clumps,
then there is suggestive evidence (0.7) that
  the identity of the organism is staphylococcus.
```

This is the form in which the rules are stated to the user. They are actually represented internally in an easy-to-manipulate LISP list structure. The rule we just saw would be represented internally as

```
PREMISE: ($AND (SAME CNTXT GRAM GRAMPOS)
               (SAME CNTXT MORPH COCCUS)
               (SAME CNTXT CONFORM CLUMPS))
ACTION:  (CONCLUDE CNTXT IDENT STAPHYLOCOCCUS TALLY 0.7)
```

MYCIN uses these rules to reason backward to the clinical data available from its goal of finding significant disease-causing organisms. Once it finds the identities of such organisms, it then attempts to select a therapy by which the disease(s) may be treated. In order to understand how MYCIN exploits uncertain information, we need answers to two questions: "What do certainty factors mean?" and "How does MYCIN combine the estimates of certainty in each of its rules to produce a final estimate of the certainty of its conclusions?" A further question that we need to answer, given our observations about the intractability of pure Bayesian reasoning, is, "What compromises does the MYCIN technique make and what risks are associated with those compromises?" In the rest of this section we answer all these questions.

Let's start first with a simple answer to the first question (to which we return with a more detailed answer later). A certainty factor ($CF[h,e]$) is defined in terms of two components:

- $MB[h,e]$—a measure (between 0 and 1) of belief in hypothesis h given the evidence e. MB measures the extent to which the evidence supports the hypothesis. It is zero if the evidence fails to support the hypothesis.

- $MD[h,e]$—a measure (between 0 and 1) of disbelief in hypothesis h given the evidence e. MD measures the extent to which the evidence supports the negation of the hypothesis. It is zero if the evidence supports the hypothesis.

From these two measures, we can define the certainty factor as

$$CF[h,e] = MB[h,e] - MD[h,e]$$

Since any particular piece of evidence either supports or denies a hypothesis (but not both), and since each MYCIN rule corresponds to one piece of evidence (although it may be a compound piece of evidence), a single number suffices for each rule to define both the MB and MD and thus the CF.

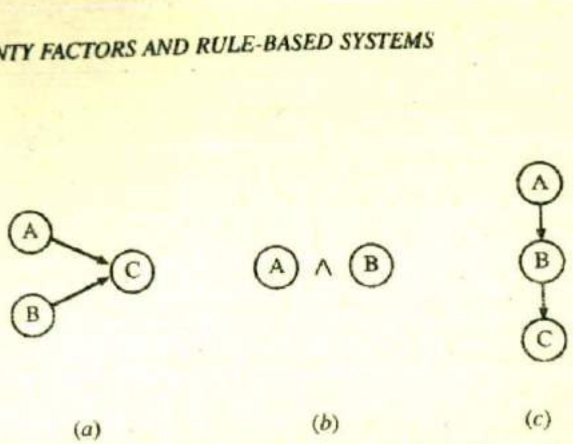


Figure 8.1: Combining Uncertain Rules

The *CF*'s of MYCIN's rules are provided by the experts who write the rules. They reflect the experts' assessments of the strength of the evidence in support of the hypothesis. As MYCIN reasons, however, these *CF*'s need to be combined to reflect the operation of multiple pieces of evidence and multiple rules applied to a problem. Figure 8.1 illustrates three combination scenarios that we need to consider. In Figure 8.1(*a*), several rules all provide evidence that relates to a single hypothesis. In Figure 8.1(*b*), we need to consider our belief in a collection of several propositions taken together. In Figure 8.1(*c*), the output of one rule provides the input to another.

What formulas should be used to perform these combinations? Before we answer that question, we need first to describe some properties that we would like the combining functions to satisfy:

- Since the order in which evidence is collected is arbitrary, the combining functions should be commutative and associative.
- Until certainty is reached, additional confirming evidence should increase *MB* (and similarly for disconfirming evidence and *MD*).
- If uncertain inferences are chained together, then the result should be less certain than either of the inferences alone.

Having accepted the desirability of these properties, let's first consider the scenario in Figure 8.1(*a*), in which several pieces of evidence are combined to determine the *CF* of one hypothesis. The measures of belief and disbelief of a hypothesis given two observations s_1 and s_2 are computed from:

$$MB[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MD[h, s_1 \wedge s_2] = 1 \\ MB[h, s_1] + MB[h, s_2] \cdot (1 - MB[h, s_1]) & \text{otherwise} \end{cases}$$

$$MD[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MB[h, s_1 \wedge s_2] = 1 \\ MD[h, s_1] + MD[h, s_2] \cdot (1 - MD[h, s_1]) & \text{otherwise} \end{cases}$$

One way to state these formulas in English is that the measure of belief in h is 0 if h is disbelieved with certainty. Otherwise, the measure of belief in h given two observations is the measure of belief given only one observation plus some increment for the second observation. This increment is computed by first taking the difference between 1 (certainty) and the belief given only the first observation. This difference is the most that can be added by the second observation. The difference is then scaled by the belief in h given only the second observation. A corresponding explanation can be given, then, for the formula for computing disbelief. From MB and MD, CF can be computed. Notice that if several sources of corroborating evidence are pooled, the absolute value of CF will increase. If conflicting evidence is introduced, the absolute value of CF will decrease.

A simple example shows how these functions operate. Suppose we make an initial observation that confirms our belief in h with $MB = 0.3$. Then $MD[h, s_1] = 0$ and $CF[h, s_1] = 0.3$. Now we make a second observation, which also confirms h, with $MB[h, s_2] = 0.2$. Now:

$$\begin{aligned} MB[h, s_1 \wedge s_2] &= 0.3 + 0.2 \cdot 0.7 \\ &= 0.44 \\ MD[h, s_1 \wedge s_2] &= 0.0 \\ CF[h, s_1 \wedge s_2] &= 0.44 \end{aligned}$$

You can see from this example how slight confirmatory evidence can accumulate to produce increasingly larger certainty factors.

Next let's consider the scenario of Figure 8.1(b), in which we need to compute the certainty factor of a combination of hypotheses. In particular, this is necessary when we need to know the certainty factor of a rule antecedent that contains several clauses (as, for example, in the staphylococcus rule given above). The combination certainty factor can be computed from its MB and MD. The formulas MYCIN uses for the MB of the conjunction and the disjunction of two hypotheses are:

$$MB[h_1 \wedge h_2, e] = \min(MB[h_1, e], MB[h_2, e])$$

$$MB[h_1 \vee h_2, e] = \max(MB[h_1, e], MB[h_2, e])$$

MD can be computed analogously.

Finally, we need to consider the scenario in Figure 8.1(c), in which rules are chained together with the result that the uncertain outcome of one rule must provide the input to another. Our solution to this problem will also handle the case in which we must assign a measure of uncertainty to initial inputs. This could easily happen in situations where the evidence is the outcome of an experiment or a laboratory test whose results are not completely accurate. In such a case, the certainty factor of the hypothesis must take into account both the strength with which the evidence suggests the hypothesis and the level of confidence in the evidence. MYCIN provides a chaining rule that is defined as follows. Let $MB'[h, s]$ be the measure of belief in h given that we are absolutely sure of the validity of s. Let e be the evidence that led us to believe in s (for example, the actual readings of the laboratory instruments or the results of applying other rules). Then:

$$MB[h,s] = MB'[h,s] \cdot \max(0, CF[s,e])$$

Since initial *CF*'s in MYCIN are estimates that are given by experts who write the rules, it is not really necessary to state a more precise definition of what a *CF* means than the one we have already given. The original work did, however, provide one by defining *MB* (which can be thought of as a proportionate decrease in disbelief in *h* as a result of *e*) as:

$$MB[h,e] = \begin{cases} 1 & \text{if } P(h) = 1 \\ \frac{\max[P(h|e),P(h)] - P(h)}{1 - P(h)} & \text{otherwise} \end{cases}$$

Similarly, the *MD* is the proportionate decrease in belief in *h* as a result of *e*:

$$MD[h,e] = \begin{cases} 1 & \text{if } P(h) = 0 \\ \frac{\min[P(h|e),P(h)] - P(h)}{-P(h)} & \text{otherwise} \end{cases}$$

It turns out that these definitions are incompatible with a Bayesian view of conditional probability. Small changes to them, however, make them compatible [Heckerman, 1986]. In particular, we can redefine *MB* as

$$MB[h,e] = \begin{cases} 1 & \text{if } P(h) = 1 \\ \frac{\max[P(h|e),P(h)] - P(h)}{(1 - P(h)) \cdot P(h|e)} & \text{otherwise} \end{cases}$$

The definition of *MD* must also be changed similarly.

With these reinterpretations, there ceases to be any fundamental conflict between MYCIN's techniques and those suggested by Bayesian statistics. We argued at the end of the last section that pure Bayesian statistics usually leads to intractable systems. But MYCIN works [Buchanan and Shortliffe, 1984]. Why?

Each *CF* in a MYCIN rule represents the contribution of an individual rule to MYCIN's belief in a hypothesis. In some sense then, it represents a conditional probability, $P(H|E)$. But recall that in a pure Bayesian system, $P(H|E)$ describes the conditional probability of H given that the only relevant evidence is E. If there is other evidence, joint probabilities need to be considered. This is where MYCIN diverges from a pure Bayesian system, with the result that it is easier to write and more efficient to execute, but with the corresponding risk that its behavior will be counterintuitive. In particular, the MYCIN formulas for all three combination scenarios of Figure 8.1 make the assumption that all rules are independent. The burden of guaranteeing independence (at least to the extent that it matters) is on the rule writer. Each of the combination scenarios is vulnerable when this independence assumption is violated.

Let's first consider the scenario in Figure 8.1(*a*). Our example rule has three antecedents with a single *CF* rather than three separate rules; this makes the combination rules unnecessary. The rule writer did this because the three antecedents are not independent. To see how much difference MYCIN's independence assumption can make, suppose for a moment that we had instead had three separate rules and that the *CF* of each was 0.6. This could happen and still be consistent with the combined *CF* of 0.7 if the three conditions overlap substantially. If we apply the MYCIN combination formula to the three separate rules, we get

$$\begin{aligned} MB[h, s_1 \wedge s_2] &= 0.6 + (0.6 \cdot 0.4) \\ &= 0.84 \\ MB[h, (s_1 \wedge s_2) \wedge s_3] &= 0.84 + (0.6 \cdot 0.16) \\ &= 0.936 \end{aligned}$$

This is a substantially different result than the true value, as expressed by the expert, of 0.7.

Now let's consider what happens when independence assumptions are violated in the scenario of Figure 8.1(*c*). Let's consider a concrete example in which:

S: sprinkler was on last night
W: grass is wet
R: it rained last night

We can write MYCIN-style rules that describe predictive relationships among these three events:

```
If: the sprinkler was on last night
then there is suggestive evidence (0.9) that
  the grass will be wet this morning
```

Taken alone, this rule may accurately describe the world. But now consider a second rule:

```
If: the grass is wet this morning
then there is suggestive evidence (0.8) that
  it rained last night
```

Taken alone, this rule makes sense when rain is the most common source of water on the grass. But if the two rules are applied together, using MYCIN's rule for chaining, we get

$$\begin{aligned} MB[W, S] &= 0.8 && \{\text{sprinkler suggests wet}\} \\ MB[R, W] &= 0.8 \cdot 0.9 = 0.72 && \{\text{wet suggests rains}\} \end{aligned}$$

In other words, we believe that it rained because we believe the sprinkler was on. We get this despite the fact that if the sprinkler is known to have been on and to be the cause of the grass being wet, then there is actually almost no evidence for rain (because the wet grass has been explained some other way). One of the major advantages of the modularity of the MYCIN rule system is that it allows us to consider individual antecedent/consequent relationships independently of others. In particular, it lets us talk about the implications of a proposition without going back and considering the evidence that supported it. Unfortunately, this example shows that there is a danger in this approach whenever the justifications of a belief are important to determining its consequences. In this case, we need to know why we believe the grass is wet (e.g., because we observed it to be wet as opposed to because we know the sprinkler was on) in order to determine whether the wet grass is evidence for it having just rained.

It is worth pointing out here that this example illustrates one specific rule structure that almost always causes trouble and should be avoided. Notice that our first rule describes a causal relationship (sprinkler causes wet grass). The second rule, although it looks the same, actually describes an inverse causality relationship (wet grass is caused by rain and thus is evidence for its cause). Although one can derive evidence for a symptom from its cause and for a cause from observing its symptom, it is important that evidence that is derived one way not be used again to go back the other way with no new information. To avoid this problem, many rule-based systems either limit their rules to one structure or clearly partition the two kinds so that they cannot interfere with each other. When we discuss Bayesian networks in the next section, we describe a systematic solution to this problem.

We can summarize this discussion of certainty factors and rule-based systems as follows. The approach makes strong independence assumptions that make it relatively easy to use; at the same time assumptions create dangers if rules are not written carefully so that important dependencies are captured. The approach can serve as the basis of practical application programs. It did so in MYCIN. It has done so in a broad array of other systems that have been built on the EMYCIN platform [van Melle *et al.*, 1981], which is a generalization (often called a *shell*) of MYCIN with all the domain-specific rules stripped out. One reason that this framework is useful, despite its limitations, is that it appears that in an otherwise robust system the exact numbers that are used do not matter very much. The other reason is that the rules were carefully designed to avoid the major pitfalls we have just described. One other interesting thing about this approach is that it appears to mimic quite well [Shultz *et al.*, 1989] the way people manipulate certainties.

8.3 Bayesian Networks

In the last section, we described CF's as a mechanism for reducing the complexity of a Bayesian reasoning system by making some approximations to the formalism. In this section, we describe an alternative approach, *Bayesian networks* [Pearl, 1988], in which we preserve the formalism and rely instead on the modularity of the world we are trying to model. The main idea is that to describe the real world, it is not necessary to use a huge joint probabilility table in which we list the probabilities of all conceivable combinations of events. Most events are conditionally independent of most other ones, so their interactions need not be considered. Instead, we can use a more local representation in which we will describe clusters of events that interact.

Recall that in Figure 8.1 we used a network notation to describe the various kinds of constraints on likelihoods that propositions can have on each other. The idea of constraint networks turns out to be very powerful. We expand on it in this section as a way to represent interactions among events; we also return to it later in Sections 11.3.1 and 14.3, where we talk about other ways of representing knowledge as sets of constraints.

Let's return to the example of the sprinkler, rain, and grass that we introduced in the last section. Figure 8.2(*a*) shows the flow of constraints we described in MYCIN-style

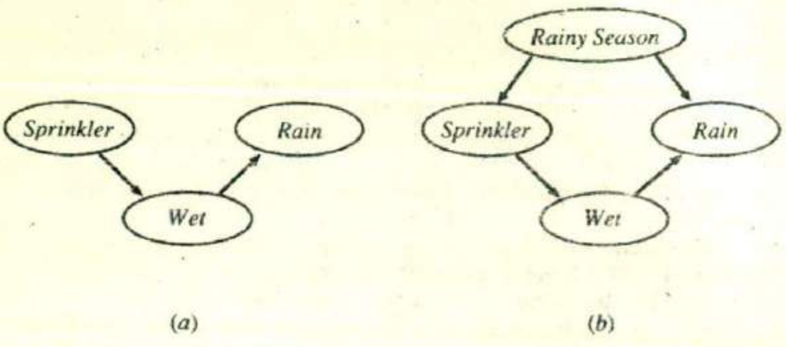


Figure 8.2: Representing Causality Uniformly

rules. But recall that the problem that we encountered with that example was that the constraints flowed incorrectly from "sprinkler on" to "rained last night." The problem was that we failed to make a distinction that turned out to be critical. There are two different ways that propositions can influence the likelihood of each other. The first is that causes influence the likelihood of their symptoms; the second is that observing a symptom affects the likelihood of all of its possible causes. The idea behind the Bayesian network structure is to make a clear distinction between these two kinds of influence.

Specifically, we construct a directed acyclic graph (DAG) that represents causality relationships among variables. The idea of a causality graph (or network) has proved to be very useful in several systems, particularly medical diagnosis systems such as CASNET [Weiss *et al.*, 1978] and INTERNIST/CADUCEUS [Pople, 1982]. The variables in such a graph may be propositional (in which case they can take on the values TRUE and FALSE) or they may be variables that take on values of some other type (e.g., a specific disease, a body temperature, or a reading taken by some other diagnostic device). In Figure 8.2(*b*), we show a causality graph for the wet grass example. In addition to the three nodes we have been talking about, the graph contains a new node corresponding to the propositional variable that tells us whether it is currently the rainy season.

A DAG, such as the one we have just drawn, illustrates the causality relationships that occur among the nodes it contains. In order to use it as a basis for probabilistic reasoning, however, we need more information. In particular, we need to know, for each value of a parent node, what evidence is provided about the values that the child node can take on. We can state this in a table in which the conditional probabilities are provided. We show such a table for our example in Figure 8.3. For example, from the table we see that the prior probability of the rainy season is 0.5. Then, if it is the rainy season, the probability of rain on a given night is 0.9; if it is not, the probability is only 0.1.

To be useful as a basis for problem solving, we need a mechanism for computing the influence of any arbitrary node on any other. For example, suppose that we have observed that it rained last night. What does that tell us about the probability that it is the

Attribute	*Probability*
p(Wet|Sprinkler, Rain)	0.95
p(Wet|Sprinkler, ¬Rain)	0.9
p(Wet|¬Sprinkler, Rain)	0.8
p(Wet|¬Sprinkler, ¬Rain)	0.1
p(Sprinkler|RainySeason)	0.0
p(Sprinkler|¬RainySeason)	1.0
p(Rain|RainySeason)	0.9
p(Rain|¬RainySeason)	0.1
p(RainySeason)	0.5

Figure 8.3 Conditional Probabilities for a Bayesian Network

rainy season? To answer this question requires that the initial DAG be converted to an undirected graph in which the arcs can be used to transmit probabilities in either direction, depending on where the evidence is coming from. We also require a mechanism for using the graph that guarantees that probabilities are transmitted correctly. For example, while it is true that observing wet grass may be evidence for rain, and observing rain is evidence for wet grass, we must guarantee that no cycle is ever traversed in such a way that wet grass is evidence for rain, which is then taken as evidence for wet grass, and so forth.

There are three broad classes of algorithms for doing these computations: a message-passing method [Pearl, 1988], a clique triangulation method [Lauritzen and Spiegelhalter, 1988], and a variety of stochastic algorithms. The idea behind these methods is to take advantage of the fact that nodes have limited domains of influence. Thus, although in principle the task of updating probabilities consistently throughout the network is intractable, in practice it may not be. In the clique triangulation method, for example, explicit arcs are introduced between pairs of nodes that share a common descendent. For the case shown in Figure 8.2(*b*), a link would be introduced between *Sprinkler* and *Rain*. This explicit link supports assessing the impact of the observation *Sprinkler* on the hypothesis *Rain*. This is important since wet grass could be evidence of either of them, but wet grass plus one of its causes is not evidence for the competing cause since an alternative explanation for the observed phenomenon already exists.

The message-passing approach is based on the observation that to compute the probability of a node A given what is known about other nodes in the network, it is necessary to know three things:

- π - the total support arriving at A from its parent nodes (which represent its causes).
- λ - the total support arriving at A from its children (which represent its symptoms).
- The entry in the fixed conditional probability matrix that relates A to its causes.

Several methods for propagating π and λ messages and updating the probabilities at the nodes have been developed. The structure of the network determines what approach can be used. For example, in singly connected networks (those in which there is only a single path between every pair of nodes), a simpler algorithm can be used than in the case of multiply connected ones. For details, see Pearl [1988].

Finally, there are stochastic, or randomized algorithms for updating belief networks. One such algorithm [Chavez, 1989] transforms an arbitrary network into a Markov chain. The idea is to shield a given node probabilistically from most of the other nodes in the network. Stochastic algorithms run fast in practice, but may not yield absolutely correct results.

8.4 Dempster-Shafer Theory

So far, we have described several techniques, all of which consider individual propositions and assign to each of them a point estimate (i.e., a single number) of the degree of belief that is warranted given the evidence. In this section, we consider an alternative technique, called *Dempster-Shafer theory* [Dempster, 1968; Shafer, 1976]. This new approach considers sets of propositions and assigns to each of them an interval

$$[Belief, Plausibility]$$

in which the degree of belief must lie. Belief (usually denoted *Bel*) measures the strength of the evidence in favor of a set of propositions. It ranges from 0 (indicating no evidence) to 1 (denoting certainty).

Plausibility (*Pl*) is defined to be

$$Pl(s) = 1 - Bel(\neg s)$$

It also ranges from 0 to 1 and measures the extent to which evidence in favor of $\neg s$ leaves room for belief in s. In particular, if we have certain evidence in favor of $\neg s$, then $Bel(\neg s)$ will be 1 and $Pl(s)$ will be 0. This tells us that the only possible value for $Bel(s)$ is also 0.

The belief-plausibility interval we have just defined measures not only our level of belief in some propositions, but also the amount of information we have. Suppose that we are currently considering three competing hypotheses: A, B, and C. If we have no information, we represent that by saying, for each of them, that the true likelihood is in the range [0, 1]. As evidence is accumulated, this interval can be expected to shrink, representing increased confidence that we know how likely each hypothesis is. Note that this contrasts with a pure Bayesian approach, in which we would probably begin by distributing the prior probability equally among the hypotheses and thus assert for each that $P(h) = 0.33$. The interval approach makes it clear that we have no information when we start. The Bayesian approach does not, since we could end up with the same probability values if we collected volumes of evidence, which taken together suggest that the three values occur equally often. This difference can matter if one of the decisions that our program needs to make is whether to collect more evidence or to act on the basis of the evidence it already has.

So far, we have talked intuitively about *Bel* as a measure of our belief in some hypothesis given some evidence. Let's now define it more precisely. To do this, we need to start, just as with Bayes' theorem, with an exhaustive universe of mutually exclusive hypotheses. We'll call this the *frame of discernment* and we'll write it as Θ. For example, in a simplified diagnosis problem, Θ might consist of the set $\{All, Flu, Cold, Pneu\}$:

All: allergy
Flu: flu
Cold: cold
Pneu: pneumonia

Our goal is to attach some measure of belief to elements of Θ. However, not all evidence is directly supportive of individual elements. Often it supports sets of elements (i.e., subsets of Θ). For example, in our diagnosis problem, fever might support $\{Flu, Cold, Pneu\}$. In addition, since the elements of Θ are mutually exclusive, evidence in favor of some may have an affect on our belief in the others. In a purely Bayesian system, we can handle both of these phenomena by listing all of the combinations of conditional probabilities. But our goal is not to have to do that. Dempster-Shafer theory lets us handle interactions by manipulating sets of hypotheses directly.

The key function we use is a probability density function, which we denote as m. The function m is defined not just for elements of Θ but for all subsets of it (including singleton subsets, which correspond to individual elements). The quantity $m(p)$ measures the amount of belief that is currently assigned to exactly the set p of hypotheses. If Θ contains n elements, then there are 2^n subsets of Θ. We must assign m so that the sum of all the m values assigned to the subsets of Θ is 1. Although dealing with 2^n values may appear intractable, it usually turns out that many of the subsets will never need to be considered because they have no significance in the problem domain (and so their associated value of m will be 0).

Let's see how m works for our diagnosis problem. Assume that we have no information about how to choose among the four hypotheses when we start the diagnosis task. Then we define m as:

$$\{\Theta\} \quad (1.0)$$

All other values of m are thus 0. Although this means that the actual value must be some one element *All*, *Flu*, *Cold*, or *Pneu*, we do not have any information that allows us to assign belief in any other way than to say that we are sure the answer is somewhere in the whole set. Now suppose we acquire a piece of evidence that suggests (at a level of 0.6) that the correct diagnosis is in the set $\{Flu, Cold, Pneu\}$. Fever might be such a piece of evidence. We update m as follows:

$$\begin{array}{ll} \{Flu, Cold, Pneu\} & (0.6) \\ \{\Theta\} & (0.4) \end{array}$$

At this point, we have assigned to the set $\{Flu, Cold, Pneu\}$ the appropriate belief. The remainder of our belief still resides in the larger set Θ. Notice that we do not make the commitment that the remainder must be assigned to the complement of $\{Flu, Cold, Pneu\}$.

Having defined m, we can now define $Bel(p)$ for a set p as the sum of the values of m for p and for all of its subsets. Thus $Bel(p)$ is our overall belief that the correct answer lies somewhere in the set p.

In order to be able to use m (and thus *Bel* and *Pl*) in reasoning programs, we need to define functions that enable us to combine m's that arise from multiple sources of evidence.

Recall that in our discussion of *CF*'s, we considered three combination scenarios, which we illustrated in Figure 8.1. When we use Dempster-Shafer theory, on the other hand, we do not need an explicit combining function for the scenario in Figure 8.1(b) since we have that capability already in our ability to assign a value of m to a set of hypotheses. But we do need a mechanism for performing the combinations of scenarios (a) and (c). Dempster's rule of combination serves both these functions. It allows us to combine any two belief functions (whether they represent multiple sources of evidence for a single hypothesis or multiple sources of evidence for different hypotheses).

Suppose we are given two belief functions m_1 and m_2. Let X be the set of subsets of Θ to which m_1 assigns a nonzero value and let Y be the corresponding set for m_2. We define the combination m_3 of m_1 and m_2 to be

$$m_3(Z) = \frac{\sum_{X \cap Y = Z} m_1(X) \cdot m_2(Y)}{1 - \sum_{X \cap Y = \emptyset} m_1(X) \cdot m_2(Y)}$$

This gives us a new belief function that we can apply to any subset Z of Θ. We can describe what this formula is doing by looking first at the simple case in which all ways of intersecting elements of X and elements of Y generate nonempty sets. For example, suppose m_1 corresponds to our belief after observing fever:

$\{Flu, Cold, Pneu\}$	(0.6)
Θ	(0.4)

Suppose m_2 corresponds to our belief after observing a runny nose:

$\{All, Flu, Cold\}$	(0.8)
Θ	(0.2)

Then we can compute their combination m_3 using the following table (in which we further abbreviate disease names), which we can derive using the numerator of the combination rule:

		$\{A,F,C\}$	(0.8)	Θ	(0.2)
$\{F,C,P\}$	(0.6)	$\{F,C\}$	(0.48)	$\{F,C,P\}$	(0.12)
Θ	(0.4)	$\{A,F,C\}$	(0.32)	Θ	(0.08)

The four sets that are generated by taking all ways of intersecting an element of X and an element of Y are shown in the body of the table. The value of m_3 that the combination rule associates with each of them is computed by multiplying the values of m_1 and m_2 associated with the elements from which they were derived. Although it did not happen in this simple case, it is possible for the same set to be derived in more than one way during this intersection process. If that does occur, then to compute m_3 for that set, it is necessary to compute the sum of all the individual values that are generated for all the distinct ways in which the set is produced (thus the summation sign in the numerator of the combination formula).

A slightly more complex situation arises when some of the subsets created by the intersection operation are empty. Notice that we are guaranteed by the way we compute m_3 that the sum of all its individual values is 1 (assuming that the sums of all the values of m_1 and m_2 are 1). If some empty subsets are created, though, then some of m_3 will be assigned to them. But from the fact that we assumed that Θ is exhaustive, we know that the true value of the hypothesis must be contained in some nonempty subset of Θ. So we need to redistribute any belief that ends up in the empty subset proportionately across the nonempty ones. We do that with the scaling factor shown in the denominator of the combination formula. If no nonempty subsets are created, the scaling factor is 1, so we were able to ignore it in our first example. But to see how it works, let's add a new piece of evidence to our example. As a result of applying m_1 and m_2, we produced m_3:

$\{Flu, Cold\}$	(0.48)
$\{All, Flu, Cold\}$	(0.32)
$\{Flu, Cold, Pneu\}$	(0.12)
Θ	(0.08)

Now, let m_4 correspond to our belief given just the evidence that the problem goes away when the patient goes on a trip:

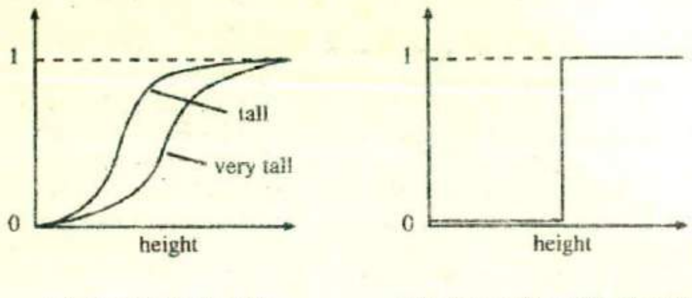
$\{All\}$	(0.9)
Θ	(0.1)

We can apply the numerator of the combination rule to produce (where $\emptyset$ denotes the empty set):

		$\{A\}$	(0.9)	Θ	(0.1)
$\{F, C\}$	(0.48)	$\emptyset$	(0.432)	$\{F, C\}$	(0.048)
$\{A, F, C\}$	(0.32)	$\{A, F, C\}$	(0.288)	$\{A, F, C\}$	(0.032)
$\{F, C, P\}$	(0.12)	$\emptyset$	(0.108)	$\{F, C, P\}$	(0.012)
Θ	(0.08)	$\{A\}$	(0.072)	Θ	(0.008)

But there is now a total belief of 0.54 associated with $\emptyset$; only 0.45 is associated with outcomes that are in fact possible. So we need to scale the remaining values by the factor $1 - 0.54 = 0.46$. If we do this, and also combine alternative ways of generating the set $\{All, Flu, Cold\}$, then we get the final combined belief function, m_5:

$\{Flu, Cold\}$	(0.104)
$\{All, Flu, Cold\}$	(0.696)
$\{Flu, Cold, Pneu\}$	(0.026)
$\{All\}$	(0.157)
Θ	(0.017)



(*a*) Fuzzy Membership (*b*) Conventional Membership

Figure 8.4: Fuzzy versus Conventional Set Membership

In this example, the percentage of m_5 that was initially assigned to the empty set was large (over half). This happens whenever there is conflicting evidence (as in this case between m_1 and m_4).

8.5 Fuzzy Logic

In the techniques we have discussed so far, we have not modified the mathematical underpinnings provided by set theory and logic. We have instead augmented those ideas with additional constructs provided by probability theory. In this section, we take a different approach and briefly consider what happens if we make fundamental changes to our idea of set membership and corresponding changes to our definitions of logical operations.

The motivation for fuzzy sets is provided by the need to represent such propositions as:

John is very tall.
Mary is slightly ill.
Sue and Linda are close friends.
Exceptions to the rule are nearly impossible.
Most Frenchmen are not very tall.

While traditional set theory defines set membership as a boolean predicate, fuzzy set theory allows us to represent set membership as a possibility distribution, such as the ones shown in Figure 8.4(*a*) for the set of tall people and the set of very tall people. Notice how this contrasts with the standard boolean definition for tall people shown in Figure 8.4(*b*). In the latter, one is either tall or not and there must be a specific height that defines the boundary. The same is true for very tall. In the former, one's tallness increases with one's height until the value of 1 is reached.

Once set membership has been redefined in this way, it is possible to define a reasoning system based on techniques for combining distributions [Zadeh, 1979] (or see the papers in the journal *Fuzzy Sets and Systems*). Such reasoners have been applied in control systems for devices as diverse as trains and washing machines.

8.6 Summary

In this chapter we have shown that Bayesian statistics provide a good basis for reasoning under various kinds of uncertainty. We have also, though, talked about its weaknesses in complex real tasks, and so we have talked about ways in which it can be modified to work in practical domains. The thing that all of these modifications have in common is that they substitute, for the huge joint probability matrix that a pure Bayesian approach requires, a more structured representation of the facts that are relevant to a particular problem. They typically do this by combining probabilistic information with knowledge that is represented using one or more other representational mechanisms, such as rules or constraint networks.

Comparing these approaches for use in a particular problem-solving program is not always straightforward, since they differ along several dimensions, for example:

- They provide different mechanisms for describing the ways in which propositions are not independent of each other.
- They provide different techniques for representing ignorance.
- They differ substantially in the ease with which systems that use them can be built and in the computational complexity that the resulting systems exhibit.

We have also presented fuzzy logic as an alternative for representing some kinds of uncertain knowledge. Although there remain many arguments about the relative overall merits of the Bayesian and the fuzzy approaches, there is some evidence that they may both be useful in capturing different kinds of information. As an example, consider the proposition

John was pretty sure that Mary was seriously ill

Bayesian approaches naturally capture John's degree of certainty, while fuzzy techniques can describe the degree of Mary's illness.

Throughout all of this discussion, it is important to keep in mind the fact that although we have been discussing techniques for representing knowledge, there is another perspective from which what we have really been doing is describing ways of representing *lack* of knowledge. In this sense, the techniques we have described in this chapter are fundamentally different from the ones we talked about earlier. For example, the truth values that we manipulate in a logical system characterize the formulas that we write; certainty measures, on the other hand, describe the exceptions—the facts that do not appear anywhere in the formulas that we have written. The consequences of this distinction show up in the ways that we can interpret and manipulate the formulas that we write. The most important difference is that logical formulas can be treated as though they represent independent propositions. As we have seen throughout this chapter, uncertain assertions cannot. As a result, for example, while implication is transitive in logical systems, we often get into trouble in uncertain systems if we treat it as though it were (as we saw in our first treatment of the sprinkler and grass example). Another difference is that in logical systems it is necessary to find only a single proof to be able to assert the truth value of a proposition. All other proofs, if there are any, can safely be ignored. In uncertain systems, on the other hand, computing belief in a proposition requires that all available reasoning paths be followed and combined.

One final comment is in order before we end this discussion. You may have noticed throughout this chapter that we have not maintained a clear distinction among such concepts as probability, certainty, and belief. This is because although there has been a great deal of philosophical debate over the meaning of these various terms, there is no clear argreement on how best to interpret them if our goal is to create working programs. Although the idea that probability should be viewed as a measure of belief rather than as a summary of past experience is now quite widely held, we have chosen to avoid the debate in this presentation. Instead, we have used all those words with their everyday, undifferentiated meaning, and we have concentrated on providing simple descriptions of how several algorithms actually work. If you are interested in the philosophical issues, see, for example, Shafer [1976] and Pearl [1988].

Unfortunately, although in the last two chapters we have presented several important approaches to the problem of uncertainty management, we have barely scraped the surface of this area. For more information, see Kanal and Lemmer [1986], Kanal and Lemmer [1988], Kanal *et al.* [1989], Shafer and Pearl [1990], Clark [1990]. In particular, our list of specific techniques is by no means complete. For example, you may wish to look into probabilistic logic [Nilsson, 1986; Halpern, 1989], in which probability theory is combined with logic so that the truth value of a formula is a probability value (between 0 and 1) rather than a boolean value (TRUE or FALSE). Or you may wish to ask not what statistics can do for AI but rather what AI can do for statistics. In that case, see Gale [1986].

8.7 Exercises

1. Consider the following puzzle:

> A pea is placed under one of three shells, and the shells are then manipulated in such a fashion that all three appear to be equally likely to contain the pea. Nevertheless, you win a prize if you guess the correct shell, so you make a guess. The person running the game does know the correct shell, however, and uncovers one of the shells that you did not choose and that is empty. Thus, what remains are two shells: one you chose and one you did not choose. Furthermore, since the uncovered shell did not contain the pea, one of the two remaining shells does contain it. You are offered the opportunity to change your selection to the other shell. Should you?

Work through the conditional probabilities mentioned in this problem using Bayes' theorem. What do the results tell about what you should do?

2. Using MYCIN's rules for inexact reasoning, compute CF, MB, and MD of h_1 given three observations where

$$\begin{aligned} CF(h_1, o_1) &= 0.5 \\ CF(h_1, o_2) &= 0.3 \\ CF(h_1, o_3) &= -0.2 \end{aligned}$$

3. Show that MYCIN's combining rules satisfy the three properties we gave for them.

4. Consider the following set of propositions:

   patient has spots
   patient has measles
   patient has high fever
   patient has Rocky Mountain Spotted Fever
   patient has previously been innoculated against measles
   patient was recently bitten by a tick
   patient has an allergy

   (a) Create a network that defines the causal connections among these nodes.
   (b) Make it a Bayesian network by constructing the necessary conditional probability matrix.

5. Consider the same propositions again, and assume our task is to identify the patient's disease using Dempster-Shafer theory.

   (a) What is Θ?
   (b) Define a set of m functions that describe the dependencies among sources of evidence and elements of Θ.'
   (c) Suppose we have observed spots, fever, and a tick bite. In that case, what is our $Bel(\{RockyMountainSpottedFever\})$?

6. Define fuzzy sets that can be used to represent the list of propositions that we gave at the beginning of Section 8.5.

7. Consider again the ABC Murder story from Chapter 7. In our discussion of it there, we focused on the use of symbolic techniques for representing and using uncertain knowledge. Let's now explore the use of numeric techniques to solve the same problem. For each part below, show how knowledge could be represented. Whenever possible, show how it can be combined to produce a prediction of who committed the murder given at least one possible configuration of the evidence.

   (a) Use MYCIN-style rules and CF's. Example rules might include:

```
If (1) relative (x,y), and
   (2) on speaking terms (x,y),
then there is suggestive evidence (0.7) that
   will-lie-for (x,y)
```

(b) Use Bayesian networks. Represent as nodes such propositions as brother-in-law-lied, Cabot-at-ski-meet, and so forth.

(c) Use Dempster-Shafer theory. Examples of m's might be:

$$\begin{array}{llll} m_1 = & \{Abbott, Babbitt\} & (0.8) & \{beneficiaries\ in\ will\} \\ & \Theta & (0.2) & \\ & & & \\ m_2 = & \{Abbott, Cabot\} & (0.7) & \{in\ line\ for\ his\ job\} \\ & \Theta & (0.3) & \end{array}$$

(d) Use fuzzy logic. For example, you might want to define such fuzzy sets as honest people or greedy people and describe Abbott, Babbitt, and Cabot's memberships in those sets.

(e) What kinds of information are easiest (and hardest) to represent in each of these frameworks?

Chapter 9

Weak Slot-and-Filler Structures

In this chapter, we continue the discussion we began in Chapter 4 of slot-and-filler structures. Recall that we originally introduced them as a device to support property inheritance along *isa* and *instance* links. This is an important aspect of these structures. Monotonic inheritance can be performed substantially more efficiently with such structures than with pure logic, and nonmonotonic inheritance is easily supported. The reason that inheritance is easy is that the knowledge in slot-and-filler systems is structured as a set of entities and their attributes. This structure turns out to be a useful one for other reasons besides the support of inheritance, though, including:

- It indexes assertions by the entities they describe. More formally, it indexes binary predicates [such as *team(Three-Finger-Brown, Chicago-Cubs)*] by their first argument. As a result, retrieving the value for an attribute of an entity is fast.
- It makes it easy to describe properties of relations. To do this in a purely logical system requires some higher-order mechanisms.
- It is a form of object-oriented programming and has the advantages that such systems normally have, including modularity and ease of viewing by people.

We describe two views of this kind of structure: semantic nets and frames. We talk about the representations themselves and about techniques for reasoning with them. We do not say much, though, about the specific knowledge that the structures should contain. We call these "knowledge-poor" structures "weak," by analogy with the weak methods for problem solving that we discussed in Chapter 3. In the next chapter, we expand this discussion to include "strong" slot-and-filler structures, in which specific commitments to the content of the representation are made.

9.1 Semantic Nets

The main idea behind semantic nets is that the meaning of a concept comes from the ways in which it is connected to other concepts. In a semantic net, information is represented as a set of nodes connected to each other by a set of labeled arcs, which

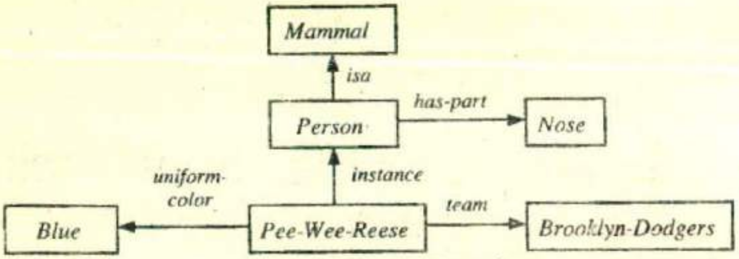


Figure 9.1: A Semantic Network

represent relationships among the nodes. A fragment of a typical semantic net is shown in Figure 9.1.

This network contains examples of both the *isa* and *instance* relations, as well as some other, more domain-specific relations like *team* and *uniform-color*. In this network, we could use inheritance to derive the additional relation

has-part(Pee-Wee-Reese, Nose)

9.1.1 Intersection Search

One of the early ways that semantic nets were used was to find relationships among objects by spreading activation out from each of two nodes and seeing where the activation met. This process is called *intersection search* [Quillian, 1968]. Using this process, it is possible to use the network of Figure 9.1 to answer questions such as "What is the connection between the Brooklyn Dodgers and blue?"[1] This kind of reasoning exploits one of the important advantages that slot-and-filler structures have over purely logical representations because it takes advantage of the entity-based organization of knowledge that slot-and-filler representations provide.

To answer more structured questions, however, requires networks that are themselves more highly structured. In the next few sections we expand and refine our notion of a network in order to support more sophisticated reasoning.

9.1.2 Representing Nonbinary Predicates

Semantic nets are a natural way to represent relationships that would appear as ground instances of binary predicates in predicate logic. For example, some of the arcs from Figure 9.1 could be represented in logic as

[1] Actually, to do this we need to assume that the inverses of the links we have shown also exist.

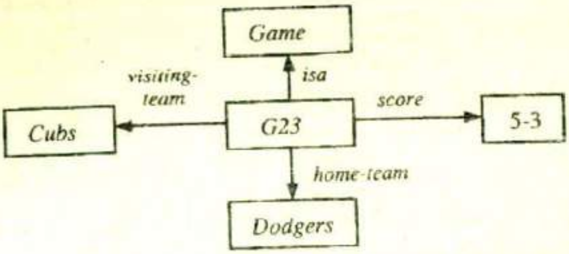


Figure 9.2: A Semantic Net for an *n*-Place Predicate

isa(Person, Mammal)
instance(Pee-Wee-Reese, Person)
team(Pee-Wee-Reese, Brooklyn-Dodgers)
uniform-color(Pee-Wee-Reese, Blue)

But the knowledge expressed by predicates of other arities can also be expressed in semantic nets. We have already seen that many unary predicates in logic can be thought of as binary predicates using some very general-purpose predicates, such as *isa* and *instance*. So, for example,

man(Marcus)

could be rewritten as

instance(Marcus, Man)

thereby making it easy to represent in a semantic net.

Three or more place predicates can also be converted to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe the relationship to this new object of each of the original arguments. For example, suppose we know that

score(Cubs, Dodgers, 5-3)

This can be represented in a semantic net by creating a node to represent the specific game and then relating each of the three pieces of information to it. Doing this produces the network shown in Figure 9.2.

This technique is particularly useful for representing the contents of a typical declarative sentence that describes several aspects of a particular event. The sentence

John gave the book to Mary

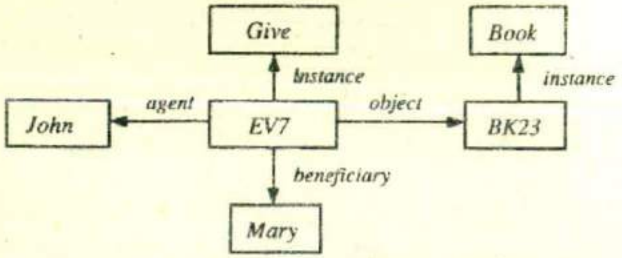
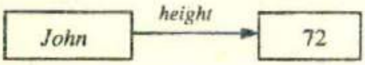


Figure 9.3: A Semantic Net Representing a Sentence

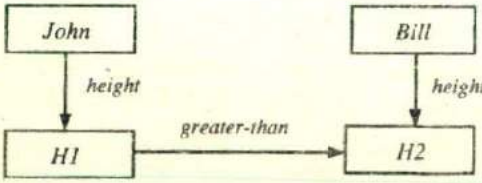
could be represented by the network shown in Figure 9.3.[2] In fact, several of the earliest uses of semantic nets were in English-understanding programs.

9.1.3 Making Some Important Distinctions

In the networks we have described so far, we have glossed over some distinctions that are important in reasoning. For example, there should be a difference between a link that defines a new entity and one that relates two existing entities. Consider the net



Both nodes represent objects that exist independently of their relationship to each other. But now suppose we want to represent the fact that John is taller than Bill, using the net

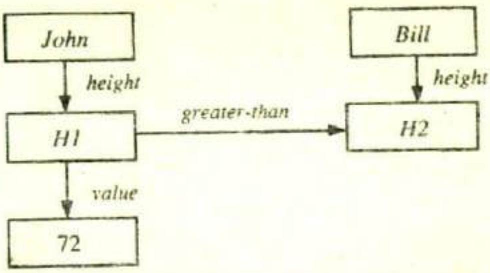


The nodes $H1$ and $H2$ are new concepts representing John's height and Bill's height, respectively. They are defined by their relationships to the nodes *John* and *Bill*. Using these defined concepts, it is possible to represent such facts as that John's height increased, which we could not do before. (The number 72 increased?)

Sometimes it is useful to introduce the arc *value* to make this distinction clear. Thus we might use the following net to represent the fact that John is 6 feet tall and that he is

[2]The node labeled *BK23* represents the particular book that was referred to by the phrase "the book." Discovering which particular book was meant by that phrase is similar to the problem of deciding on the correct referent for a pronoun, and it can be a very hard problem. These issues are discussed in Section 15.4.

taller than Bill:



The procedures that operate on nets such as this can exploit the fact that some arcs, such as *height*, define new entities, while others, such as *greater-than* and *value*, merely describe relationships among existing entities.

Another example of an important distinction we have missed is the difference between the properties of a node itself and the properties that a node simply holds and passes on to its instances. For example, it is a property of the node *Person* that it is a subclass of the node *Mammal*. But the node *Person* does not have as one of its parts a nose. Instances of the node *Person* do, and we want them to inherit it.

It is difficult to capture these distinctions without assigning more structure to our notions of node, link, and value. In the next section, when we talk about frame systems, we do that. But first, we discuss a network-oriented solution to a simpler problem; this solution illustrates what can be done in the network model but at what price in complexity.

9.1.4 Partitioned Semantic Nets

Suppose we want to represent simple quantified expressions in semantic nets. One way to do this is to *partition* the semantic net into a hierarchical set of *spaces*, each of which corresponds to the scope of one or more variables [Hendrix, 1977]. To see how this works, consider first the simple net shown in Figure 9.4(*a*). This net corresponds to the statement

The dog bit the mail carrier.

The nodes *Dogs*, *Bite*, and *Mail-Carrier* represent the classes of dogs, bitings, and mail carriers, respectively, while the nodes *d*, *b*, and *m* represent a particular dog, a particular biting, and a particular mail carrier. This fact can easily be represented by a single net with no partitioning.

But now suppose that we want to represent the fact

Every dog has bitten a mail carrier.

or, in logic:

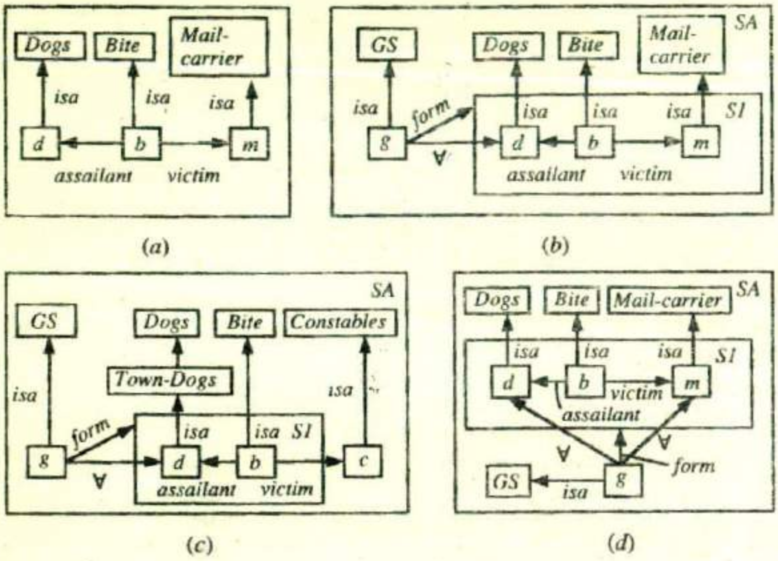


Figure 9.4: Using Partitioned Semantic Nets

$$\forall x : Dog(x) \rightarrow \exists y : Mail\text{-}Carrier(y) \wedge Bite(x, y)$$

To represent this fact, it is necessary to encode the scope of the universally quantified variable x. This can be done using partitioning as shown in Figure 9.4(b). The node g stands for the assertion given above. Node g is an instance of the special class GS of general statements about the world (i.e., those with universal quantifiers). Every element of GS has at least two attributes: a *form*, which states the relation that is being asserted, and one or more ∀ connections, one for each of the universally quantified variables. In this example, there is only one such variable d, which can stand for any element of the class *Dogs*. The other two variables in the form, b and m, are understood to be existentially quantified. In other words, for every dog d, there exists a biting event b, and a mail carrier m, such that d is the assailant of b and m is the victim.

To see how partitioning makes variable quantification explicit, consider next the similar sentence:

Every dog in town has bitten the constable.

The representation of this sentence is shown in Figure 9.4(c). In this net, the node c representing the victim lies outside the form of the general statement. Thus it is not viewed as an existentially quantified variable whose value may depend on the value of d. Instead it is interpreted as standing for a specific entity (in this case, a particular constable), just as do other nodes in a standard, nonpartitioned net.

Figure 9.4(*d*) shows how yet another similar sentence:

Every dog has bitten every mail carrier.

would be represented. In this case, *g* has two ∀ links, one pointing to *d*, which represents any dog, and one pointing to *m*, representing any mail carrier.

The spaces of a partitioned semantic net are related to each other by an inclusion hierarchy. For example, in Figure 9.4(*d*), space *S*1 is included in space *SA*. Whenever a search process operates in a partitioned semantic net, it can explore nodes and arcs in the space from which it starts and in other spaces that contain the starting point, but it cannot go downward, except in special circumstances, such as when a *form* arc is being traversed. So, returning to Figure 9.4(*d*), from node *d* it can be determined that *d* must be a dog. But if we were to start at the node *Dogs* and search for all known instances of dogs by traversing *isa* links, we would not find *d* since it and the link to it are in the space *S*1, which is at a lower level than space *SA*, which contains *Dogs*. This is important, since *d* does not stand for a particular dog; it is merely a variable that can be instantiated with a value that represents a dog.

9.1.5 The Evolution into Frames

The idea of a semantic net started out simply as a way to represent labeled connections among entities. But, as we have just seen, as we expand the range of problem-solving tasks that the representation must support, the representation itself necessarily begins to become more complex. In particular, it becomes useful to assign more structure to nodes as well as to links. Although there is no clear distinction between a semantic net and a frame system, the more structure the system has, the more likely it is to be termed a frame system. In the next section we continue our discussion of structured slot-and-filler representations by describing some of the most important capabilities that frame systems offer.

9.2 Frames

A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world. Sometimes a frame describes an entity in some absolute sense; sometimes it represents the entity from a particular point of view (as it did in the vision system proposal [Minsky, 1975] in which the term *frame* was first introduced). A single frame taken alone is rarely useful. Instead, we build frame systems out of collections of frames that are connected to each other by virtue of the fact that the value of an attribute of one frame may be another frame. In the rest of this section, we expand on this simple definition and explore ways that frame systems can be used to encode knowledge and support reasoning.

9.2.1 Frames as Sets and Instances

Set theory provides a good basis for understanding frame systems. Although not all frame systems are defined this way, we do so here. In this view, each frame represents either a class (a set) or an instance (an element of a class). To see how this works, consider the frame system shown in Figure 9.5, which is a slightly modified form of the network we showed in Figure 4.5. In this example, the frames *Person*, *Adult-Male*, *ML-Baseball Player* (corresponding to major league baseball players), *Pitcher*, and *ML-Baseball-Team* (for major league baseball team) are all classes. The frames *Pee-Wee-Reese* and *Brooklyn-Dodgers* are instances.

The *isa* relation that we have been using without a precise definition is in fact the *subset* relation. The set of adult males is a subset of the set of people. The set of major league baseball players is a subset of the set of adult males, and so forth. Our *instance* relation corresponds to the relation *element-of*. Pee Wee Reese is an element of the set of fielders. Thus he is also an element of all of the supersets of fielders, including major league baseball players and people. The transitivity of *isa* that we have taken for granted in our description of property inheritance follows directly from the transitivity of the subset relation.

Both the *isa* and *instance* relations have inverse attributes, which we call *subclasses* and *all-instances*. We do not bother to write them explicitly in our examples unless we need to refer to them. We assume that the frame system maintains them automatically, either explicitly or by computing them if necessary.

Because a class represents a set, there are two kinds of attributes that can be associated with it. There are attributes about the set itself, and there are attributes that are to be inherited by each element of the set. We indicate the difference between these two by prefixing the latter with an asterisk (*). For example, consider the class *ML-Baseball-Player*. We have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624 (i.e., there are 624 major league baseball players). We have listed five properties that all major league baseball players have (*height*, *bats*, *batting-average*, *team*, and *uniform-color*), and we have specified default values for the first three of them. By providing both kinds of slots, we allow a class both to define a set of objects and to describe a prototypical object of the set.

Sometimes, the distinction between a set and an individual instance may not seem clear. For example, the team *Brooklyn-Dodgers*, which we have described as an instance of the class of major league baseball teams, could be thought of as a set of players. In fact, notice that the value of the slot *players* is a set. Suppose, instead, that we want to represent the Dodgers as a class instead of an instance. Then its instances would be the individual players. It cannot stay where it is in the *isa* hierarchy; it cannot be a subclass of *ML-Baseball-Team*, because if it were, then its elements, namely the players, would also, by the transitivity of subclass, be elements of *ML-Baseball-Team*, which is not what we want to say. We have to put it somewhere else in the *isa* hierarchy. For example, we could make it a subclass of major league baseball players. Then its elements, the players, are also elements of *ML-Baseball-Player*, *Adult-Male*, and *Person*. That is acceptable. But if we do that, we lose the ability to inherit properties of the Dodgers from general information about baseball teams. We can still inherit attributes for the elements of the team, but we cannot inherit properties of the team as a whole, i.e., of the set of players. For example, we might like to know what the default size of the team is,

Person	
isa :	*Mammal*
cardinality :	6,000,000,000
* *handed* :	*Right*

Adult-Male	
isa :	*Person*
cardinality :	2,000,000,000
* *height* :	5-10

ML-Baseball-Player	
isa :	*Adult-Male*
cardinality :	624
* *height* :	6-1
* *bats* :	equal to handed
* *batting-average* :	.252
* *team* :	
* *uniform-color* :	

Fielder	
isa :	*ML-Baseball-Player*
cardinality :	376
* *batting-average* :	.262

Pee-Wee-Reese	
instance :	*Fielder*
height :	5-10
bats :	*Right*
batting-average :	.309
team :	*Brooklyn-Dodgers*
uniform-color :	*Blue*

ML-Baseball-Team	
isa :	*Team*
cardinality :	26
* *team-size* :	24
* *manager* :	

Brooklyn-Dodgers	
instance :	*ML-Baseball-Team*
team-size :	24
manager :	*Leo-Durocher*
players :	{*Pee-Wee-Reese*, ...}

Figure 9.5: A Simplified Frame System

that it has a manager, and so on. The easiest way to allow for this is to go back to the idea of the Dodgers as an instance of *ML-Baseball-Team*, with the set of players given as a slot value.

But what we have encountered here is an example of a more general problem. A class is a set, and we want to be able to talk about properties that its elements possess. We want to use inheritance to infer those properties from general knowledge about the set. But a class is also an entity in itself. It may possess properties that belong not to the individual instances but rather to the class as a whole. In the case of *Brooklyn-Dodgers*, such properties included team size and the existence of a manager. We may even want to inherit some of these properties from a more general kind of set. For example, the Dodgers can inherit a default team size from the set of all major league baseball teams. To support this, we need to view a class as two things simultaneously: a subset (*isa*) of a larger class that also contains its elements and an instance (*instance*) of a class of sets, from which it inherits its set-level properties.

To make this distinction clear, it is useful to distinguish between regular classes, whose elements are individual entities, and *metaclasses*, which are special classes whose elements are themselves classes. A class is now an element of (*instance*) some class (or classes) as well as a subclass (*isa*) of one or more classes. A class inherits properties from the class of which it is an instance, just as any instance does. In addition, a class passes inheritable properties down from its superclasses to its instances.

Let's consider an example. Figure 9.6 shows how we could represent teams as classes using this distinction. Figure 9.7 shows a graphic view of the same classes. The most basic metaclass is the class *Class*. It represents the set of all classes. All classes are instances of it, either directly or through one of its subclasses. In the example, *Team* is a subclass (subset) of *Class* and *ML-Baseball-Team* is a subclass of *Team*. The class *Class* introduces the attribute *cardinality*, which is to be inherited by all instances of *Class* (including itself). This makes sense since all the instances of *Class* are sets and all sets have a cardinality.

Team represents a subset of the set of all sets, namely those whose elements are sets of players on a team. It inherits the property of having a cardinality from *Class*. *Team* introduces the attribute *team-size*, which all its elements possess. Notice that *team-size* is like *cardinality* in that it measures the size of a set. But it applies to something different: *cardinality* applies to sets of sets and is inherited by all elements of *Class*. The slot *team-size* applies to the elements of those sets that happen to be teams. Those elements are sets of individuals.

ML-Baseball-Team is also an instance of *Class*, since it is a set. It inherits the property of having a cardinality from the set of which it is an instance, namely *Class*. But it is a subset of *Team*. All of its instances will have the property of having a *team-size* since they are also instances of the superclass *Team*. We have added at this level the additional fact that the default team size is 24, so all instances of *ML-Baseball-Team* will inherit that as well. In addition, we have added the inheritable slot *manager*.

Brooklyn-Dodgers is an instance of a *ML-Baseball-Team*. It is not an instance of *Class* because its elements are individuals, not sets. *Brooklyn-Dodgers* is a subclass of *ML-Baseball-Player* since all of its elements are also elements of that set. Since it is an instance of a *ML-Baseball-Team*, it inherits the properties *team-size* and *manager*, as well as their default values. It specifies a new attribute *uniform-color*, which is to be inherited by all of its instances (who will be individual players).

```
Class
    instance :          Class
    isa :               Class
    * cardinality :

Team
    instance :          Class
    isa :               Class
    cardinality :       {the number of teams that exist}
    * team-size :       {each team has a size}

ML-Baseball-Team
    instance :          Class
    isa :               Team
    cardinality :       26 {the number of baseball teams that exist}
    * team-size :       24 {default 24 players on a team}
    * manager :

Brooklyn-Dodgers
    instance :          ML-Baseball-Team
    isa :               ML-Baseball-Player
    team-size :         24
    manager :           Leo-Durocher
    * uniform-color :   Blue

Pee-Wee-Reese
    instance :          Brooklyn-Dodgers
    instance :          Fielder
    uniform-color :     Blue
    batting-average :   .309
```

Figure 9.6: Representing the Class of All Teams as a Metaclass

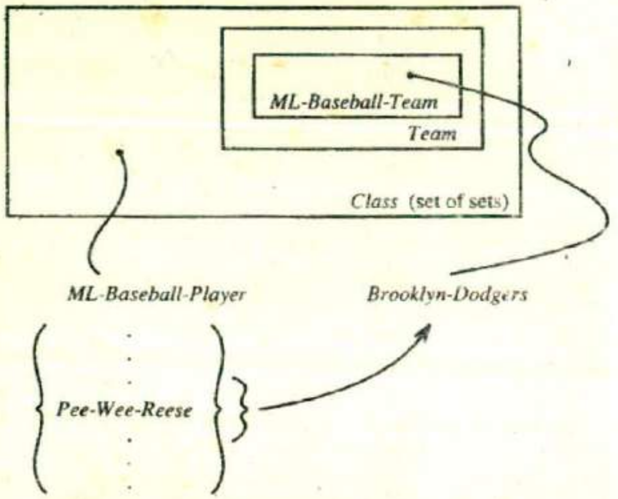


Figure 9.7: Classes and Metaclasses

Finally, ***Pee-Wee-Reese*** is an instance of ***Brooklyn-Dodgers***. That makes him also, by transitivity up *isa* links, an instance of ***ML-Baseball-Player***. But recall that in our earlier example we also used the class *Fielder*, to which we attached the fact that fielders have above-average batting averages. To allow that here, we simply make Pee Wee an instance of *Fielder* as well. He will thus inherit properties from both *Brooklyn-Dodgers* and from *Fielder*, as well as from the classes above these. We need to guarantee that when multiple inheritance occurs, as it does here, that it works correctly. Specifically, in this case, we need to assure that *batting-average* gets inherited from *Fielder* and not from ***ML-Baseball-Player*** through ***Brooklyn-Dodgers***. We return to this issue in Section 9.2.5.

In all the frame systems we illustrate, all classes are instances of the metaclass *Class*. As a result, they all have the attribute *cardinality*. We leave the class *Class*, the *isa* links to it, and the attribute *cardinality* out of our descriptions of our examples, though, unless there is some particular reason to include them.

Every class is a set. But not every set should be described as a class. A class describes a set of entities that share significant properties. In particular, the default information associated with a class can be used as a basis for inferring values for the properties of its individual elements. So there is an advantage to representing as a class those sets for which membership serves as a basis for nonmonotonic inheritance. Typically, these are sets in which membership is not highly ephemeral. Instead, membership is based on some fundamental structural or functional properties. To see the difference, consider the following sets:

- People
- People who are major league baseball players

- People who are on my plane to New York

The first two sets can be advantageously represented as classes, with which a substantial number of inheritable attributes can be associated. The last, though, is different. The only properties that all the elements of that set probably share are the definition of the set itself and some other properties that follow from the definition (e.g., they are being transported from one place to another). A simple set, with some associated assertions, is adequate to represent these facts; nonmonotonic inheritance is not necessary.

9.2.2 Other Ways of Relating Classes to Each Other

We have talked up to this point about two ways in which classes (sets) can be related to each other. $Class_1$ can be a subset of $Class_2$. Or, if $Class_2$ is a metaclass, then $Class_1$ can be an instance of $Class_2$. But there are other ways that classes can be related to each other, corresponding to ways that sets of objects in the world can be related.

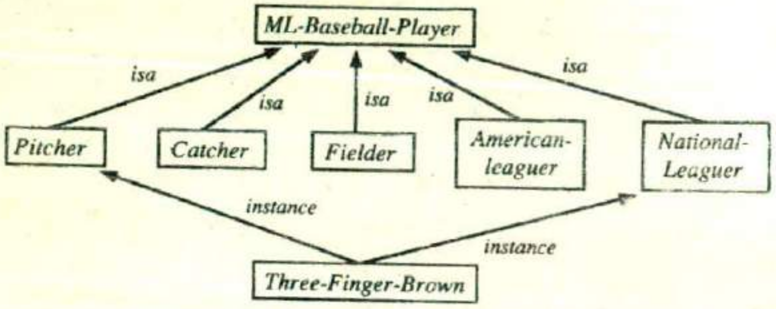
One such relationship is *mutually-disjoint-with*, which relates a class to one or more other classes that are guaranteed to have no elements in common with it. Another important relationship is *is-covered-by*, which relates a class to a set of subclasses, the union of which is equal to it. If a class *is-covered-by* a set S of mutually disjoint classes, then S is called a *partition* of the class.

For examples of these relationships, consider the classes shown in Figure 9.8, which represent two orthogonal ways of decomposing the class of major league baseball players. Everyone is either a pitcher, a catcher, or a fielder (and no one is more than one of these). In addition, everyone plays in either the National League or the American League, but not both.

9.2.3 Slots as Full-Fledged Objects

So far, we have provided a way to describe sets of objects and individual objects, both in terms of attributes and values. Thus we have made extensive use of attributes, which we have represented as slots attached to frames. But it turns out that there are several reasons why we would like to be able to represent attributes explicitly and describe their properties. Some of the properties we would like to be able to represent and use in reasoning include:

- The classes to which the attribute can be attached, i.e., for what classes does it make sense? For example, weight makes sense for physical objects but not for conceptual ones (except in some metaphorical sense).
- Constraints on either the type or the value of the attribute. For example, the age of a person must be a numeric quantity measured in some time frame, and it must be less than the ages of the person's biological parents.
- A value that all instances of a class must have by the definition of the class.
- A default value for the attribute.
- Rules for inheriting values for the attribute. The usual rule is to inherit down *isa* and *instance* links. But some attributes inherit in other ways. For example, *last-name* inherits down the *child-of* link.



ML-Baseball-Player	
is-covered-by :	{*Pitcher, Catcher, Fielder*}, {*American-Leaguer, National-Leaguer*}
Pitcher	
isa :	*ML-Baseball-Player*
mutually-disjoint-with:	{*Catcher, Fielder*}
Catcher	
isa :	*ML-Baseball-Player*
mutually-disjoint-with:	{*Pitcher, Fielder*}
Fielder	
isa :	*ML-Baseball-Player*
mutually-disjoint-with:	{*Pitcher, Catcher*}
American-Leaguer	
isa :	*ML-Baseball-Player*
mutually-disjoint-with:	{*National-Leaguer*}
National-Leaguer	
isa :	*ML-Baseball-Player*
mutually-disjoint-with:	{*American-Leaguer*}
Three-Finger-Brown	
instance :	*Pitcher*
instance :	*National-Leaguer*

Figure 9.8: Representing Relationships among Classes

- Rules for computing a value separately from inheritance. One extreme form of such a rule is a procedure written in some procedural programming language such as LISP.
- An inverse attribute.
- Whether the slot is single-valued or multivalued.

In order to be able to represent these attributes of attributes, we need to describe attributes (slots) as frames. These frames will be organized into an *isa* hierarchy, just as any other frames are, and that hierarchy can then be used to support inheritance of values for attributes of slots. Before we can describe such a hierarchy in detail, we need to formalize our notion of a slot.

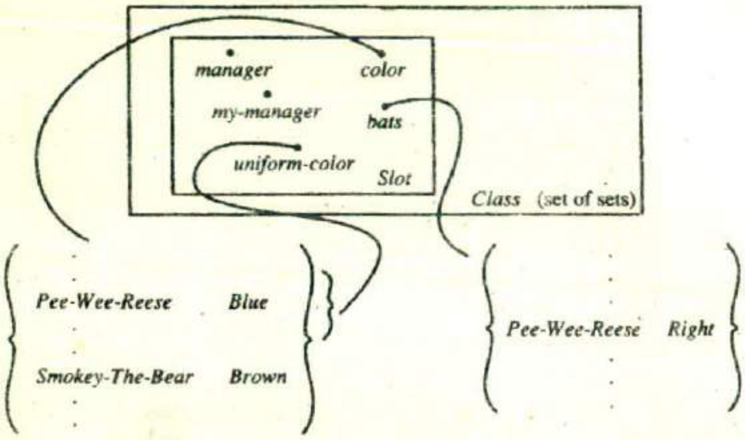
A slot is a relation. It maps from elements of its domain (the classes for which it makes sense) to elements of its range (its possible values). A relation is a set of ordered pairs. Thus it makes sense to say that one relation (R_1) is a subset of another (R_2). In that case, R_1 is a specialization of R_2, so in our terminology $isa(R_1, R_2)$. Since a slot is a set, the set of all slots, which we will call *Slot*, is a metaclass. Its instances are slots, which may have subslots.

Figures 9.9 and 9.10 illustrate several examples of slots represented as frames. *Slot* is a metaclass. Its instances are slots (each of which is a set of ordered pairs). Associated with the metaclass are attributes that each instance (i.e., each actual slot) will inherit. Each slot, since it is a relation, has a domain and a range. We represent the domain in the slot labeled *domain*. We break up the representation of the range into two parts: *range* gives the class of which elements of the range must be elements; *range-constraint* contains a logical expression that further constrains the range to be elements of *range* that also satisfy the constraint. If *range-constraint* is absent, it is taken to be TRUE. The advantage to breaking the description apart into these two pieces is that type checking is much cheaper than is arbitrary constraint checking, so it is useful to be able to do it separately and early during some reasoning processes.

The other slots do what you would expect from their names. If there is a value for *definition*, it must be propagated to all instances of the slot. If there is a value for *default*, that value is inherited to all instances of the slot unless there is an overriding value. The attribute *transfers-through* lists other slots from which values for this slot can be derived through inheritance. The *to-compute* slot contains a procedure for deriving its value. The *inverse* attribute contains the inverse of the slot. Although in principle all slots have inverses, sometimes they are not useful enough in reasoning to be worth representing. And *single-valued* is used to mark the special cases in which the slot is a function and so can have only one value.

Of course, there is no advantage to representing these properties of slots if there is no reasoning mechanism that exploits them. In the rest of our discussion, we assume that the frame-system interpreter knows how to reason with all of these slots of slots as part of its built-in reasoning capability. In particular, we assume that it is capable of performing the following reasoning actions:

- Consistency checking to verify that when a slot value is added to a frame
  - The slot makes sense for the frame. This relies on the *domain* attribute of the slot.



Slot

isa :	*Class*
instance :	*Class*
* *domain* :	
* *range* :	
* *range-constraint* :	
* *definition* :	
* *default* :	
* *transfers-through* :	
* *to-compute* :	
* *inverse* :	
* *single-valued* :	

manager

instance :	*Slot*
domain :	*ML-Baseball-Team*
range :	*Person*
range-constraint :	λx (*baseball-experience* x.*manager*)
default :	
inverse :	*manager-of*
single-valued :	TRUE

Figure 9.9: Representing Slots as Frames, I

my-manager

instance :	*Slot*
domain :	*ML-Baseball-Player*
range :	*Person*
range-constraint :	λx (*baseball-experience* x.*my-manager*)
to-compute :	λx (x.*team*).*manager*
single-valued :	TRUE

color

instance :	*Slot*
domain :	*Physical-Object*
range :	*Color-Set*
transfers-through	*top-level-part-of*
visual-salience :	*High*
single-valued :	FALSE

uniform-color

instance :	*Slot*
isa :	*color*
domain :	*team-player*
range :	*Color-Set*
range-constraint :	not *Pink*
visual-salience :	*High*
single-valued :	FALSE

bats

instance :	*Slot*
domain :	*ML-Baseball-Player*
range :	{*Left*, *Right*, *Switch*}
to-compute	λx x.*handed*
single-valued :	TRUE

Figure 9.10: Representing Slots as Frames. II

  - The value is a legal value for the slot. This relies on the *range* and *range-constraints* attributes.

- Maintenance of consistency between the values for slots and their inverses whenever one is updated.

- Propagation of *definition* values along *isa* and *instance* links.

- Inheritance of *default* values along *isa* and *instance* links.

- Computation of a value of a slot as needed. This relies on the *to-compute* and *transfers-through* attributes.

- Checking that only a single value is asserted for *single-valued* slots. This is usually done by replacing an old value by the new one when it is asserted. An alternative is to force explicit retraction of the old value and to signal a contradiction if a new value is asserted when another is already there.

There is something slightly counterintuitive about this way of defining slots. We have defined the properties *range-constraint* and *default* as parts of a slot. But we often think of them as being properties of a slot associated with a particular class. For example, in Figure 9.5, we listed two defaults for the *batting-average* slot, one associated with major league baseball players and one associated with fielders. Figure 9.11 shows how this can be represented correctly, by creating a specialization of *batting-average* that can be associated with a specialization of *ML-Baseball-Player* to represent the more specific information that is known about the specialized class. This seems cumbersome. It is natural, though, given our definition of a slot as a relation. There are really two relations here, one a specialization of the other. And below we will define inheritance so that it looks for values of either the slot it is given or any of that slot's generalizations.

Unfortunately, although this model of slots is simple and it is internally consistent, it is not easy to use. So we introduce some notational shorthand that allows the four most important properties of a slot (domain, range, definition, and default) to be defined implicitly by how the slot is used in the definitions of the classes in its domain. We describe the domain implicitly to be the class where the slot appears. We describe the range and any range constraints with the clause MUST BE, as the value of an inherited slot. Figure 9.12 shows an example of this notation. And we describe the definition and the default, if they are present, by inserting them as the value of the slot when it appears. The two will be distinguished by prefixing a definitional value with an asterisk (*). We then let the underlying bookkeeping of the frame system create the frames that represent slots as they are needed.

Now let's look at examples of how these slots can be used. The slots *bats* and *my-manager* illustrate the use of the *to-compute* attribute of a slot. The variable x will be bound to the frame to which the slot is attached. We use the dot notation to specify the value of a slot of a frame. Specifically, $x.y$ describes the value(s) of the y slot of frame x. So we know that to compute a frame's value for *my-manager*, it is necessary to find the frame's value for *team*, then find the resulting team's manager. We have simply composed two slots to form a new one.[3] Computing the value of the *bats* slot is even simpler. Just go get the value of the *handed* slot.

[3]Notice that since slots are relations rather than functions, their composition may return a set of values.

batting-average

instance :	*Slot*
domain :	*ML-Baseball-Player*
range :	*Number*
range-constraint :	$\lambda x\ (0 \leq x.\mathit{range\text{-}constraint} < 1)$
default :	.252
single-valued :	TRUE

fielder-batting-average

instance :	*Slot*
isa :	*batting-average*
domain :	*Fielder*
range :	*Number*
range-constraint :	$\lambda x\ (0 \leq x.\mathit{range\text{-}constraint} \leq 1)$
default :	.262
single-valued :	TRUE

Figure 9.11: Associating Defaults with Slots

ML-Baseball-Player

bats :	MUST BE {*Left*, *Right*, *Switch*}

Figure 9.12: A Shorthand Notation for Slot-Range Specification

The *manager* slot illustrates the use of a range constraint. It is stated in terms of a variable x, which is bound to the frame whose *manager* slot is being described. It requires that any manager be not only a person but someone with baseball experience. It relies on the domain-specific function *baseball-experience*, which must be defined somewhere in the system.

The slots *color* and *uniform-color* illustrate the arrangement of slots in an *isa* hierarchy. The relation *color* is a fairly general one that holds between physical objects and colors. The attribute *uniform-color* is a restricted form of *color* that applies only between team players and the colors that are allowed for team uniforms (anything but pink). Arranging slots in a hierarchy is useful for the same reason that arranging anything else in a hierarchy is: it supports inheritance. In this example, the general slot *color* is known to have high visual salience. The more specific slot *uniform-color* then inherits this property, so it too is known to have high visual salience.

The slot *color* also illustrates the use of the *transfers-through* slot, which defines a way of computing a slot's value by retrieving it from the same slot of a related object. In this example, we used *transfers-through* to capture the fact that if you take an object and chop it up into several top level parts (in other words, parts that are not contained inside each other), then they will all be the same color. For example, the arm of a sofa is the same color as the sofa. Formally, what *transfers-through* means in this example is

```
John
    height :    72

Bill
    height :
```

Figure 9.13: Representing Slot-Values

$color(x, y) \land top\text{-}level\text{-}part\text{-}of(z, x) \rightarrow color(z, y)$

In addition to these domain-independent slot attributes, slots may have domain-specific properties that support problem solving in a particular domain. Since these slots are not treated explicitly by the frame-system interpreter, they will be useful precisely to the extent that the domain problem solver exploits them.

9.2.4 Slot-Values as Objects

In the last section, we reified the notion of a slot by making it an explicit object that we could make assertions about. In some sense this was not necessary. A finite relation can be completely described by listing its elements. But in practical knowledge-based systems one often does not have that list. So it can be very important to be able to make assertions about the list without knowing all of its elements. Reification gave us a way to do this.

The next step along this path is to do the same thing to a particular attribute-value (an instance of a relation) that we did to the relation itself. We can reify it and make it an object about which assertions can be made. To see why we might want to do this, let us return to the example of John and Bill's height that we discussed in Section 9.1.3. Figure 9.13 shows a frame-based representation of some of the facts. We could easily record Bill's height if we knew it. Suppose, though, that we do not know it. All we know is that John is taller than Bill. We need a way to make an assertion about the value of a slot without knowing what that value is. To do that, we need to view the slot and its value as an object.

We could attempt to do this the same way we made slots themselves into objects, namely by representing them explicitly as frames. There seems little advantage to doing that in this case, though, because the main advantage of frames does not apply to slot values: frames are organized into an *isa* hierarchy and thus support inheritance. There is no basis for such an organization of slot values. So instead, we augment our value representation language to allow the value of a slot to be stated as either or both of:

- A value of the type required by the slot.
- A logical constraint on the value. This constraint may relate the slot's value to the values of other slots or to domain constants.

John
  height : 72; λx (x.*height* > *Bill.height*)

Bill
  height : λx (x.*height* < *John.height*)

Figure 9.14: Representing Slot-Values with Lambda Notation

If we do this to the frames of Figure 9.13, then we get the frames of Figure 9.14. We again use the lambda notation as a way to pick up the name of the frame that is being described.

9.2.5 Inheritance Revisited

In Chapter 4, we presented a simple algorithm for inheritance. But that algorithm assumed that the *isa* hierarchy was a tree. This is often not the case. To support flexible representations of knowledge about the world, it is necessary to allow the hierarchy to be an arbitrary directed acyclic graph (DAG). We know that acyclic graphs are adequate because *isa* corresponds to the subset relation. Hierarchies that are not trees are called *tangled hierarchies*. Tangled hierarchies require a new inheritance algorithm. In the rest of this section, we discuss an algorithm for inheriting values for single-valued slots in a tangled hierarchy. We leave the problem of inheriting multivalued slots as an exercise.

Consider the two examples shown in Figure 9.15 (in which we return to a network notation to make it easy to visualize the *isa* structure). In Figure 9.15(*a*), we want to decide whether *Fifi* can fly. The correct answer is no. Although birds in general can fly, the subset of birds, ostriches, does not. Although the class *Pet-Bird* provides a path from *Fifi* to *Bird* and thus to the answer that *Fifi* can fly, it provides no information that conflicts with the special case knowledge associated with the the class *Ostrich*, so it should have no affect on the answer. To handle this case correctly, we need an algorithm for traversing the *isa* hierarchy that guarantees that specific knowledge will always dominate more general facts.

In Figure 9.15(*b*), we return to a problem we discussed in Section 7.2.1, namely determining whether Dick is a pacifist. Again, we must traverse multiple *instance* links and more than one answer can be found along the paths. But in this case, there is no well-founded basis for choosing one answer over the other. The classes that are associated with the candidate answers are incommensurate with each other in the partial ordering that is defined by the DAG formed by the *isa* hierarchy. Just as we found that in Default Logic this theory had two extensions and there was no principled basis for choosing between them, what we need here is an inheritance algorithm that reports the ambiguity; we do not want an algorithm that finds one answer (arbitrarily) and stops without noticing the other.

One possible basis for a new inheritance algorithm is path length. This can be implemented by executing a breadth-first search, starting with the frame for which a slot value is needed. Follow its *instance* links, then follow *isa* links upward. If a path produces a value, it can be terminated, as can all other paths once their length

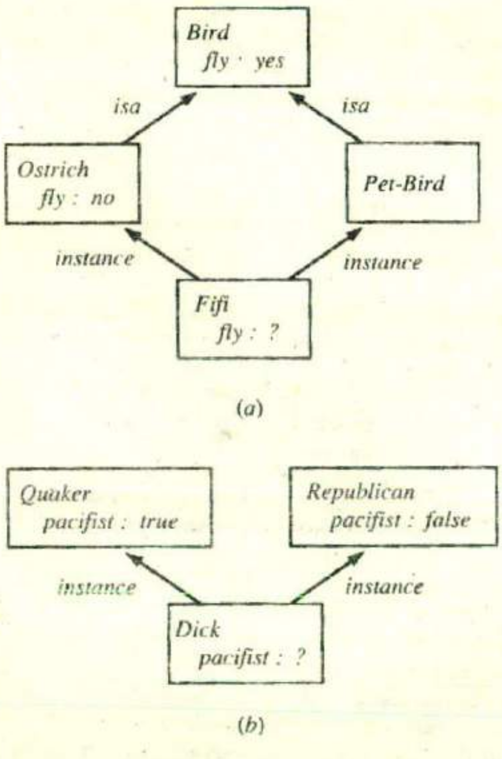


Figure 9.15: Tangled Hierarchies

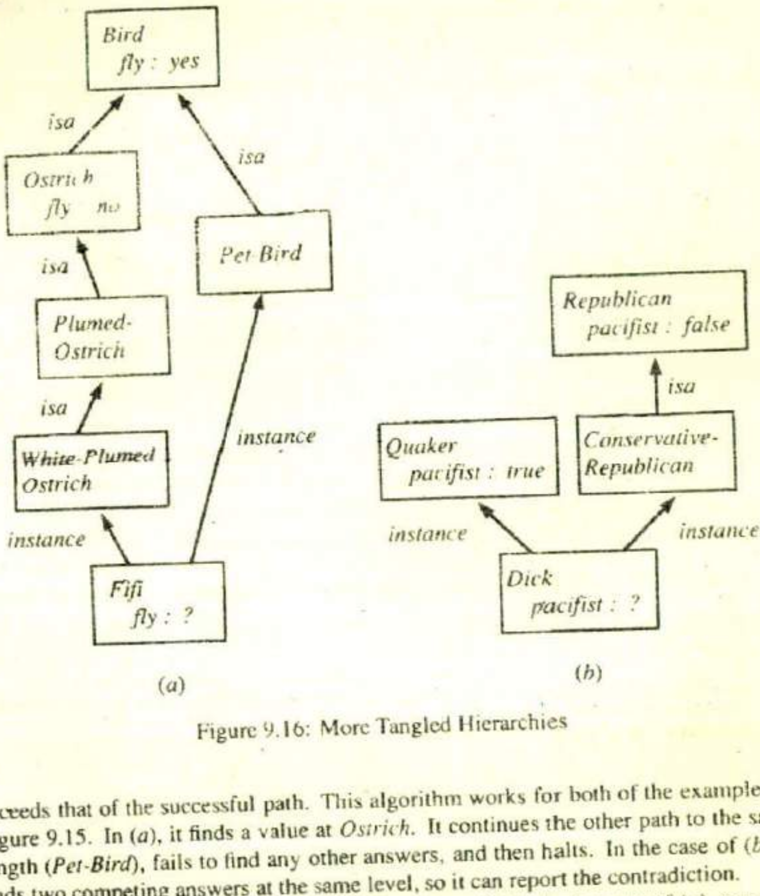


Figure 9.16: More Tangled Hierarchies

exceeds that of the successful path. This algorithm works for both of the examples in Figure 9.15. In (*a*), it finds a value at *Ostrich*. It continues the other path to the same length (*Pet-Bird*), fails to find any other answers, and then halts. In the case of (*b*), it finds two competing answers at the same level, so it can report the contradiction.

But now consider the examples shown in Figure 9.16. In the case of (*a*), our new algorithm reaches *Bird* (via *Pet-Bird*) before it reaches *Ostrich*. So it reports that *Fifi* can fly. In the case of (*b*), the algorithm reaches *Quaker* and stops without noticing a contradiction. The problem is that path length does not always correspond to the level of generality of a class. Sometimes what it really corresponds to is the degree of elaboration of classes in the knowledge base. If some regions of the knowledge base have been elaborated more fully than others, then their paths will tend to be longer. But this should not influence the result of inheritance if no new information about the desired attribute has been added.

The solution to this problem is to base our inheritance algorithm not on path length but on the notion of *inferential distance* [Touretzky, 1986], which can be defined as follows:

> *Class*$_1$ is closer to *Class*$_2$ than to *Class*$_3$ if and only if *Class*$_1$ has an inference path through *Class*$_2$ to *Class*$_3$ (in other words, *Class*$_2$ is between *Class*$_1$ and *Class*$_3$).

Notice that inferential distance defines only a partial ordering. Some classes are incommensurate with each other under it.

We can now define the result of inheritance as follows: The set of competing values for a slot S in a frame F contains all those values that

- Can be derived from some frame X that is above F in the *isa* hierarchy
- Are not contradicted by some frame Y that has a shorter inferential distance to F than X does

Notice that under this definition competing values that are derived from incommensurate frames continue to compete.

Using this definition, let us return to our examples. For Figure 9.15(*a*), we had two candidate classes from which to get an answer. But *Ostrich* has a shorter inferential distance to *Fifi* than *Bird* does, so we get the single answer no. For Figure 9.15(*b*), we get two answers, and neither is closer to *Dick* than the other, so we correctly identify a contradiction. For Figure 9.16(*a*), we get two answers, but again *Ostrich* has a shorter inferential distance to *Fifi* than *Bird* does. The significant thing about the way we have defined inferential distance is that as long as *Ostrich* is a subclass of *Bird*, it will be closer to all its instances than *Bird* is, no matter how many other classes are added to the system. For Figure 9.16(*b*), we again get two answers and again neither is closer to *Dick* than the other.

There are several ways that this definition can be implemented as an inheritance algorithm. We present a simple one. It can be made more efficient by caching paths in the hierarchy, but we do not do that here.

Algorithm: Property Inheritance

To retrieve a value V for slot S of an instance F do:

1. Set *CANDIDATES* to empty.
2. Do breadth-first or depth-first search up the *isa* hierarchy from F, following all *instance* and *isa* links. At each step, see if a value for S or one of its generalizations is stored.
    (a) If a value is found, add it to *CANDIDATES* and terminate that branch of the search.
    (b) If no value is found but there are *instance* or *isa* links upward, follow them.
    (c) Otherwise, terminate the branch.
3. For each element C of *CANDIDATES* do:
    (a) See if there is any other element of *CANDIDATES* that was derived from a class closer to F than the class from which C came.

   (b) If there is, then, remove *C* from *CANDIDATES*.

4. Check the cardinality of *CANDIDATES*:
   (a) If it is 0, then report that no value was found.
   (b) If it is 1, then return the single element of *CANDIDATES* as *V*.
   (c) If it is greater than 1, report a contradiction.

This algorithm is guaranteed to terminate because the *isa* hierarchy is represented as an acyclic graph.

9.2.6 Frame Languages

The idea of a frame system as a way to represent declarative knowledge has been encapsulated in a series of frame-oriented knowledge representation languages, whose features have evolved and been driven by an increased understanding of the sort of representation issues we have been discussing. Examples of such languages include KRL [Bobrow and Winograd, 1977], FRL [Roberts and Goldstein, 1977], RLL [Greiner and Lenat, 1980], KL-ONE [Brachman, 1979; Brachman and Schmolze, 1985], KRYPTON [Brachman *et al.*, 1985], NIKL [Kaczmarek *et al.*, 1986], CYCL [Lenat and Guha, 1990], conceptual graphs [Sowa, 1984], THEO [Mitchell *et al.*, 1989], and FRAMEKIT [Nyberg, 1988]. Although not all of these systems support all of the capabilities that we have discussed, the more modern of these systems permit elaborate and efficient representation of many kinds of knowledge. Their reasoning methods include most of the ones described here, plus many more, including subsumption checking, automatic classification, and various methods for consistency maintenance.

9.3 Exercises

1. Construct semantic net representations for the following:
   (a) *Pompeian*(*Marcus*), *Blacksmith*(*Marcus*)
   (b) Mary gave the green flowered vase to her favorite cousin.

2. Suppose we want to use a semantic net to discover relationships that could help in disambiguating the word "bank" in the sentence

   John went downtown to deposit his money in the bank

   The financial institution meaning for bank should be preferred over the river bank meaning.

   (a) Construct a semantic net that contains representations for the relevant concepts.
   (b) Show how intersection search could be used to find the connection between the correct meaning for bank and the rest of the sentence more easily than it can find a connection with the incorrect meaning.

3. Construct partitioned semantic net representations for the following:

   (a) Every batter hit a ball

   (b) All the batters like the pitcher.

4. Construct one consistent frame representation of all the baseball knowledge that was described in this chapter. You will need to choose between the two representations for team that we considered.

5. Modify the property inheritance algorithm of Section 9.2 to work for multiple-valued attributes, such as the attribute *believes-in-principles*, defined as follows:

believes-in-principles	
instance :	*Slot*
domain :	*Person*
range :	*Philosophical-Principles*
single-valued :	FALSE

6. Define the value of a multiple-valued slot *S* of class *C* to be the union of the values that are found for *S* and all its generalizations at *C* and all its generalizations. Modify your technique to allow a class to exclude specific values that are associated with one or more of its superclasses.

7. Pick a problem area and represent some knowledge about it the way we represented baseball knowledge in this chapter.

Chapter 10

Strong Slot-and-Filler Structures

The slot-and-filler structures described in the previous chapter are very general. Individual semantic networks and frame systems may have specialized links and inference procedures, but there are no hard and fast rules about what kinds of objects and links are good in general for knowledge representation. Such decisions are left up to the builder of the semantic network or frame system.

The three structures discussed in this chapter, *conceptual dependency*, *scripts*, and *CYC*, on the other hand, embody specific notions of what types of objects and relations are permitted. They stand for powerful theories of how AI programs can represent and use knowledge about common situations.

10.1 Conceptual Dependency

Conceptual dependency (often nicknamed CD) is a theory of how to represent the kind of knowledge about events that is usually contained in natural language sentences. The goal is to represent the knowledge in a way that

- Facilitates drawing inferences from the sentences.
- Is independent of the language in which the sentences were originally stated.

Because of the two concerns just mentioned, the CD representation of a sentence is built not out of primitives corresponding to the words used in the sentence, but rather out of conceptual primitives that can be combined to form the meanings of words in any particular language. The theory was first described in Schank [1973] and was further developed in Schank [1975]. It has since been implemented in a variety of programs that read and understand natural language text. Unlike semantic nets, which provide only a structure into which nodes representing information at any level can be placed, conceptual dependency provides both a structure and a specific set of primitives, at a particular level of granularity, out of which representations of particular pieces of information can be constructed.

$$\begin{tikzcd}
 & & & & \text{man} \\
\text{I} \arrow[r, Leftrightarrow, "p"] & \text{ATRANS} & \text{book} \arrow[l, "o"'] & \cdot \arrow[l, "R"'] \arrow[ur, "to"] & \\
 & & & & \text{I} \arrow[ul, "from"]
\end{tikzcd}$$

where the symbols have the following meanings:

- Arrows indicate direction of dependency.
- Double arrow indicates two way link between actor and action.
- p indicates past tense.
- ATRANS is one of the primitive acts used by the theory. It indicates transfer of possession.
- o indicates the object case relation.
- R indicates the recipient case relation.

Figure 10.1: A Simple Conceptual Dependency Representation

As a simple example of the way knowledge is represented in CD, the event represented by the sentence

I gave the man a book.

would be represented as shown in Figure 10.1.

In CD, representations of actions are built from a set of primitive acts. Although there are slight differences in the exact set of primitive actions provided in the various sources on CD, a typical set is the following, taken from Schank and Abelson [1977]:

ATRANS	Transfer of an abstract relationship (e.g., give)
PTRANS	Transfer of the physical location of an object (e.g., go)
PROPEL	Application of physical force to an object (e.g., push)
MOVE	Movement of a body part by its owner (e.g., kick)
GRASP	Grasping of an object by an actor (e.g., clutch)
INGEST	Ingestion of an object by an animal (e.g., eat)
EXPEL	Expulsion of something from the body of an animal (e.g., cry)
MTRANS	Transfer of mental information (e.g., tell)
MBUILD	Building new information out of old (e.g., decide)
SPEAK	Production of sounds (e.g., say)
ATTEND	Focusing of a sense organ toward a stimulus (e.g., listen)

A second set of CD building blocks is the set of allowable dependencies among the conceptualizations described in a sentence. There are four primitive conceptual categories from which dependency structures can be built. These are

ACTs	Actions
PPs	Objects (picture producers)
AAs	Modifiers of actions (action aiders)
PAs	Modifiers of PPs (picture aiders)

In addition, dependency structures are themselves conceptualizations and can serve as components of larger dependency structures.

The dependencies among conceptualizations correspond to semantic relations among the underlying concepts. Figure 10.2 lists the most important ones allowed by CD.[1] The first column contains the rules; the second contains examples of their use; and the third contains an English version of each example. The rules shown in the figure can be interpreted as follows:

- Rule 1 describes the relationship between an actor and the event he or she causes. This is a two-way dependency since neither actor nor event can be considered primary. The letter p above the dependency link indicates past tense.
- Rule 2 describes the relationship between a PP and a PA that is being asserted to describe it. Many state descriptions, such as height, are represented in CD as numeric scales.
- Rule 3 describes the relationship between two PPs, one of which belongs to the set defined by the other.
- Rule 4 describes the relationship between a PP and an attribute that has already been predicated of it. The direction of the arrow is toward the PP being described.
- Rule 5 describes the relationship between two PPs, one of which provides a particular kind of information about the other. The three most common types of information to be provided in this way are possession (shown as POSS-BY), location (shown as LOC), and physical containment (shown as CONT). The direction of the arrow is again toward the concept being described.
- Rule 6 describes the relationship between an ACT and the PP that is the object of that ACT. The direction of the arrow is toward the ACT since the context of the specific ACT determines the meaning of the object relation.
- Rule 7 describes the relationship between an ACT and the source and the recipient of the ACT.
- Rule 8 describes the relationship between an ACT and the instrument with which it is performed. The instrument must always be a full conceptualization (i.e., it must contain an ACT), not just a single physical object.

[1] The table shown in the figure is adapted from several tables in Schank [1973].

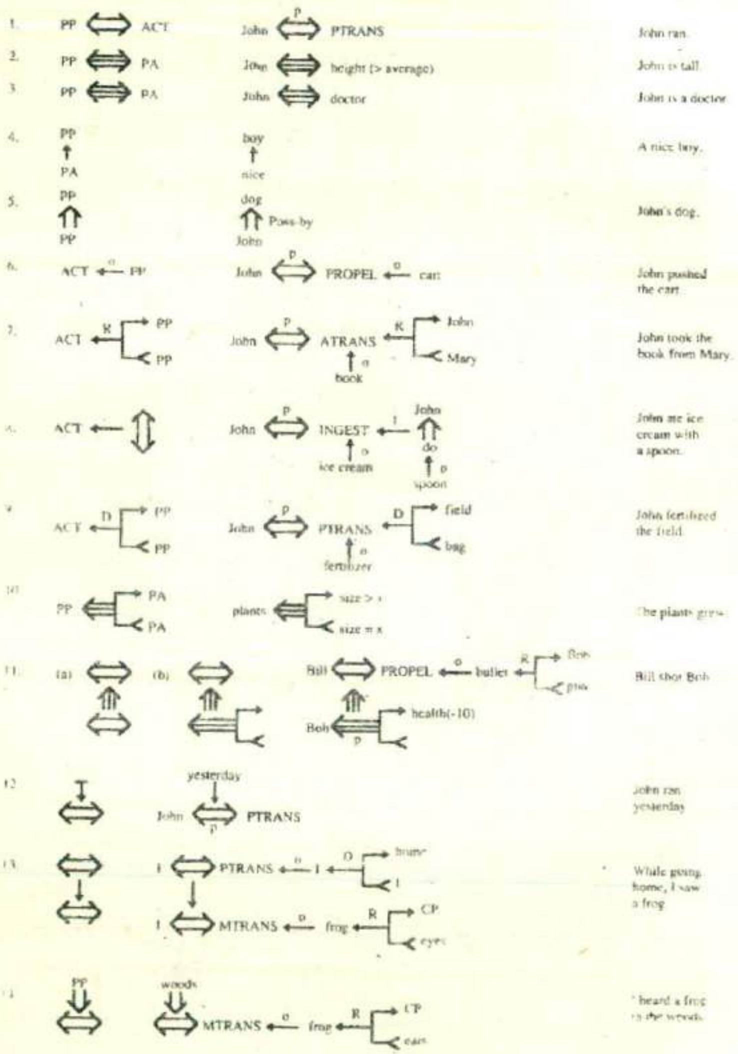

1. PP ⇔ ACT — John ⇔ PTRANS — John ran.
2. PP ⇔ PA — John ⇔ height (> average) — John is tall.
3. PP ⇔ PA — John ⇔ doctor — John is a doctor.
4. PP ↑ PA — boy ↑ nice — A nice boy.
5. PP ⇑ PP — dog ⇑ Poss-by John — John's dog.
6. ACT ← o PP — John ⇔ PROPEL ← o cart — John pushed the cart.
7. ACT ← R PP PP — John ⇔ ATRANS ← R John Mary; o book — John took the book from Mary.
8. ACT ← I — John ⇔ INGEST ← I John do; o ice cream; o spoon — John ate ice cream with a spoon.
9. ACT ← D PP PP — John ⇔ PTRANS ← D field bag; o fertilizer — John fertilized the field.
10. PP ⇔ PA PA — plants ⇔ size > x, size = x — The plants grew.
11. (a) (b) — Bill ⇔ PROPEL ← o bullet ← R Bob gun; Bob ⇔ health(-10) — Bill shot Bob.
12. T — yesterday John ⇔ PTRANS — John ran yesterday.
13. I ⇔ PTRANS ← o I ← D home I; I ⇔ MTRANS ← o frog ← R CP eyes — While going home, I saw a frog.
14. PP — woods ⇔ MTRANS ← o frog ← R CP ears — I heard a frog in the woods.

Figure 10.2: The Dependencies of CD

- Rule 9 describes the relationship between an ACT and its physical source and destination.
- Rule 10 represents the relationship between a PP and a state in which it started and another in which it ended.
- Rule 11 describes the relationship between one conceptualization and another that causes it. Notice that the arrows indicate dependency of one conceptualization on another and so point in the opposite direction of the implication arrows. The two forms of the rule describe the cause of an action and the cause of a state change.
- Rule 12 describes the relationship between a conceptualization and the time at which the event it describes occurred.
- Rule 13 describes the relationship between one conceptualization and another that is the time of the first. The example for this rule also shows how CD exploits a model of the human information processing system; *see* is represented as the transfer of information between the eyes and the conscious processor.
- Rule 14 describes the relationship between a conceptualization and the place at which it occurred.

Conceptualizations representing events can be modified in a variety of ways to supply information normally indicated in language by the tense, mood, or aspect of a verb form. The use of the modifier p to indicate past tense has already been shown. The set of conceptual tenses proposed by Schank [1973] includes

p	Past
f	Future
t	Transition
t_s	Start transition
t_f	Finished transition
k	Continuing
?	Interrogative
/	Negative
nil	Present
delta	Timeless
c	Conditional

As an example of the use of these tenses, consider the CD representation shown in Figure 10.3 (taken from Schank [1973]) of the sentence

Since smoking can kill you, I stopped.

The vertical causality link indicates that smoking kills one. Since it is marked c, however, we know only that smoking can kill one, not that it necessarily does. The horizontal causality link indicates that it is that first causality that made me stop smoking. The qualification t_{fp} attached to the dependency between I and INGEST indicates that the smoking (an instance of INGESTING) has stopped and that the stopping happened in the past.

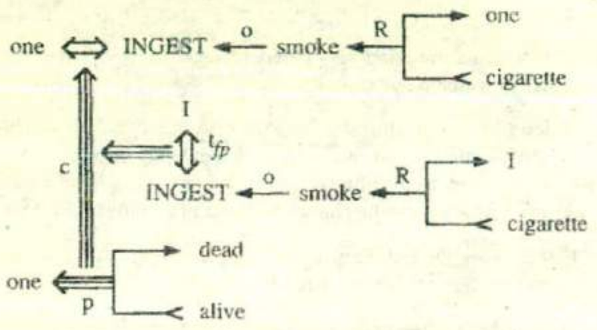


Figure 10.3: Using Conceptual Tenses

There are three important ways in which representing knowledge using the conceptual dependency model facilitates reasoning with the knowledge:

1. Fewer inference rules are needed than would be required if knowledge were not broken down into primitives.
2. Many inferences are already contained in the representation itself.
3. The initial structure that is built to represent the information contained in one sentence will have holes that need to be filled. These holes can serve as an attention focuser for the program that must understand ensuing sentences.

Each of these points merits further discussion.

The first argument in favor of representing knowledge in terms of CD primitives rather than in the higher-level terms in which it is normally described is that using the primitives makes it easier to describe the inference rules by which the knowledge can be manipulated. Rules need only be represented once for each primitive ACT rather than once for every word that describes that ACT. For example, all of the following verbs involve a transfer of ownership of an object:

- Give
- Take
- Steal
- Donate

If any of them occurs, then inferences about who now has the object and who once had the object (and thus who may know something about it) may be important. In a CD representation, those possible inferences can be stated once and associated with the primitive ACT ATRANS.

A second argument in favor of the use of CD representation is that to construct it, we must use not only the information that is stated explicitly in a sentence but also a set

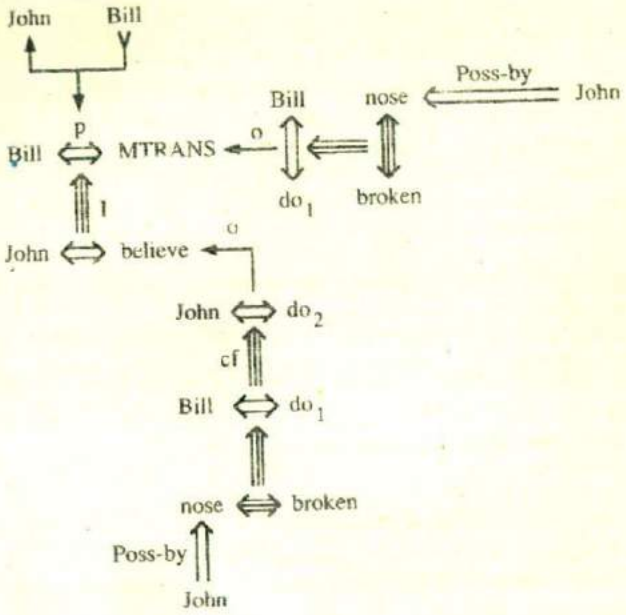


Figure 10.4: The CD Representation of a Threat

of inference rules associated with the specific information. Having applied these rules once, we store these results as part of the representation and they can be used repeatedly without the rules being reapplied. For example, consider the sentence

Bill threatened John with a broken nose.

The CD representation of the information contained in this sentence is shown in Figure 10.4. (For simplicity, *believe* is shown as a single unit. In fact, it must be represented in terms of primitive ACTs and a model of the human information processing system.) It says that Bill informed John that he (Bill) will do something to break John's nose. Bill did this so that John will believe that if he (John) does some other thing (different from what Bill will do to break his nose), then Bill will break John's nose. In this representation, the word "believe" has been used to simplify the example. But the idea behind *believe* can be represented in CD as an MTRANS of a fact into John's memory. The actions do_1 and do_2 are dummy placeholders that refer to some as yet unspecified actions.

A third argument for the use of the CD representation is that unspecified elements of the representation of one piece of information can be used as a focus for the understanding of later events as they are encountered. So, for example, after hearing that

Bill threatened John with a broken nose.

we might expect to find out what action Bill was trying to prevent John from performing. That action could then be substituted for the dummy action represented in Figure 10.4 as do_2. The presence of such dummy objects provides clues as to what other events or objects are important for the understanding of the known event.

Of course, there are also arguments against the use of CD as a representation formalism. For one thing, it requires that all knowledge be decomposed into fairly low-level primitives. In Section 4.3.3 we discussed how this may be inefficient or perhaps even impossible in some situations. As Schank and Owens [1987] put it,

CD is a theory of representing fairly simple actions. To express, for example, "John bet Sam fifty dollars that the Mets would win the World Series" takes about two pages of CD forms. This does not seem reasonable.

Thus, although there are several arguments in favor of the use of CD as a model for representing events, it is not always completely appropriate to do so, and it may be worthwhile to seek out higher-level primitives.

Another difficulty with the theory of conceptual dependency as a general model for the representation of knowledge is that it is only a theory of the representation of events. But to represent all the information that a complex program may need, it must be able to represent other things besides events. There have been attempts to define a set of primitives, similar to those of CD for actions, that can be used to describe other kinds of knowledge. For example, physical objects, which in CD are simply represented as atomic units, have been analyzed in Lehnert [1978]. A similar analysis of social actions is provided in Schank and Carbonell [1979]. These theories continue the style of representation pioneered by CD, but they have not yet been subjected to the same amount of empirical investigation (i.e., use in real programs) as CD.

We have discussed the theory of conceptual dependency in some detail in order to illustrate the behavior of a knowledge representation system built around a fairly small set of specific primitive elements. But CD is not the only such theory to have been developed and used in AI programs. For another example of a primitive-based system, see Wilks [1972].

10.2 Scripts

CD is a mechanism for representing and reasoning about events. But rarely do events occur in isolation. In this section, we present a mechanism for representing knowledge about common sequences of events.

A *script* is a structure that describes a stereotyped sequence of events in a particular context. A script consists of a set of slots. Associated with each slot may be some information about what kinds of values it may contain as well as a default value to be used if no other information is available. So far, this definition of a script looks very similar to that of a frame given in Section 9.2, and at this level of detail, the two structures are identical. But now, because of the specialized role to be played by a script, we can make some more precise statements about its structure.

Figure 10.5 shows part of a typical script, the restaurant script (taken from Schank and Abelson [1977]). It illustrates the important components of a script:

Entry conditions	Conditions that must, in general, be satisfied before the events described in the script can occur.
Result	Conditions that will, in general, be true after the events described in the script have occurred.
Props	Slots representing objects that are involved in the events described in the script. The presence of these objects can be inferred even if they are not mentioned explicitly.
Roles	Slots representing people who are involved in the events described in the script. The presence of these people, too, can be inferred even if they are not mentioned explicitly. If specific individuals are mentioned, they can be inserted into the appropriate slots.
Track	The specific variation on a more general pattern that is represented by this particular script. Different tracks of the same script will share many but not all components.
Scenes	The actual sequences of events that occur. The events are represented in conceptual dependency formalism.

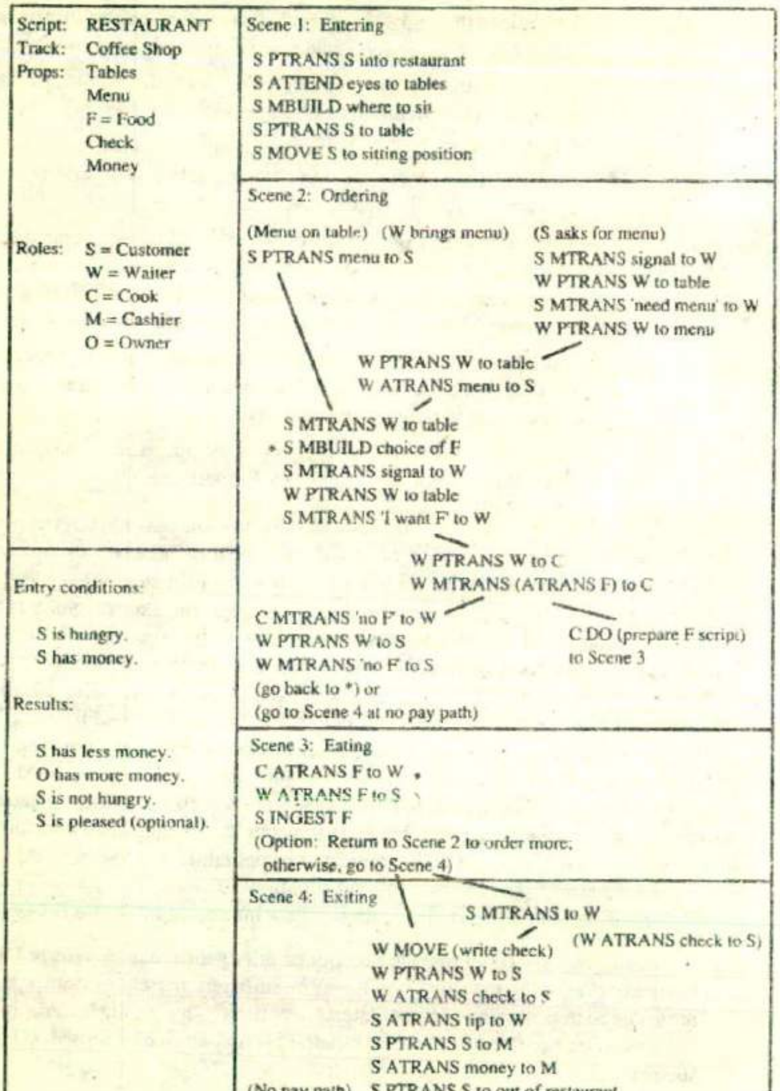
Scripts are useful because, in the real world, there are patterns to the occurrence of events. These patterns arise because of causal relationships between events. Agents will perform one action so that they will then be able to perform another. The events described in a script form a giant *causal chain*. The beginning of the chain is the set of entry conditions which enable the first events of the script to occur. The end of the chain is the set of results which may enable later events or event sequences (possibly described by other scripts) to occur. Within the chain, events are connected both to earlier events that make them possible and to later events that they enable.

If a particular script is known to be appropriate in a given situation, then it can be very useful in predicting the occurrence of events that were not explicitly mentioned. Scripts can also be useful by indicating how events that were mentioned relate to each other. For example, what is the connection between someone's ordering steak and someone's eating steak? But before a particular script can be applied, it must be activated (i.e., it must be selected as appropriate to the current situation). There are two ways in which it may be useful to activate a script, depending on how important the script is likely to be:

- For fleeting scripts (ones that are mentioned briefly and may be referred to again but are not central to the situation), it may be sufficient merely to store a pointer to the script so that it can be accessed later if necessary. This would be an appropriate strategy to take with respect to the restaurant script when confronted with a story such as

  Susan passed her favorite restaurant on her way to the museum. She really enjoyed the new Picasso exhibit.

- For nonfleeting scripts it is appropriate to activate the script fully and to attempt to fill in its slots with particular objects and people involved in the current situation.



Script: RESTAURANT
Track: Coffee Shop
Props: Tables
Menu
F = Food
Check
Money

Roles: S = Customer
W = Waiter
C = Cook
M = Cashier
O = Owner

Entry conditions:
S is hungry.
S has money.

Results:
S has less money.
O has more money.
S is not hungry.
S is pleased (optional).

Scene 1: Entering
S PTRANS S into restaurant
S ATTEND eyes to tables
S MBUILD where to sit
S PTRANS S to table
S MOVE S to sitting position

Scene 2: Ordering
(Menu on table) (W brings menu) (S asks for menu)
S PTRANS menu to S
S MTRANS signal to W
W PTRANS W to table
S MTRANS 'need menu' to W
W PTRANS W to menu
W PTRANS W to table
W ATRANS menu to S
S MTRANS W to table
* S MBUILD choice of F
S MTRANS signal to W
W PTRANS W to table
S MTRANS 'I want F' to W
W PTRANS W to C
W MTRANS (ATRANS F) to C
C MTRANS 'no F' to W
W PTRANS W to S
W MTRANS 'no F' to S
(go back to *) or
(go to Scene 4 at no pay path)
C DO (prepare F script)
to Scene 3

Scene 3: Eating
C ATRANS F to W
W ATRANS F to S
S INGEST F
(Option: Return to Scene 2 to order more;
otherwise, go to Scene 4)

Scene 4: Exiting
S MTRANS to W
(W ATRANS check to S)
W MOVE (write check)
W PTRANS W to S
W ATRANS check to S
S ATRANS tip to W
S PTRANS S to M
S ATRANS money to M
(No pay path) S PTRANS S to out of restaurant

Figure 10.5: The Restaurant Script

The headers of a script (its preconditions, its preferred locations, its props, its roles, and its events) can all serve as indicators that the script should be activated. In order to cut down on the number of times a spurious script is activated, it has proved useful to require that a situation contain at least two of a script's headers before the script will be activated.

Once a script has been activated, there are, as we have already suggested, a variety of ways in which it can be useful in interpreting a particular situation. The most important of these is the ability to predict events that have not explicitly been observed. Suppose, for example, that you are told the following story:

John went out to a restaurant last night. He ordered steak. When he paid for it, he noticed that he was running out of money. He hurried home since it had started to rain.

If you were then asked the question

Did John eat dinner last night?

you would almost certainly respond that he did, even though you were not told so explicitly. By using the restaurant script, a computer question-answerer would also be able to infer that John ate dinner, since the restaurant script could have been activated. Since all of the events in the story correspond to the sequence of events predicted by the script, the program could infer that the entire sequence predicted by the script occurred normally. Thus it could conclude, in particular, that John ate. In their ability to predict unobserved events, scripts are similar to frames and to other knowledge structures that represent stereotyped situations. Once one of these structures is activated in a particular situation, many predictions can be made.

A second important use of scripts is to provide a way of building a single coherent interpretation from a collection of observations. Recall that a script can be viewed as a giant causal chain. Thus it provides information about how events are related to each other. Consider, for example, the following story:

Susan went out to lunch. She sat down at a table and called the waitress. The waitress brought her a menu and she ordered a hamburger.

Now consider the question

Why did the waitress bring Susan a menu?

The script provides two possible answers to that question:

- Because Susan asked her to. (This answer is gotten by going backward in the causal chain to find out what caused her to do it.)
- So that Susan could decide what she wanted to eat. (This answer is gotten by going forward in the causal chain to find out what event her action enables.)

A third way in which a script is useful is that it focuses attention on unusual events. Consider the following story:

John went to a restaurant. He was shown to his table. He ordered a large steak. He sat there and waited for a long time. He got mad and left.

The important part of this story is the place in which it departs from the expected sequence of events in a restaurant. John did not get mad because he was shown to his table. He did get mad because he had to wait to be served. Once the typical sequence of events is interrupted, the script can no longer be used to predict other events. So, for example, in this story, we should not infer that John paid his bill. But we can infer that he saw a menu, since reading the menu would have occurred before the interruption. For a discussion of SAM, a program that uses scripts to perform this kind of reasoning, see Cullingford [1981].

From these examples, we can see how information about typical sequences of events, as represented in scripts, can be useful in interpreting a particular, observed sequence of events. The usefulness of a script in some of these examples, such as the one in which unobserved events were predicted, is similar to the usefulness of other knowledge structures, such as frames. In other examples, we have relied on specific properties of the information stored in a script, such as the causal chain represented by the events it contains. Thus although scripts are less general structures than are frames, and so are not suitable for representing all kinds of knowledge, they can be very effective for representing the specific kinds of knowledge for which they were designed.

10.3 CYC

CYC [Lenat and Guha, 1990] is a very large knowledge base project aimed at capturing human commonsense knowledge. Recall that in Section 5.1, our first attempt to prove that Marcus was not loyal to Caesar failed because we were missing the simple fact that all men are people. The goal of CYC is to encode the large body of knowledge that is so obvious that it is easy to forget to state it explicitly. Such a knowledge base could then be combined with specialized knowledge bases to produce systems that are less brittle than most of the ones available today.

Like CD, CYC represents a specific theory of how to describe the world, and like CD, it can be used for AI tasks such as natural language understanding. CYC, however, is more comprehensive; while CD provided a specific theory of representation for events, CYC contains representations of events, objects, attitudes, and so forth. In addition, CYC is particularly concerned with issues of scale, that is, what happens when we build knowledge bases that contain millions of objects.

10.3.1 Motivations

Why should we want to build large knowledge bases at all? There are many reasons, among them:

- Brittleness—Specialized knowledge-based systems are brittle. They cannot cope with novel situations, and their performance degradation is not graceful. Programs built on top of deep, commonsense knowledge about the world should rest on firmer foundations.

- Form and Content—The techniques we have seen so far for representing and using knowledge may or may not be sufficient for the purposes of AI. One good way to find out is to start coding large amounts of commonsense knowledge and see where the difficulties crop up. In other words, one strategy is to focus temporarily on the content of knowledge bases rather than on their form.

- Shared Knowledge—Small knowledge-based systems must make simplifying assumptions about how to represent things like space, time, motion, and structure. If these things can be represented once at a very high level, then domain-specific systems can gain leverage cheaply. Also, systems that share the same primitives can communicate easily with one another.

Building an immense knowledge base is a staggering task, however. We should ask whether there are any methods for acquiring this knowledge automatically. Here are two possibilities:

1. Machine Learning—In Chapter 17, we discuss some techniques for automated learning. However, current techniques permit only modest extensions of a program's knowledge. In order for a system to learn a great deal, it must already know a great deal. In particular, systems with a lot of knowledge will be able to employ powerful analogical reasoning.

2. Natural Language Understanding—Humans extend their own knowledge by reading books and talking with other humans. Since we now have on-line versions of encyclopedias and dictionaries, why not feed these texts into an AI program and have it assimilate all the information automatically? Although there are many techniques for building language understanding systems (see Chapter 15), these methods are themselves very knowledge-intensive. For example, when we hear the sentence

   John went to the bank and withdrew $50.

   we easily decide that "bank" means a financial institution, and not a river bank. To do this, we apply fairly deep knowledge about what a financial institution is, what it means to withdraw money, etc. Unfortunately, for a program to assimilate the knowledge contained in an encyclopedia, that program must already know quite a bit about the world.

The approach taken by CYC is to hand-code (what its designers consider to be) the ten million or so facts that make up commonsense knowledge. It may then be possible to bootstrap into more automatic methods.

10.3.2 CYCL

CYC's knowledge is encoded in a representation language called CYCL. CYCL is a frame-based system that incorporates most of the techniques described in Chapter 9 (multiple inheritance, slots as full-fledged objects, *transfers-through*, *mutually-disjoint-with*, etc). CYCL generalizes the notion of inheritance so that properties can be inherited along any link, not just *isa* and *instance*. Consider the two statements:

```
Mary
  likes:           ???
  constraints:     (LispConstraint)

LispConstraint
  slotConstrained:  (likes)
  slotValueSubsumes:
     (TheSetOf X (Person allInstances)
       (And (programsIn X LispLanguage)
            (Not (ThereExists Y (Languages allInstances)
                   (And (Not (Equal Y LispLanguage))
                        (programsIn X Y)))))))
   propagationDirection:   forward

Bob
   programsIn:     (LispLanguage)

Jane
   programsIn:     (LispLanguage CLanguage)
```

Figure 10.6: Frames and Constraint Expressions in CYC

1. All birds have two legs.
2. All of Mary's friends speak Spanish.

We can easily encode the first fact using standard inheritance—any frame with *Bird* on its *instance* slot inherits the value 2 on its *legs* slot. The second fact can be encoded in a similar fashion if we allow inheritance to proceed along the *friend* relation—any frame with *Mary* on its *friend* slot inherits the value *Spanish* on its *languagesSpoken* slot. CYC further generalizes inheritance to apply to a chain of relations, allowing us to express facts like, "All the parents of Mary's friends are rich," where the value *Rich* is inherited through a composition of the *friend* and *parentOf* links.

In addition to frames, CYCL contains a *constraint language* that allows the expression of arbitrary first-order logical expressions. For example, Figure 10.6 shows how we can express the fact "Mary likes people who program solely in Lisp." *Mary* has a constraint called *lispConstraint*, which restricts the values of her *likes* slot. The *slotValueSubsumes* attribute of *lispConstraint* ensures that Mary's *likes* slot will be filled with at least those individuals that satisfy the logical condition, namely that they program in *LispLanguage* and no others.

The time at which the default reasoning is actually performed is determined by the direction of the *slotValueSubsumes* rules. If the direction is *backward*, the rule is an if-needed rule, and it is invoked whenever someone inquires as to the value of Mary's *likes* slot. (In this case, the rule infers that Mary likes Bob but not Jane.) If the direction is *forward*, the rule is an if-added rule, and additions are automatically propagated to Mary's *likes* slot. For example, after we place LISP on Bob's *programsIn* slot, then the system quickly places Bob on Mary's *likes* slot for us. A truth maintenance system (see Chapter 7) ensures that if Bob ceases to be a Lisp programmer (or if he starts using Pascal), then he will also cease to appear on Mary's *likes* slot.

While forward rules can be very useful, they can also require substantial time and space to propagate their values. If a rule is entered as backward, then the system defers reasoning until the information is specifically requested. CYC maintains a separate background process for accomplishing forward propagations. A knowledge engineer can continue entering knowledge while its effects are propagated during idle keyboard time.[7]

Now let us return to the constraint language itself. Recall that it allows for the expression of facts as arbitrary logical expressions. Since first-order logic is much more powerful than CYC's frame language, why does CYC maintain both? The reason is that frame-based inference is very efficient, while general logical reasoning is computationally hard. CYC actually supports about twenty types of efficient inference mechanisms (including inheritance and transfers-through), each with its own truth maintenance facility. The constraint language allows for the expression of facts that are too complex for any of these mechanisms to handle.

The constraint language also provides an elegant, abstract layer of representation. In reality, CYC maintains two levels of representation: the *epistemological level* (EL) and the *heuristic level* (HL). The EL contains facts stated in the logical constraint language, while the HL contains the same facts stored using efficient inference templates. There is a translation program for automatically converting an EL statement into an efficient HL representation. The EL provides a clean, simple functional interface to CYC so that users and computer programs can easily insert and retrieve information from the knowledge base. The EL/HL distinction represents one way of combining the formal neatness of logic with the computational efficiency of frames.

In addition to frames, inference mechanisms, and the constraint language, CYCL performs consistency checking (e.g., detecting when an illegal value is placed on a slot) and conflict resolution (e.g., handling cases where multiple inference procedures assign incompatible values to a slot).

10.3.3 Control and Meta-Knowledge

Recall our discussion of control knowledge in Chapter 6, where we saw how to take information about control out of a production system interpreter and represent it declaratively using rules. CYCL strives to accomplish the same thing with frames. We have already seen how to specify whether a fact is propagated in the forward or backward direction—this is a type of control information. Associated with each slot is a set of inference mechanisms that can be used to compute values for it. For any given problem, CYC's reasoning is constrained to a small range of relevant, efficient procedures. A query in CYCL can be tagged with a level of effort. At the lowest level of effort, CYC merely checks whether the fact is stored in the knowledge base. At higher levels, CYC will invoke backward reasoning and even entertain metaphorical chains of inference. As the knowledge base grows, it will become necessary to use control knowledge to restrict reasoning to the most relevant portions of the knowledge base. This control knowledge can, of course, be stored in frames.

[7] Another idea is to have the system do forward propagation of knowledge during periods of infrequent use, such as at night.

In the tradition of its predecessor RLL (Representation Language Language) [Greiner and Lenat, 1980], many of the inference mechanisms used by CYC are stored explicitly as EL templates in the knowledge base. These templates can be modified like any other frames, and a user can create a new inference template by copying and editing an old one. CYC generates LISP code to handle the various aspects of an inference template. These aspects include recognizing when an EL statement can be transformed into an instance of the template, storing justifications of facts that are deduced (and retracting those facts when the justifications disappear), and applying the inference mechanism efficiently. As with production systems, we can build a more flexible, reflective system by moving inference procedures into a declarative representation.

It should be clear that many of the same control issues exist for frames and rules. Unlike numerical heuristic evaluation functions, control knowledge often has a commonsense, "knowledge about the world" flavor to it. It therefore begins to bridge the gap between two usually disparate types of knowledge: knowledge that is typically used for search control and knowledge that is typically used for natural language disambiguation.

10.3.4 Global Ontology

Ontology is the philosophical study of what exists. In the AI context, ontology is concerned with which categories we can usefully quantify over and how those categories relate to each other. All knowledge-based systems refer to entities in the world, but in order to capture the breadth of human knowledge, we need a well-designed *global ontology* that specifies at a very high level what kinds of things exist and what their general properties are. As mentioned above, such a global ontology should provide a more solid foundation for domain-specific AI programs and should also allow them to communicate with each other.

The highest level concept in CYC is called *Thing*. Everything is an instance of *Thing*. Below this top-level concept, CYC makes several distinctions, including:

- *IndividualObject* versus *Collection*—The CYCL concept *Collection* corresponds to the class CLASS described in Chapter 9. Here are some examples of frames that are instances of *Collection*: *Person, Nation, Nose*. Some instances of *IndividualObject* are *Fred, Greece, Fred'sNose*. These two sets share no common instances, and any instance of *Thing* must be an instance of one of the two sets. Anything that is an instance of *Collection* is a subset of *Thing*. Only *Collections* may have supersets and subsets; only *IndividualObjects* may have parts.

- *Intangible, Tangible*, and *Composite*—Instances of *Intangible* are things without mass, e.g., sets, numbers, laws, and events. Instances of *TangibleObject* are things with mass that have no intangible aspect, e.g., a person's body, an orange, and dirt. Every instance of *TangibleObject* is also an instance of *IndividualObject* since sets have no mass. Instances of *CompositeObject* have two key slots, *physicalExtent* and *intangibleExtent*. For example, a person is a *CompositeObject* whose *physicalExtent* is his body and whose *intangibleExtent* is his mind.

- *Substance*—*Substance* is a subclass of *IndividualObject*. Any subclass of *Substance* is something that retains its properties when it is cut up into smaller pieces.

For example, *Wood* is a *Substance*.[3] A concept like *Table34* can be an instance of both *Wood* (a *Substance*) and *Table* (an *IndividualObject*).

- *Intrinsic* versus *Extrinsic* properties—A property is intrinsic if when an object has that property all parts of the object also have that property. For example, *color* is an intrinsic property. Objects tend to inherit their intrinsic properties from *Substances*. Extrinsic properties include things like *number-of-legs*. Objects tend to inherit their extrinsic properties from *IndividualObjects*.

- *Event* and *Process*—An *Event* is anything with temporal extent, e.g., *Walking*. *Process* is a subclass of *Event*. If every temporal slice of an *Event* is essentially the same as the entire *Event*, then that *Event* is also a *Process*. For example, *Walking* is a *Process*, but *WalkingTwoMiles* is not. This relationship is analogous to *Substance* and *IndividualObject*.

- Slots—*Slot* is a subclass of *Intangible*. There are many types of *Slot*. *BookkeepingSlots* record such information as when a frame was created and by whom. *DefiningSlots* refer not to properties of the frame but to properties of the object represented by the frame. *DefiningSlots* are further divided into intensional, taxonomic, and extensional categories. *QuantitativeSlots* are those which take on a scalar range of values, e.g., *height*, as opposed to *gender*.

- Time—*Events* can have temporal properties, such as *duration* and *startsBefore*. CYC deals with two basic types of temporal measures: intervals, and sets of intervals. A number of basic interval properties, such as *endsDuring*, are defined from the property *before*, which applies to starting and ending times for events. Sets of intervals are built up from basic intervals through operations like union and intersection. Thus, it is possible to state facts like "John goes to the movies at three o'clock every Sunday."

- *Agent*—An important subset of *CompositeObject* is *Agent*, the collection of intelligent beings. *Agents* can be collective (e.g., corporations) or individual (e.g., people). *Agents* have a number of properties, one of which is *beliefs*. Agents often ascribe their own beliefs to other agents in order to facilitate communication. An agent's beliefs may be incorrect, so CYC must be able to distinguish between facts in its own knowledge base (CYC's beliefs) and "facts" that are possibly inconsistent with the knowledge base.

These are but a few of the ontological decisions that the builders of a large knowledge base must make. Other problems arise in the representation of space, causality, structures, and the persistence of objects through time. We return to some of these issues in Chapter 19.

10.3.5 Tools

CYC is a multi-user system that provides each knowledge enterer with a textual and graphical interface to the knowledge base. Users' modifications to the knowledge base

[3]Of course, if we cut a substance up *too* finely, it ceases to be the same substance. For each substance type, CYC stores its *granule size*, e.g., *Wood.granule = PlantCell*, *Crowd.granule = Person*, etc.

are transmitted to a central server, where they are checked and then propagated to other users.

We do not yet have much experience with the engineering problems of building and maintaining very large knowledge bases. In the future, it will be necessary to have tools that check consistency in the knowledge base, point out areas of incompleteness, and ensure that users do not step on each others' toes.

10.4 Exercises

1. Show a conceptual dependency representation of the sentence

   John begged Mary for a pencil.

   How does this representation make it possible to answer the question

   Did John talk to Mary?

2. One difficulty with representations that rely on a small set of semantic primitives, such as conceptual dependency, is that it is often difficult to represent distinctions between fine shades of meaning. Write CD representations for each of the following sentences. Try to capture the differences in meaning between the two sentences of each pair.

   John slapped Bill.
   John punched Bill.

   Bill drank his Coke.
   Bill slurped his Coke.

   Sue likes Dickens.
   Sue adores Dickens.

3. Construct a script for going to a movie from the viewpoint of the movie goer.

4. Consider the following paragraph:

   > Jane was extremely hungry. She thought about going to her favorite restaurant for dinner, but it was the day before payday. So instead she decided to go home and pop a frozen pizza in the oven. On the way, though, she ran into her friend, Judy. Judy invited Jane to go out to dinner with her and Jane instantly agreed. When they got to their favorite place, they found a good table and relaxed over their meal.

   How could the restaurant script be invoked by the contents of this story? Trace the process throughout the story. Might any other scripts also be invoked? For example, how would you answer the question, "Did Jane pay for her dinner?"

5. Would conceptual dependency be a good way to represent the contents of a typical issue of *National Geographic*?

6. State where in the CYC ontology following concepts should fall:

   - cat
   - court case
   - New York Times
   - France
   - glass of water

Chapter 11

Knowledge Representation Summary

In this chapter, we review the representational schemes that have been discussed so far and we mention briefly some additional representational techniques that are sometimes useful. You may find it useful at this point to reread Chapter 4 for a review of the knowledge representation issues that we outlined there.

11.1 Syntactic-Semantic Spectrum of Representation

One way to review the representational schemes we have just described is to consider an important dimension along which they can be characterized. At one extreme are purely *syntactic* systems, in which no concern is given to the meaning of the knowledge that is being represented. Such systems have simple, uniform rules for manipulating the representation. They do not care what information the representation contains. At the other extreme are purely *semantic* systems, in which there is no unified form. Every aspect of the representation corresponds to a different piece of information, and the inference rules are correspondingly complicated.

So far, we have discussed eight declarative structures in which knowledge can be represented:

- Predicate logic
- Production rules
- Nonmonotonic systems
- Statistical reasoning systems
- Semantic nets
- Frames
- Conceptual dependency

- Scripts
- CYC

Of these, the logical representations (predicate logic and the nonmonotonic systems) and the statistical ones are the most purely syntactic. Their rules of inference are strictly syntactic procedures that operate on well-formed formulas (wff) regardless of what those formulas represent. Production rule systems are primarily syntactic also. The interpreters for these systems usually use only syntactic information (such as the form of the pattern on the left side, the position of the rule in the knowledge base, or the position of the matched object in short-term memory) to decide which rules to fire. Again here we see the similarity between logic and production rules as ways of representing and using knowledge. But it is possible to build production-rule systems that have more semantics embedded in them. For example, in EMYCIN and other systems that provide explicit support for certainty factors, the semantics of certainty factors are used by the rule interpreter to guide its behavior.

Slot-and-filler structures are typically more semantically oriented, although they span a good distance in this spectrum. Semantic nets, as their name implies, are designed to capture semantic relationships among entities, and they are usually employed with a set of inference rules that have been specially designed to handle correctly the specific types of arcs present in the network. (For example, *isa* links are treated differently from most other kinds of links.) Frame systems are typically more highly structured than are semantic nets, and they contain an even larger set of specialized inference rules, including those that implement a whole array of default inheritance rules, as well as other procedures such as consistency checking.

Conceptual dependency moves even further toward being a semantic rather than a syntactic representation. It provides not only the abstract structure of a representation but also a specific indication of what components the representation should contain (such as the primitive ACTs and the dependency relationships). Thus, although CD representations can be thought of as instances of semantic nets, they can be used by more powerful inference mechanisms that exploit specific knowledge about what they contain. And although scripts appear very similar to frames, they are frames in which the slots have been carefully chosen to represent the information that is useful when reasoning about situations. This makes it possible for script manipulation procedures to exploit knowledge about what they are working with in order to solve problems more efficiently. CYC uses both frames and logic (depending on the level at which we view the knowledge) to encode specific types of knowledge and inference aimed at commonsense reasoning. CYC is the most semantic of the systems we have described, since it provides the most built-in knowledge of how to manipulate specific kinds of knowledge structures. It also contains a comprehensive ontology into which new knowledge can be put.

In general, syntactic representations are to knowledge representation what the weak methods of Chapter 3 are to problem-solving. They are, in principle, adequate for any problem. But for hard problems, their generality often means that answers cannot be found quickly. Stronger, more semantically oriented approaches make it possible to use knowledge more effectively to guide search. This does not mean that there is no place for weak or syntactic methods. Sometimes they are adequate, and their simplicity makes a formal analysis of programs that use them much more straightforward than a comparable analysis of a program based on semantic methods. But powerful programs depend on powerful knowledge, some of which is typically embedded in their problem-solving procedures and some of which is embedded in their knowledge representation mechanisms. In fact, as we have seen throughout Part II of this book, it is not usually possible to separate the two facets cleanly.

However, as we have seen in the last few chapters, knowledge representation systems can play the role of support systems that underly specific problem-solving programs. The knowledge representation system is typically expected not just to hold knowledge but also to provide a set of basic inference procedures, such as property inheritance or truth maintenance, that are defined on the knowledge. Specific problem-solving procedures can then be implemented as a level on top of that.

When knowledge representation systems are viewed as modules that are going to be incorporated as black boxes into larger programs, a good argument can be made [Brachman and Levesque, 1984] that their functionality should be restricted to purely syntactic operations about which very precise statements can be made. Essentially, this argument follows standard software engineering principles. To use a module effectively, one must have access to precise functional specifications of that module. If a knowledge representation system performs operations that are highly semantic in nature, it is difficult or impossible to write such a set of specifications. Among the kinds of operations that pose difficulties in this regard are the following:

- Operations whose result is defined to be the first or the best object satisfying some set of specifications. One example of such an operation is the resolution of a contradiction in a default-reasoning system. These operations require heuristics to define first or best and thus cannot usually be described in a straightforward way without appealing to the heuristics.

- Operations that are given resource limitations and whose output depends on how effectively those resources can be used. One common example of such an operation is default reasoning, when it is stated in a form such as, "Assume x unless $\neg x$ can be shown within z inference steps." The semantics of these operations then depend on how the resources happen to be exploited.

Of course, we are not saying that operations with these properties should not be done in reasoning programs. They are necessary. We are only saying that they should be within the control of some domain-specific problem solver rather than hidden within a general-purpose black box.

11.2 Logic and Slot-and-Filler Structures

Slot-and-filler structures have proven very valuable in the efficient storing and retrieving of knowledge for AI programs. They are usually poor, however, when it comes to representing rule-like assertions of the form "If x, y, and z, then conclude w." Predicate logic, on the other hand, does a reasonable job of representing such assertions, although general reasoning using these assertions is inefficient. Slot-and-filler representations are usually more semantic, meaning that their reasoning procedures are more varied, more efficient, and tied more closely to specific types of knowledge.

Hayes [1973] and Nilsson [1980] have shown how slot-and-filler structures can be translated into predicate logic. Concepts become one-place predicates, e.g., *dog*(x), and slots become two-place predicates, e.g., *color*(*canary*, *yellow*). Inference mechanisms like property inheritance can be expressed in logical notation, as a series of logical implications, which can then be manipulated with resolution. Working through a translation of a slot-and-filler structure to logic helps clear up what are often imprecisely specified reasoning methods in these structures. In practical terms, however, moving to logic means losing efficiency. For example, a typical slot-and-filler system has procedures for doing property inheritance that are much faster than doing property inheritance via resolution-based theorem proving. Part of the inefficiency of general reasoning methods like resolution can be overcome by intelligent indexing schemes, but the more heavily cross-indexed predicate logic clauses are, the more they come to resemble slot-and-filler structures.

On the other hand, it is difficult to express assertions more complex than inheritance in slot-and-filler structures. Is it possible to create a *hybrid* representational structure that combines the advantages of slot-and-filler structures with the advantages of predicate logic? We have already seen one system (CYC) that maintains both a logical (epistemological level) and frame-based (heuristic level) version of each fact. Another system, called KRYPTON [Brachman *et al.*, 1985], divides its knowledge into two distinct repositories, called the TBox and the ABox. The TBox is a slot-and-filler structure that contains *terminological* information. In it are concepts like "person," "car," and "person with three children." The ABox contains logical assertions, such as "Every person with three children owns a car." The atomic predicates used in ABox assertions refer to concepts defined in the TBox.

In logic-based systems, predicates such as *triangle* and *polygon* are primitive notions. These primitives are tied to one another via assertions, e.g., *isa*(*triangle*, *polygon*) and *isa*(*rectangle*, *polygon*). KRYPTON relates concepts like *triangle* and *polygon* terminologically, in the TBox, rather than assertionally. Thus we can do efficient terminological reasoning in the TBox and more general reasoning in the ABox. Terminological reasoning involves answering questions about subsumption and inheritance, such as "Can something be both a triangle and a rectangle?"

Consider a resolution theorem prover running with assertions in the ABox. A standard operation in resolution is determining when pairs of literals such as $f(x)$ and $\neg f(x)$ are inconsistent. Standard resolution requires that the literals be textually unifiable (except for the negation sign). KRYPTON extends the idea of textual inconsistency to *terminological* inconsistency in order to make the theorem prover more efficient. The TBox can tell that the two assertions *triangle*(x) and *rectangle*(x) are inconsistent and can thus be resolved against each other. The TBox can also determine the inconsistency of *triangle*(x) and $\neg$*polygon*(x); moreover, the two assertions $\neg$*rectangle*(x) and *polygon*(x) can be resolved against each other as long as we add to the resolvent the fact that x must have an angle which is not 90 degrees. If TBox computations are very efficient, then ABox proofs will be generated much faster than they would be in a pure logic framework.

11.3 Other Representational Techniques

In the last several chapters, we have described various techniques that can be used to represent knowledge. But our survey is by no means complete. There are other ways of representing knowledge; some of them are quite similar to the ones we have discussed and some are quite different. In this section we briefly discuss three additional methods: constraints, simulation models, and subsymbolic systems. Keep in mind throughout this discussion that it is not always the case that these various representational systems are mutually inconsistent. They often overlap, either in the way they use component representational mechanisms, the reasoning algorithms they support, or the problem-solving tasks for which they are appropriate.

11.3.1 Representing Knowledge as Constraints

Much of what we know about the world can be represented as sets of constraints. We talked in Section 3.5 about a very simple problem, cryptarithmetic, that can be described this way. But constraint-based representations are also useful in more complex problems. For example, we can describe an electronic circuit as a set of constraints that the states of various components of the circuit impose on the states of other components by virtue of being connected together. If the state of one of these components changes, we can propagate the effect of the change throughout the circuit by using the constraints. As a second example, consider the problem of interpreting visual scenes. We can write down a set of constraints that characterize the set of interpretations that can make sense in our physical world. For example, a single edge must be interpreted consistently, at both of its ends, as either a convex or a concave boundary. Finally, as we saw in Section 8.3, there are several kinds of relationships that can be represented as sets of constraints on the likelihoods that we can assign to collections of interdependent events.

In some sense, everything we write in any representational system is a constraint on the world models or problem solutions that we want our program to accept. For example, a wff [e.g., $\forall x : man(x) \rightarrow mortal(x)$] constrains the set of consistent models to those that do not include any man who is not mortal. But there is a very specific sense in which it is useful to talk about a specific class of techniques as constraint-based. Recall that in Section 3.5 we presented an algorithm for constraint satisfaction that was based on the notion of propagating constraints throughout a system until a final state was reached. This algorithm is particularly effective precisely when knowledge is represented in a way that makes it efficient to propagate constraints. This will be true whenever it is easy to locate the objects that a given object influences. This occurs when the objects in the system are represented as a network whose links correspond to constraints among the objects. We considered one example of this when we talked about Bayesian networks in Section 8.3. We consider other examples later in this book. For example, we return to the problem of simulating physical processes, such as electronic circuits, in Section 19.1. We present in Section 14.3 a constraint-propagation solution (known as the Waltz algorithm) to a simple vision problem. And in Section 15.5 we outline a view of natural language understanding as a constraint satisfaction task

11.3.2 Models and Model-Based Reasoning

For many kinds of problem-solving tasks, it is necessary to model the behavior of some object or system. To diagnose faults in physical devices, such as electronic circuits or electric motors, it is necessary to model the behavior of both the correctly functioning device and some number of ill-functioning variants of it. To evaluate potential designs of such devices requires the same capability. Of course, as soon as we begin to think about modeling such complex entities, it becomes clear that the best we will be able to do is create an approximate model. There are various techniques that we can use to do that.

When we think about constructing a model of some entity in the world, the issue of what we mean by a model soon arises. To what extent should the structure of the model mirror the structure of the object being modeled? Some representational techniques tend to support models whose structure is very different from the structure of the objects being modeled. For example, in predicate logic we write wff's such as $\forall x : raven(x) \rightarrow black(x)$. In the real world, though, this single fact has no single realization; it is distributed across all known ravens. At the other extreme are representations, such as causal networks, in which the physical structure of the world is closely modeled in the structure of the representation.

There are arguments in favor of both ends of this spectrum (and many points in the middle). For example, if the knowledge structure closely matches the problem structure, then the frame problem may be easier to solve. Suppose, for example, that we have a robot-planning program and we want to know if we move a table into another room, what other objects also change location. A model that closely matches the structure of the world (as shown in Figure 11.1(*a*)) will make answering this question easy, while alternative representations (such as the one shown in Figure 11.1(*b*)) will not. For more on this issue, see Johnson-Laird [1983]. There are, however, arguments for representations whose structures do not closely model the world. For example, such representations typically do a better job of capturing generalizations and thus of making predictions about some kinds of novel situations.

11.3.3 Subsymbolic Systems

So far, all of the representations that we have discussed are symbolic, in the sense we defined in Section 1.2. There are alternative representations, many of them based on a neural model patterned after the human brain. These systems are often called neural nets or connectionist systems. We discuss such systems in Chapter 18.

11.4 Summary of the Role of Knowledge

In the last several chapters we have focused on the kinds of knowledge that may be useful to programs and on ways of representing and using that knowledge within programs. To sum up, for now, our treatment of knowledge within AI programs, let us return to a brief discussion of the two roles that knowledge can play in those programs.

- It may define the search space and the criteria for determining a solution to a problem. We call this knowledge *essential knowledge*.

```
(Livingroom1:
    contains:
        (Table1:
            made-of: Wood
            has-on:  (Vase1:
                          made-of: Glass)
                     (Lamp1: ...))
        (Table2:
            has-on: (Vase2: ...)))
```

(*a*)

in(*Table1*, *Livingroom*)
made-of(*Table1*, *Wood*)
on(*Vase1*, *Table1*)
made-of(*Vase1*, *Glass*)
on(*Vase2*, *Table2*)
on(*Lamp1*, *Table1*)

(*b*)

Figure 11.1: Capturing Structure in Models

- It may improve the efficiency of a reasoning procedure by informing that procedure of the best places to look for a solution. We call that knowledge *heuristic knowledge*.

In formal tasks, such as theorem proving and game playing, there is only a small amount of essential knowledge and the need for a large amount of heuristic knowledge may be challenged by several brute force programs that perform quite successfully (e.g., the chess programs HITECH [Berliner and Ebeling, 1989] and DEEP THOUGHT [Anantharaman *et al.*, 1990]). The real knowledge challenge arises when we tackle naturally occurring problems, such as medical diagnosis, natural language processing, or engineering design. In those domains, substantial bodies of both essential and heuristic knowledge are absolutely necessary.

11.5 Exercises

1. Artificial intelligence systems employ a variety of formalisms for representing knowledge and reasoning with it. For each of the following sets of sentences, indicate the formalism that best facilitates the representation of the knowledge given in the statements in order to answer the question that is posed. Explain your choice briefly. Show how the statements would be encoded in the formalism you have selected. Then show how the question could be answered.

John likes fruit.
Kumquats are fruit.
People eat what they like.
Does John eat kumquats?

Assume that candy contains sugar unless you know
specifically that it is dietetic.
M&M's are candy.
Diabetics should not eat sugar.
Bill is a diabetic.
Should Bill eat M&M's?

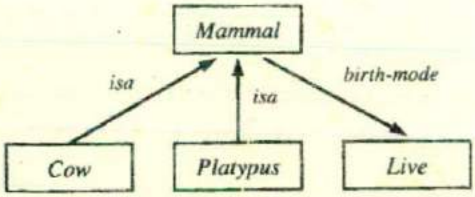
Most people like candy.
Most people who give parties like to serve food that
their guests like.
Tom is giving a party.
What might Tom like to serve?

When you go to a movie theatre, you usually buy a ticket,
hand the ticket to the ticket taker, and then go and
find a seat.
Sometimes you buy popcorn before going to your seat.
When the movie is over, you leave the theatre.
John went to the movies.
Did John buy a ticket?

2. Give five examples of facts that are difficult to represent and manipulate in predicate logic.

3. Suppose you had a predicate logic-based system in which you had represented the information in Figure 4.5. What additional knowledge would you have to include in order to cause properties to be inherited downward in the hierarchy? For example, how could you answer the question of how tall a pitcher is?

4. Property inheritance is a very common form of default reasoning. Consider the semantic net



(a) How could the information in this network be represented in a JTMS?

(b) What will happen when the additional fact that the platypus lays eggs is inserted into this system?