CHAPTER 3

Knowledge Representation

If, for a given problem, we have a means of checking a proposed solution, then we can solve the problem by testing all possible answers. But this always takes much too long to be of practical interest. Any device that can reduce this search may be of value.

—Marvin Minsky, *Steps Toward Artificial Intelligence*

Study is like the heaven's glorious sun,
That will not be deep-search'd with saucy looks;
Small have continual plodders ever won,
Save base authority from others' books.
These earthly godfathers of Heaven's lights
That give a name to every fixed star,
Have no more profit of their shining nights
Than those that walk and wot not what they are.

—William Shakespeare, *Love's Labours Lost*

Better the rudest work that tells a story or records a fact, than the richest without meaning.

—John Ruskin, *Seven Lamps of Architecture*

3.1 Introduction

Throughout this book we will be discussing representations. The reason for this is that in order for a computer to solve a problem that relates to the real world, it first needs some way to represent the real world internally. In dealing with that internal representation, the computer is then able to solve problems.

This chapter introduces a number of representations that are used elsewhere in this book, such as semantic nets, goal trees, and search trees, and explains why these representations provide such a powerful way to solve a wide range of problems.

This chapter also introduces frames and the way in which inheritance can be used to provide a powerful representational system.

This chapter is illustrated with a number of problems and suitable representations that can be used to solve those problems.

3.2 The Need for a Good Representation

As we will see elsewhere in this book, the representation that is used to represent a problem is very important. In other words, the way in which the computer represents a problem, the variables it uses, and the operators it applies to those variables can make the difference between an efficient algorithm and an algorithm that doesn't work at all. This is true of all Artificial Intelligence problems, and as we see in the following chapters, it is vital for search.

Imagine that you are looking for a contact lens that you dropped on a football field. You will probably use some knowledge about where you were on the field to help you look for it. If you spent time in only half of the field, you do not need to waste time looking in the other half.

Now let us suppose that you are having a computer search the field for the contact lens, and let us further suppose that the computer has access to an omniscient oracle that will answer questions about the field and can accurately identify whether the contact lens is in a particular spot.

Now we must choose a representation for the computer to use so that it can formulate the correct questions to ask.

One representation might be to have the computer divide the field into four equal squares and ask the oracle for each square, "Is the lens in this square?" This will identify the location on the field of the lens but will not really be very helpful to you because you will still have a large area to search once you find which quarter of the field the lens is in.

Another representation might be for the computer to have a grid containing a representation of every atom contained in the field. For each

atom, the computer could ask its oracle, "Is the lens in contact with this atom?"

This would give a very accurate answer indeed, but would be an extremely inefficient way of finding the lens. Even an extremely powerful computer would take a very long time indeed to locate the lens.

Perhaps a better representation would be to divide the field up into a grid where each square is one foot by one foot and to eliminate all the squares from the grid that you know are nowhere near where you were when you lost the lens. This representation would be much more helpful.

In fact, the representations we have described for the contact lens problem are all really the same representation, but at different levels of granularity. The more difficult problem is to determine the data structure that will be used to represent the problem we are exploring. As we will see throughout this book, there are a wide range of representations used in Artificial Intelligence.

When applying Artificial Intelligence to search problems, a useful, efficient, and meaningful representation is essential. In other words, the representation should be such that the computer does not waste too much time on pointless computations, it should be such that the representation really does relate to the problem that is being solved, and it should provide a means by which the computer can actually solve the problem.

In this chapter, we look at a number of representations that are used in search, and in particular we will look at search trees, which are used throughout this part of the book.

3.3 Semantic Nets

The semantic net is a commonly used representation in Artificial Intelligence. A semantic net is a **graph** consisting of **nodes** that are connected by **edges**. The nodes represent objects, and the links between nodes represent relationships between those objects. The links are usually labeled to indicate the nature of the relationship.

A simple example of a semantic net is shown in Figure 3.1.

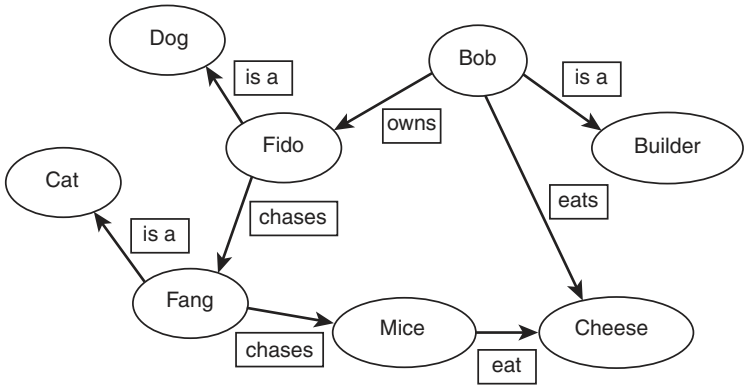

Figure 3.1
A simple semantic net

Note that in this semantic net, the links are arrows, meaning that they have a direction. In this way, we can tell from the diagram that Fido chases Fang, not that Fang chases Fido. It may be that Fang does chase Fido as well, but this information is not presented in this diagram.

Semantic nets provide a very intuitive way to represent knowledge about objects and the relationships that exist between those objects. The data in semantic nets can be reasoned about in order to produce systems that have knowledge about a particular domain. Semantic nets do have limitations, such as the inability to represent negations: "Fido is not a cat." As we see in Chapter 7, this kind of fact can be expressed easily in first-order predicate logic and can also be managed by rule-based systems.

Note that in our semantic net we have represented some specific individuals, such as Fang, Bob, and Fido, and have also represented some general classes of things, such as cats and dogs. The specific objects are generally referred to as **instances** of a particular **class**. Fido is an instance of the class dog. Bob is an instance of the class Builder.

It is a little unclear from Figure 3.1 whether cheese is a class or an instance of a class. This information would need to be derived by the system that is manipulating the semantic net in some way. For example, the system might have a rule that says "any object that does not have an 'is-a' relationship to a class is considered to represent a class of objects." Rules such as this must be applied with caution and must be remembered when building a semantic net.

An important feature of semantic nets is that they convey meaning. That is to say, the relationship between nodes and edges in the net conveys information about some real-world situation. A good example of a semantic net is a family tree diagram. Usually, nodes in these diagrams represent people, and there are edges that represent parental relationships, as well as relationships by marriage.

Each node in a semantic net has a label that identifies what the node represents. Edges are also labeled. Edges represent connections or relationships between nodes. In the case of searching a dictionary for a page that contains a particular word, each node might represent a single page, and each edge would represent a way of getting from one page to another.

The particular choice of semantic net representation for a problem will have great bearing on how the problem is solved. A simple representation for searching for a word in a dictionary would be to have the nodes arranged in a chain with one connection from the first node to the second, and then from the second to the third, and so on. Clearly, any method that attempts to search this graph will be fairly inefficient because it means visiting each node in turn until the desired node is found. This is equivalent to flicking through the pages of the dictionary in order until the desired page is found.

As we see in Section 3.7, representing the dictionary by a different data structure can give much more efficient ways of searching.

3.4 Inheritance

Inheritance is a relationship that can be particularly useful in AI and in programming. The idea of inheritance is one that is easily understood intuitively. For example, if we say that all mammals give birth to live babies, and we also say that all dogs are mammals, and that Fido is a dog, then we can conclude that Fido gives birth to live mammals. Of course, this particular piece of reasoning does not take into account the fact that Fido might be male, or if Fido is female, might be too young or too old to give birth.

So, inheritance allows us to specify properties of a **superclass** and then to define a **subclass**, which inherits the properties of the superclass. In our

example, mammals are the superclass of dogs and Fido. Dogs are the subclass of mammals and the superclass of Fido.

If you have programmed with an object-oriented programming language such as C++ or Java, then you will be familiar with the concept of inheritance and will appreciate its power. Object-oriented programming is discussed further in Section 3.6.

As has been shown, although inheritance is a useful way to express generalities about a class of objects, in some cases we need to express exceptions to those generalities (such as, "Male animals do not give birth" or "Female dogs below the age of six months do not give birth"). In such cases, we say that the **default value** has been **overridden** in the subclass.

As we will see, it is usually useful to be able to express in our chosen representation which values can be overridden and which cannot.

3.5 Frames

Frame-based representation is a development of semantic nets and allows us to express the idea of inheritance.

As with semantic nets, a **frame system** consists of a set of frames (or nodes), which are connected together by relations. Each **frame** describes either an instance (an **instance frame**) or a class (a **class frame**).

Thus far, we have said that instances are "objects" without really saying what an object is. In this context, an object can be a physical object, but it does not have to be. An object can be a property (such as a color or a shape), or it can be a place, or a situation, or a feeling. This idea of objects is the same that is used in object-oriented programming languages, such as C++ and Java. Frames are thus an object-oriented representation that can be used to build expert systems. Object-oriented programming is further discussed in Section 3.6.

Each frame has one or more **slots**, which are assigned **slot values**. This is the way in which the frame system network is built up. Rather than simply having links between frames, each relationship is expressed by a value being placed in a slot. For example, the semantic net in Figure 3.1 might be represented by the following frames:

Frame Name	Slot	Slot Value
Bob	is a	Builder
	owns	Fido
	eats	Cheese
Fido	is a	Dog
	chases	Fang
Fang	is a	Cat
	chases	Mice
Mice	eat	Cheese
Cheese		
Builder		
Dog		
Cat		

We can also represent this frame system in a diagrammatic form using representations such as those shown in Figure 3.2.

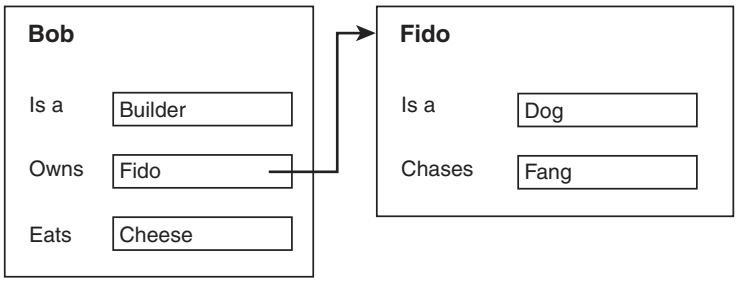

Figure 3.2
Partial representation for a frame system for the semantic net shown in Figure 3.1

When we say, "Fido is a dog," we really mean, "Fido is an instance of the class dog," or "Fido is a member of the class of dogs." Hence, the "is-a" relationship is very important in frame-based representations because it enables us to express membership of classes. This relationship is also known as **generalization** because referring to the class of mammals is more general than referring to the class of dogs, and referring to the class of dogs is more general than referring to Fido.

It is also useful to be able to talk about one object being a part of another object. For example, Fido has a tail, and so the tail is part of Fido. This relationship is known as **aggregation** because Fido can be considered an aggregate of dog parts.

Other relationships are known as **association**. An example of such a relationship is the "chases" relationship. This explains how Fido and Fang are related or associated with each other. Note that association relationships have meaning in two directions. The fact that Fido chases Fang means that Fang is chased by Fido, so we are really expressing two relationships in one association.

3.5.1 Why Are Frames Useful?

Frames can be used as a data structure by Expert Systems, which are discussed in more detail in Chapter 9.

The main advantage of using frame-based systems for expert systems over the rule-based approach is that all the information about a particular object is stored in one place. In a rule-based system, information about Fido might be stored in a number of otherwise unrelated rules, and so if Fido changes, or a deduction needs to be made about Fido, time may be wasted examining irrelevant rules and facts in the system, whereas with the frame system, the Fido frame could be quickly examined.

This difference becomes particularly clear when we consider frames that have a very large number of slots and where a large number of relationships exist between frames (i.e., a situation in which objects have a lot of properties, and a lot of objects are related to each other). Clearly, many real-world situations have these properties.

3.5.2 Inheritance

We might extend our frame system with the following additional information:

Dogs chase cats

Cats chase mice

In expressing these pieces of information, we now do not need to state explicitly that Fido chases Fang or that Fang chases mice. In this case, we can inherit this information because Fang is an instance of the class Cats, and Fido is an instance of the class Dogs.

We might also add the following additional information:

Mammals breathe

Dogs are mammals

Cats are mammals

Hence, we have now created a new superclass, mammals, of which dogs and cats are subclasses. In this way, we do not need to express explicitly that cats and dogs breathe because we can inherit this information. Similarly, we do not need to express explicitly that Fido and Fang breathe—they are instances of the classes Dogs and Cats, and therefore they inherit from those classes' superclasses.

Now let us add the following fact:

Mammals have four legs

Of course, this is not true, because humans do not have four legs, for example. In a frame-based system, we can express that this fact is the **default value** and that it may be overridden. Let us imagine that in fact Fido has had an unfortunate accident and now has only three legs. This information might be expressed as follows:

Frame Name	Slot	Slot Value
Mammal	*number of legs	four
Dog	subclass	Mammal
Cat	subclass	Mammal
Fido	is a number of legs	Dog three
Fang	is a	Cat

Here we have used an asterisk (*) to indicate that the value for the "number of legs" slot for the Mammal class is a default value and can be overridden, as has been done for Fido.

3.5.3 Slots as Frames

It is also possible to express a range of values that a slot can take—for example, the number of legs slot might be allowed a number between 1 and 4 (although, for the insects class, it might be allowed 6).

One way to express this kind of restriction is by allowing slots to be frames. In other words, the number of legs slot can be represented as a frame, which includes information about what range of values it can take:

Frame Name	Slot	Slot Value
Number of legs	minimum value	1
	maximum value	4

In this way, we can also express more complex ideas about slots, such as the **inverse** of a slot (e.g., the "chases" slot has an inverse, which is the "chased by" slot). We can also place further limitations on a slot, such as to specify whether or not it can take multiple values (e.g., the "number of legs" slot should probably only take one value, whereas the "eats" slot should be allowed to take many values).

3.5.4 Multiple Inheritance

It is possible for a frame to inherit properties from more than one other frame. In other words, a class can be a subclass of two superclasses, and an object can be an instance of more than one class. This is known as **multiple inheritance**.

For example, we might add the following frames to our system:

Frame Name	Slot	Slot Value
Human	Subclass	Mammal
	Number of legs	two
Builder	Builds	houses
Bob	is a	Human

From this, we can see that Bob is a human, as well as being a builder. Hence, we can inherit the following information about Bob:

He has two legs

He builds houses

In some cases, we will encounter **conflicts**, where multiple inheritance leads us to conclude contradictory information about a frame. For example, let us consider the following simple frame system:

Frame Name	Slot	Slot Value
Cheese	is	smelly
Thing wrapped in foil	is	not smelly
Cheddar	is a	Cheese
	is a	Thing wrapped in foil

(Note: the slot "is" might be more accurately named "has property." We have named it "is" to make the example clearer.)

Here we can see that cheddar is a type of cheese and that it comes wrapped in foil. Cheddar should inherit its smelliness from the Cheese class, but it also inherits nonsmelliness from the Thing wrapped in foil class. In this case, we need a mechanism to decide which features to inherit from which superclasses. One simple method is to simply say that conflicts are resolved by the order in which they appear. So if a fact is established by inheritance, and then that fact is contradicted by inheritance, the first fact is kept because it appeared first, and the contradiction is discarded.

This is clearly rather arbitrary, and it would almost certainly be better to build the frame system such that conflicts of this kind cannot occur.

Multiple inheritance is a key feature of most object-oriented programming languages. This is discussed in more detail in Section 3.6.

3.5.5 Procedures

In object-oriented programming languages such as C++ or Java, classes (and hence objects) have **methods** associated with them. This is also true with frames. Frames have methods associated with them, which are called **procedures**. Procedures associated with frames are also called **procedural attachments**.

A procedure is a set of instructions associated with a frame that can be executed on request. For example, a **slot reader** procedure might return the value of a particular slot within the frame. Another procedure might insert

a value into a slot (a **slot writer**). Another important procedure is the **instance constructor**, which creates an instance of a class.

Such procedures are called when needed and so are called **WHEN-NEEDED procedures**. Other procedures can be set up that are called automatically when something changes.

3.5.6 Demons

A **demon** is a particular type of procedure that is run automatically whenever a particular value changes or when a particular event occurs.

Some demons act when a particular value is read. In other words, they are called automatically when the user of the system, or the system itself, wants to know what value is placed in a particular slot. Such demons are called **WHEN-READ procedures**. In this way, complex calculations can be made that calculate a value to return to the user, rather than simply giving back static data that are contained within the slot. This could be useful, for example, in a large financial system with a large number of slots because it would mean that the system would not necessarily need to calculate every value for every slot. It would need to calculate some values only when they were requested.

WHEN-CHANGED procedures (also known as **WHEN-WRITTEN procedures**) are run automatically when the value of a slot is changed. This type of function can be particularly useful, for example, for ensuring that the values assigned to a slot fit within a set of constraints. For example, in our example above, a WHEN-WRITTEN procedure might run to ensure that the "number of legs" slot never has a value greater than 4 or less than 1. If a value of 7 is entered, a system message might be produced, telling the user that he or she has entered an incorrect value and that he or she should enter a different value.

3.5.7 Implementation

With the addition of procedures and demons, a frame system becomes a very powerful tool for reasoning about objects and relationships. The system has **procedural semantics** as opposed to **declarative semantics**, which

means that the order in which things occur affects the results that the system produces. In some cases, this can cause problems and can make it harder to understand how the system will behave in a given situation.

This lack of clarity is usually compensated for by the level of flexibility allowed by demons and the other features that frame systems possess.

Frame systems can be implemented by a very simple algorithm if we do not allow multiple inheritance. The following algorithm allows us to find the value of a slot S, for a frame F. In this algorithm definition, we will use the notation **F[S]** to indicate the value of slot S in frame F. We also use the notation **instance (F1, F2)** to indicate that frame F1 is an instance of frame F2 and **subclass (F1, F2)** to indicate that frame F1 is a subclass of frame F2.

```
Function find_slot_value (S, F)
{
     if F[S] == V             // if the slot contains
          then return V       // a value, return it.
     else if instance (F, F')
          then return find_slot_value (S, F')
     else if subclass (F, F_S)
          then return find_slot_value (S, F_S)
     else return FAILURE;
}
```

In other words, the slot value of a frame F will either be contained within that frame, or a superclass of F, or another frame of which F is an instance. If none of these provides a value, then the algorithm fails.

Clearly, frames could also be represented in an object-oriented programming language such as C++ or Java.

A frame-based expert system can be implemented in a similar way to the rule-based systems, which we examine in Chapter 9. To answer questions about an object, the system can simply examine that object's slots or the slots of classes of which the object is an instance or a subclass.

If the system needs additional information to proceed, it can ask the user questions in order to fill in additional information. In the same way as with rule-based systems, WHEN-CHANGED procedures can be set up that

monitor the values of slots, and when a particular set of values is identified, this can be used by the system to derive a conclusion and thus recommend an action or deliver an explanation for something.

3.5.8 Combining Frames with Rules

It is possible to combine frames with rules, and, in fact, many frame-based expert systems use rules in much the same way that rule-based systems do, with the addition of **pattern matching** clauses, which are used to identify values that match a set of conditions from all the frames in the system.

Typically, a frame-based system with rules will use rules to try to derive conclusions, and in some cases where it cannot find a value for a particular slot, a WHEN-NEEDED procedure will run to determine the value for that slot. If no value is found from that procedure, then the user will be asked to supply a value.

3.5.9 Representational Adequacy

We can represent the kinds of relationships that we can describe with frames in first-order predicate logic. For example:

$\forall x\ \mathtt{Dog}(x) \rightarrow \mathtt{Mammal}(x)$

First-order predicate logic is discussed in detail in Chapter 7. For now, you simply need to know how to read that expression. It is read as follows:

> "For all x's, if x is a dog, then x is a mammal."

This can be rendered in more natural English as:

> "All dogs are mammals."

In fact, we can also express this relationship by the introduction of a new symbol, which more closely mirrors the meaning encompassed by the idea of inheritance:

$$\text{Dog} \xrightarrow{\text{subset}} \text{Mammal}$$

Almost anything that can be expressed using frames can be expressed using first-order predicate logic (FPOL). The same is not true in reverse. For example, it is not easy to represent negativity ("Fido is not a cat") or quantification ("there is a cat that has only one leg"). We say that FOPL has greater **representational adequacy** than frame-based representations.

In fact, frame-based representations do have some aspects that cannot be easily represented in FOPL. The most significant of these is the idea of exceptions, or overriding default values.

Allowing exceptions to override default values for slots means that the frame-based system is not monotonic (monotonicity is discussed in Chapter 7). In other words, conclusions can be changed by adding new facts to the system.

In this section, we have discussed three main representational methods: logic, rules, and frames (or semantic nets). Each of these has advantages and disadvantages, and each is preferable over the others in different situations. The important thing is that in solving a particular problem, the correct representation must be chosen.

3.6 Object-Oriented Programming

We now briefly explore some of the ideas used in object-oriented programming, and, in particular, we see how they relate to some of the ideas we have seen in Sections 3.4 and 3.5 on inheritance and frames.

Two of the best-known object-oriented programming languages are Java and C++. These two languages use a similar syntax to define classes and objects that are instantiations of those classes.

A typical class in these languages might be defined as:

```
class animal
{
      animal ();
      Eye *eyes;
      Leg *legs;
      Head head;
      Tail tail;
}
```

This defines a class called animal that has a number of fields, which are the various body parts. It also has a **constructor**, which is a function that is called when an instantiation of the class is called. Classes can have other functions too, and these functions are equivalent to the procedures we saw in Section 3.5.5.

We can create an instance of the class animal as follows:

```
animal an_animal = new animal ();
```