The Elements of Artificial Intelligence

with Python

Steven L. Tanimoto
University of Washington

Draft version based upon The Elements of Artificial Intelligence Using Common Lisp

Part II: Chapter 4

January 23, 2005

Chapter 4

Knowledge Representation

4.1 Characteristics of Knowledge

One usually makes a distinction between data and information. Data consists of raw figures, measurements, and files that do not necessarily answer the questions that its users may have. Information, on the other hand, is somewhat more refined. It is often the result of processing crude data, giving useful statistics for the data, or answering specific questions posed by users. In AI we usually distinguish a third kind of order in memory: knowledge. We think of knowledge as a refined kind of information, often more general than that found in conventional databases. But it may be incomplete or fuzzy as well. We may think of knowledge as a collection of related facts, procedures, models, and heuristics that can be used in problem solving or inference systems. Knowledge may be regarded as information in context, as information organized so that it can be readily applied to solving problems, perception, and learning.

Knowledge varies widely in both its content and appearance. It may be specific, general, exact, fuzzy, procedural, declarative, etc. There are several commonly used methods to organize and represent knowledge. These are described in the following section.

4.2 Overview of Representation Methods

Before describing a few methods in some detail, it is useful to consider briefly a number of the general approaches that have been used for representing knowledge. These include production rules, inclusion hierarchies, mathematical logics, frames, scripts, semantic networks, constraints, and relational databases. Production rules, illustrated in Chapter 3, are a general method that is particularly appropriate when knowledge is action-oriented. Inclusion hierarchies, described later in this chapter in some detail, handle a particular kind of knowledge very

well: knowledge about objects that can be grouped into classifications such that some categories are subcategories of others. Inclusion hierarchies may be used as an organizing scheme in connection with other methods, such as the predicate calculus.

Mathematical logics such as the predicate calculus provide a general and fundamental capability that supports general logical inference. However, these logics seldom provide organizational support for *grouping* facts so that the facts can be efficiently used. In this sense, mathematical logics are low-level representation schemes that do well with details but require additional support to be useful in building nontrivial systems.

Frames provide an organizational scheme for knowledge bases, but not much more than this; the detailed representations require other methods. Scripts have been used in some experimental systems for natural language understanding to represent scenarios with standard chronology such as what a person does when he or she goes to a restaurant: gets a table, waits for the menu, orders, eats, pays the bill, and leaves; scripts are like frames with additional support for describing chronology.

Semantic networks, like frames, are a general organizational framework, but there is not necessarily any particular kind of low-level support in a semantic net system; any system in which the modules of knowledge may be described as nodes in a labeled graph may be called a "semantic net," although it tends to be systems that attempt to mimic the neuronal interconnection structure of the biological brain that are most often labeled by their creators as "semantic networks."

A kind of knowledge that is often described as a representation method is "constraints." A *constraint* is a relationship among one, two, or more objects that may be viewed as a predicate; the constraint is to be satisfied by the system in finding a solution to a problem. By emphasizing the use of constraints in representing a set of objects and their interrelations, a constraint-based approach to knowledge representation may be used.

Finally, relational databases can sometimes serve as a method for knowledge representation; as they are usually implemented, they are good at manipulating large quantities of regularly structured information in certain, largely preconceived, ways. Relational databases have not been ideal for AI applications in the past because of their inefficiency in making large numbers of small inferences involving either very small relations or small parts of larger relations; there is currently research going on to make relational databases more suitable for AI applications.

This chapter in an introduction to the most important methods for knowledge representation. Inclusion hierarchies, the predicate calculus, and frames are the methods treated here. After a brief discussion of production rules as a means of representing knowledge, we focus on the problem of representing a single concrete but very powerful relation: the inclusion relation between classes of objects. Next, the use of propositional and predicate logics for representing

knowledge is taken up. After examining semantic networks, we look at frames, schemata, and scripts. Then relational databases are considered. Finally, several issues related to knowledge representation are discussed and a comparative summary of the methods is given. This chapter must be considered an introduction and not a full survey of knowledge representation. The logical reasoning and probabilistic inference methods described in later chapters are themselves closely associated with knowledge representation. Some of the other methods and several specialized techniques for such problem domains as computer vision and natural language understanding are presented in later chapters.

4.3 Knowledge in Production Rules

If we examine the Shrink program to find the basis for its response-making ability, we would be hard-pressed to find anything but its production rules embedded into the `respond` function. The knowledge of what to say when is almost all embedded in these rules. Some of the Shrink's knowledge, however, lies outside the rules, although it is brought into play by the production rules. For example, the definition of the function `verbp` is knowledge (albeit at a primitive level) about the English language and is represented separately from the production rules, but at least one of the rules contains a condition that uses `verbp`.

Similarly, the knowledge about differentiating formulas in Leibniz lies almost entirely in its production rules. Some of Leibniz' ability comes from the control scheme's method of trying to apply productions at many levels of the current formula, but if we wanted to increase the set of problems that Leibniz could solve, we could simply add new rules.

In many of the expert systems described in the literature, such as MYCIN, AM, and PROSPECTOR, much of the knowledge is represented within production rules. The left-hand side of a production rule (the condition part) expresses the characteristics of a situation in which it is appropriate to perform the action (the right-hand side) of the production rule. The testing of the condition as well as the execution of the action may involve the manipulation of other data structures (knowledge bases). Thus the production rules might not embody all the knowledge in a system. Even when one plans to embed most of a system's knowledge in production rules, one should understand the use of other means of knowledge representation.

4.4 Concept Hierarchies

Much of our knowledge about the world is organized hierarchically. All the "things" we know of we group into classes or sets. These classes are grouped into superclasses and the superclasses into even bigger ones. With most of these classes we associate names which we use to identify the classes. There is a

class we call "dogs" and another we call "cats." These are grouped, with some other classes, into a superclass called "mammals." Plants, minerals, machines, emotions, information, and ideas are treated similarly. Much of our knowledge consists of an understanding of the inclusion relationship on all these classes and cognizance of various properties shared by all members of particular classes. "All horses have four legs" states that the property "has four legs" is shared by each member of the class of horses.

4.4.1 Inclusion, Membership, and ISA

The inclusion relation on a set of classes is very important in AI, and there are some interesting questions that arise when incorporating it into an AI system. Our first issue is deciding what statements of English express inclusion relationships. The sentence "A bear is a mammal" expresses that the class of bears is a subclass of the class of mammals. For this reason, the data structures used to represent inclusion relations are often called "ISA" hierarchies. The list (BEAR ISA MAMMAL) is one way of representing this inclusion relationship. One must beware of certain relationships that are similar in appearance to inclusion but are really quite different. "Teddy is a bear" does not really say that the class of Teddies is a subclass of the class of bears. Rather, it states that the particular object Teddy is a member of the class of bears. Using set notation, we would write

$$\begin{aligned} \text{BEARS} &\subseteq \text{MAMMALS} \\ \text{TEDDY} &\in \text{BEARS} \end{aligned}$$

The key clue that the first sentence gives us for distinguishing that case from the second is that "bear," preceded by the article "A," is indefinite and refers to any and presumably all elements of the class. The "A" before "bear" in the first sentence signals an inclusion relationship, whereas its absence before "Teddy" in the second sentence indicates that Teddy is a particular object rather than one representative of a class and that "is a" means "is an element of the class" in this case. A list representing the membership relationship in this case could be

```
['Teddy', 'is-in', 'bear']
```

The verb *to be* is used in various senses, also. "Happiness is a sunny day" illustrates the use of *to be* in expressing a metaphor. This expression would probably not be meant (by whoever uses it) to indicate that the set of happinesses is a subset of the set of sunny days. More probably, someone saying such a thing would intend that the listener understand the meaning to be that sunny days lead to happiness. "A hot fudge sundae is vanilla ice cream with chocolate sauce" uses *is* to mean *consists of* and so makes a kind of definition. These uses of *is*

do not express inclusion, so one must be careful when attempting to describe the meanings of English sentences that use *to be* (in its various forms) in terms of inclusion. This illustrates one of the many difficulties of dealing with natural language in a mechanized way.

Let us restrict ourselves to the inclusion relation for the time being. It has a very important property: transitivity. This means, for example, that if `['bear', 'is-a', 'mammal']` and `['mammal', 'is-a', 'animal']`, then `['bear', 'is-a', 'animal']`. The fact that such a deduction can be made raises an important question: which assertions should be represented explicitly in a knowledge base and which should be deduced when needed, i.e., left implicit most of the time? The best answer is not always apparent. In order to illustrate the advantages and disadvantages of various alternatives, we develop enough mathematical tools so that the properties and problems of various representations can be understood.

4.4.2 Partial Orders and Their Representation

Let S be a set. A set of ordered pairs of elements of S is a *binary relation* on S. A binary relation, $\leq$, on S is a *partial order* if and only if

1. for each $x, x \leq x$ (reflexive property)
2. for each x and for each y, if $x \leq y$ and $y \leq x$, then $x = y$ (antisymmetry)
3. for each x, for each y, and for each z, if $x \leq y$ and $y \leq z$, then $x \leq z$ (transitivity)

For example take $S = \{a, b, c, d\}$ and let $\leq$ be the relation

$$\{(a, b), (c, d), (b, d), (a, d), (a, a), (b, b), (c, c), (d, d)\}.$$

The graph[1] of this relation is shown in Figure 4.1.

It is customary to say that "a precedes b," "c precedes d," and so on. Since every node must have a self loop, these convey no information on the diagram and can be deleted. Next, if one positions the nodes of this graph so that whenever x precedes y, x is higher on the page than y, then one can dispense with the arrowheads and leave just the line segments, as shown in Figure 4.2.

If we note that $a \leq d$ is implied by $a \leq b$ and $b \leq d$, the graph becomes less cluttered if the "redundant" segment is erased. The resulting picture is called a *Hasse diagram* for the partial order, and it is shown in Figure 4.3. The ordered pairs of elements for which lines actually appear in a Hasse diagram constitute the covering relation $\trianglelefteq$ for the original relation $\leq$. The *covering* relation is also called the *transitive reduction*. The original relation $\leq$ may be derived from its

[1]The *graph* of a relation is a diagram in which each element of the set is shown by a *node*, and every pair of elements that is part of the relation is shown by an *arc* connecting the corresponding nodes. An arc connecting a node to itself is called a "self loop."

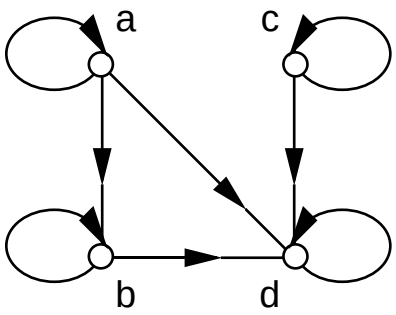

Figure 4.1: Ordinary graph of a relation.

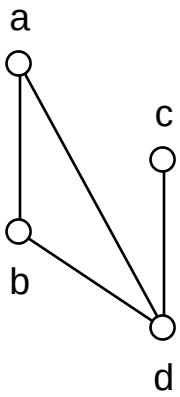

Figure 4.2: Graph with self loops and arc directions implicit.

covering relation $\trianglelefteq$ by taking the "transitive closure" of $\trianglelefteq$. The *transitive closure* of any relation is defined to be the smallest reflexive and transitive relation which includes the relation.

As an example of a partial order, let us consider the inclusion relation on some set. Let Ω be a universe of objects (i.e., some large set), and let S be the set of all subsets of Ω. If we consider any two elements x and y of S, then we have either $x \subseteq y$ or $y \subseteq x$, or $x \not\subseteq y$ and $y \not\subseteq x$. It is obvious that $\subseteq$ is a partial order. And thus when we have in Python the expression ['bear', 'isa', 'animal'], we have an explicit representation of one of the pairs in a partial order on some set of animal categories.

Suppose that a set of "facts" has been given, each of which is of the form "an X is a Y," and that inclusion is the relation expressed in each case. Two

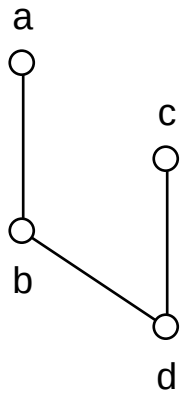

Figure 4.3: Hasse diagram for a transitive relation.

different approaches to the presentation of the set of facts are to store (1) the transitive reduction (covering relation) and (2) the transitive closure (all original and implied facts). The advantage of 1 is that less space is generally required in memory, since fewer graph arcs may be necessary. The advantage of 2 is that each fact is explicit and ready, so that less time may be required to respond to a question regarding the relationship of two entities. Presumably, the time necessary to deduce that $x \leq y$ (assuming that $x \leq y$ is true) depends largely on the length of the path from x to y in the Hasse diagram. However, if the transitive closure is stored, one must consider that the additional arcs of the graph may necessitate additional time in verifying $x \leq y$ even though the path length from x to z is only one arc. That is to say, in a graph that is closed with respect to transitivity, each node is likely to have many neighbors (i.e., a high valence), and this high valence may cause some accesses to run more slowly.

For cases in which the transitive reduction and transitive closure are almost the same, it makes little difference which of the two representations is used; however, in practical situations the two graphs are almost always quite different. Depending on how the search process is implemented, it may be possible to achieve a good compromise between the two approaches while holding both memory space and search time down. For example, by adding a few "shortcut" arcs to the transitive reduction, expected path length can sometimes be reduced considerably without any considerable increase in the degree of the graph for the covering relation, and this can shorten the lengths of some searches; however, it may lengthen others. Optimal representation of ISA hierarchies under various assumptions is a subject for research.

4.5 An ISA Hierarchy in Python

4.5.1 Preliminary Remarks

Knowledge of the inclusion relation among a set of categories is a very useful kind of information. Inclusion knowledge is instrumental in defining many nouns, and it can serve as the backbone of a representation system that incorporates other kinds of knowledge. Even without additional kinds of knowledge, however, inclusion knowledge can support a variety of types of queries. To illustrate some of the possibilities, a Python program is presented that demonstrates first how the relation of inclusion can be conveniently represented, then how the information can be accessed and used for making limited inferences, and finally how these mechanisms can be integrated into a simple conversational program. Since the knowledge manipulated by the program is all related to categories of things, we refer to the program as "Linneus." This Python program consists of the various functions that are explained subsequently.

4.5.2 The Use of Hash Tables to Store Relations

The hash-table facility of Python offers a convenient facility for representing relations such as inclusion. A hash table is an area of memory used to store associations. For example, a table of INCLUDES relationships might contain associations such as

```
['animal', ['dog', 'cat', 'bear']]
```

With this hash table, given the primary key, `'animal'`, the Python system can quickly retrieve the expression `['dog', 'cat', 'bear']`. It accomplishes the retrieval by applying an implementation-dependent "hash function" to the key to determine the memory address of the associated value. Hash functions get their name from the way they typically scramble the key to spread out the data in the table (but the scrambling is deterministic and repeatable!). To establish a hash table, we create a Python dictionary object. The following illustrates this.

```
INCLUDES = {}
INCLUDES['animal'] = ['dog', 'cat', 'bear']
INCLUDES['animal']  # evaluates to ['dog', 'cat', 'bear']
```

The information $x\ R\ y$ is accessible here by using x and R to formulate a hash-table access expression and then examining the list of strings returned, which should contain y if $x\ R\ y$ is true. To make this information accessible more generally, it could be represented also in two other forms: in a hash-table indexed on y and in one indexed on R. This could be accomplished by the following:

```
ISA = {}
ISA['dog'] = ['animal']
```

which makes 'animal' accessible from 'dog' via the ISA link, and

```
RELATIONS = {}
RELATIONS['includes'] =\
 [['animal', 'cat'], ['animal', 'dog'], ['animal', 'bear']]
```

which makes all the pairs of the INCLUDES relation accessible from the string 'includes'.

In the demonstration program, we use the first two forms: the INCLUDES hash table and the ISA hash table, but not the RELATIONS hash table.

```
'''Create hash tables to store the knowledge base components,
and define the functions for storing and retrieving knowledge.'''
from re import *   # Loads the regular expression module.

ISA = {}
INCLUDES = {}
ARTICLES = {}

def store_isa_fact(category1, category2):
    'Stores one fact of the form A BIRD IS AN ANIMAL'
    # That is, a member of CATEGORY1 is a member of CATEGORY2
    try :
        c1list = ISA[category1]
        c1list.append(category2)
    except KeyError :
        ISA[category1] = [category2]
    try :
        c2list = INCLUDES[category2]
        c2list.append(category1)
    except KeyError :
        INCLUDES[category2] = [category1]

def get_isa_list(category1):
    'Retrieves any existing list of things that CATEGORY1 is a'
    try:
        c1list = ISA[category1]
        return c1list
    except:
        return []

def get_includes_list(category1):
    'Retrieves any existing list of things that CATEGORY1 includes'
    try:
        c1list = INCLUDES[category1]
```

```
        return c1list
    except:
        return []

def store_article(noun, article):
    'Saves the article (in lower-case) associated with a noun.'
    ARTICLES[noun] = article.lower()

def get_article(noun):
    'Returns the article associated with the noun, or if none, the empty string.'
    try:
        article = ARTICLES[noun]
        return article
    except KeyError:
        return ''
```

The definitions of these functions follow:

4.5.3 Searching a Base of Facts

Given a base of facts, "A turbot is a fish," "A fish is an animal," etc., represented in Python as explained above, how can questions of the form "Is a turbot an animal?" be answered? Assuming that what is explicitly represented is a subrelation of the implied (transitive) one and that this subrelation is not necessarily transitive, the program should begin a search (for example) for `'animal'` from `'turbot'`. It should look first on the list of things that a turbot is (as represented in the hash table for ISA) and if `'animal'` is not found there, then in the hash table under each entry on the list for `'turbot'`, recursively searching until either `'animal'` is found or all possibilities have been exhausted without finding it, or all alternatives to a given depth limit have been exhausted.

Such a search is performed by the function `isa_test` described below:

```
def isa_test1(category1, category2):
    'Returns True if category 2 is (directly) on the list for category 1.'
    c1list = get_isa_list(category1)
    return c1list.__contains__(category2)

def isa_test(category1, category2, depth_limit = 10):
    'Returns True if category 1 is a subset of category 2 within depth_limit levels'
    if category1 == category2 : return True
    if isa_test1(category1, category2) : return True
    if depth_limit < 2 : return False
    for intermediate_category in get_isa_list(category1):
        if isa_test(intermediate_category, category2, depth_limit - 1):
            return True
```

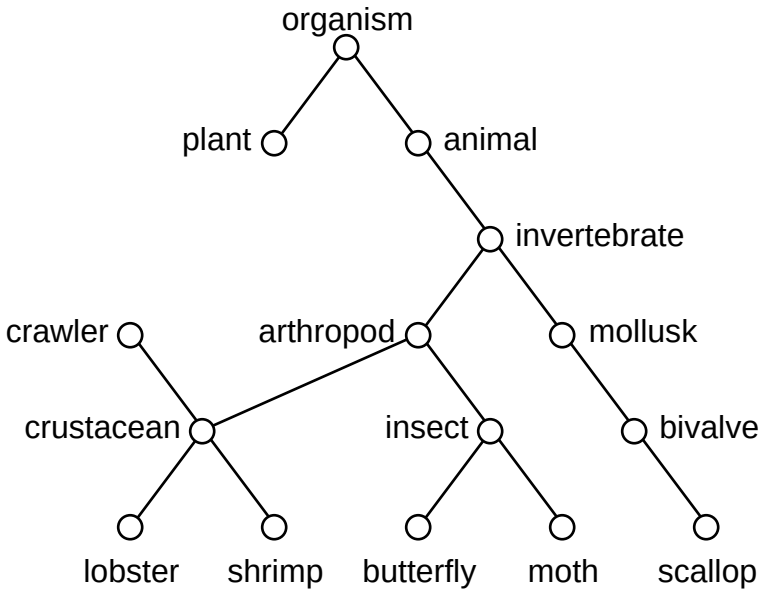

Figure 4.4: ISA hierarchy in which a lobster is an animal, and the path to prove it has fewer than 20 links.

```
    return False
```

The function `isa_test` takes three arguments, X, Y, and N where X and Y are strings like `'cat'` and `'animal'`, and N is a nonnegative integer giving the maximum number of levels for the recursive search. It sets up a catch tag and calls its helping function `isa_test1`. In `isa_test1`, a test is done to see if X is directly on the list for Y. This corresponds to a search of depth 1. The third case in `isa_test` tests N for 0 and cuts off the search along the current branch if so. In the last case, searches with maximum depth N − 1 are initiated from each of the strings appearing on X's ISA list. If any of these succeeds, `isa_test` returns True.

For the ISA hierarchy shown in Figure 4.4, the test `isa_test('lobster', 'animal', 20)` succeeds because there is a path from `'lobster'` to `'animal'` going only upwards and the length of the path is less than 20.

It may seem that the program might just as well have initiated a search from Y for X and traversed the inclusion arcs in the opposite direction. The length of the path from X to Y going forward is the same as that from Y to X going backward. However, the branching of the search may be drastically different in one case than in the other. If X is a leaf node in a tree and Y is the root, it takes less searching in general to find Y from X than to find X from Y. This is

because there are no choices when moving up a path of the tree. The strategy of searching forward from the present state or node (e.g., `'lobster'`) toward the goal (e.g., `'animal'`) is called "forward chaining." The complementary strategy is to search from the goal backward to the initial node or state, and this is called "backward chaining." Forward and backward chaining are described further in Chapter 5.

4.6 A Conversational Front End

The ability to store, retrieve, and perform simple inferences on relational data can support a variety of question-answering modes. Linneus demonstrates this by interpreting simple statements and questions and then invoking functions to search or manipulate the relational knowledge. The function `process` (shown below) is the main component of the conversational front end (human-to-knowledge-base interface). `process` implements a production system.

```
def linneus():
    'The main loop; it gets and processes user input, until "bye".'
    print 'This is Linneus.  Please tell me "ISA" facts and ask questions.'
    print 'For example, you could tell me "An ant is an insect."'
    while True :
        info = raw_input('Enter an ISA fact, or "bye" here: ')
        if info == 'bye': return 'Goodbye now!'
        process(info)

# Some regular expressions used to parse the user sentences:
assertion_pattern = compile(r"^(a|an|A|An)\s+([-\w]+)\s+is\s+(a|an)\s+([-\w]+)(\.|\!)*$'
query_pattern = compile(r"^is\s+(a|an)\s+([-\w]+)\s+(a|an)\s+([-\w]+)(\?\.)*", IGNORECAS
what_pattern = compile(r"^What\s+is\s+(a|an)\s+([-\w]+)(\?\.)*", IGNORECASE)
why_pattern = compile(r"^Why\s+is\s+(a|an)\s+([-\w]+)\s+(a|an)\s+([-\w]+)(\?\.)*", IGNOR

def process(info) :
    'Handles the user sentence, matching and responding.'
    result_match_object = assertion_pattern.match(info)
    if result_match_object != None :
        items = result_match_object.groups()
        store_article(items[1], items[0])
        store_article(items[3], items[2])
        store_isa_fact(items[1], items[3])
        print "I understand."
        return
    result_match_object = query_pattern.match(info)
    if result_match_object != None :
        items = result_match_object.groups()
```

```
        answer = isa_test(items[1], items[3])
        if answer :
            print "Yes, it is."
        else :
            print "No, as far as I have been informed, it is not."
        return
    result_match_object = what_pattern.match(info)
    if result_match_object != None :
        items = result_match_object.groups()
        supersets = get_isa_list(items[1])
        if supersets != [] :
            first = supersets[0]
            a1 = get_article(items[1]).capitalize()
            a2 = get_article(first)
            print a1 + " " + items[1] + " is " + a2 + " " + first + "."
            return
        else :
            subsets = get_includes_list(items[1])
            if subsets != [] :
                first = subsets[0]
                a1 = get_article(items[1]).capitalize()
                a2 = get_article(first)
                print a1 + " " + items[1] + " is something more general than " + a2 + " " + fir
                return
            else :
                print "I don't know."
        return
    result_match_object = why_pattern.match(info)
    if result_match_object != None :
        items = result_match_object.groups()
        if not isa_test(items[1], items[3]) :
            print "But that's not true, as far as I know!"
        else:
            answer_why(items[1], items[3])
        return
    print "I do not understand.  You entered: "
    print info

def report_chain(x, y):
    'Returns a phrase that describes a chain of facts.'
    chain = find_chain(x, y)
    all_but_last = chain[0:-1]
    last_link = chain[-1]
    main_phrase = reduce(lambda x, y: x + y, map(report_link, all_but_last))
```

```
    last_phrase = "and " + report_link(last_link)
    new_last_phrase = last_phrase[0:-2] + '.'
    return main_phrase + new_last_phrase

def report_link(link):
    'Returns a phrase that describes one fact.'
    x = link[0]
    y = link[1]
    a1 = get_article(x)
    a2 = get_article(y)
    return a1 + " " + x + " is " + a2 + " " + y + ", "

def find_chain(x, z):
    'Returns a list of lists, which each sublist representing a link.'
    if isa_test1(x, z):
        return [[x, z]]
    else:
        for y in get_isa_list(x):
            if isa_test(y, z):
                temp = find_chain(y, z)
                temp.insert(0, [x,y])
                return temp

def test() :
    process("A turtle is a reptile.")
    process("A turtle is a shelled-creature.")
    process("A reptile is an animal.")
    process("An animal is a thing.")

test()
linneus()
```

If the user types "A bear is an animal.", the first production rule will fire. That is, the `assertion_pattern` regular expression will match the input text. The result of the matching is used to obtain the components of the input corresponding to the parenthesized groups within the pattern. These are then saved either as the articles associated with the nouns in the sentence, or they are the nouns and are saved in the ISA hierarchy.

The second production rule handles user inputs such as "Is a bear an animal?" When the pattern is successfully matched to the input, `items[1]` and `items[3]` hold the two categories (e.g., `'bear'` and `'animal'`). Then `isa_test` is called on these to determine whether there is an ISA path from the first to the second.

The third rule handles questions such as "What is a bear?" In order to answer such a question, the program first checks to see if there are any paths

going up in the ISA hierarchy to supersets of `bear`. If there aren't any, then it searches in the other direction, looking for subsets of `bear`. In the latter case the answer might be something like "A bear is something more general than a grizzly."

The fourth rule handles "Why" questions. It provides explanations in response to questions such as "Why is a turbot an animal?" and makes use of the function `report_chain`. Before it does so, however, it uses `isa_test` to make sure that a turbot really is an animal (or whatever the presupposition expressed in the question happens to be).

The function `report_chain` works by first calling `find_chain` to get a list of the categories along the path from `x` to `y`. Then it creates a phrase that consists of a series of link explanations separated by commas, except that the last pair of explanations also have the word "and" separating them.

There are a couple of special cases for the Why rule do deal with. The first special case holds when the user has typed a question such as "Why is a horse a horse?" in which case Linneus reports the reason: that they are identical. The second case holds when there is a single ISA link from X to Y; this indicates that the fact in question was input by the user, rather than deduced by the program. In that case Linneus would report "Because you told me that."

The following illustrates a session with Linneus.

```
This is Linneus.  Please tell me "ISA" facts and ask questions.
For example, you could tell me "An ant is an insect."
Enter an ISA fact, or "bye" here: A mouse is an input-device.
I understand.
Enter an ISA fact, or "bye" here: What is a mouse?
A mouse is an input-device.
Enter an ISA fact, or "bye" here: A mouse is a rodent.
I understand.
Enter an ISA fact, or "bye" here: What is a mouse?
A mouse is an input-device.
Enter an ISA fact, or "bye" here: Is a mouse a rodent?
Yes, it is.
Enter an ISA fact, or "bye" here: An input-device is a machine.
I understand.
Enter an ISA fact, or "bye" here: A rodent is a mammal.
I understand.
Enter an ISA fact, or "bye" here: Why is a mouse a mammal?
Because a mouse is a rodent, and a rodent is a mammal.
Enter an ISA fact, or "bye" here: Is an input-device a mammal?
No, as far as I have been informed, it is not.
Enter an ISA fact, or "bye" here: bye
```

There are a number of interesting extensions that can be made to Linneus. For example, "HAS" links can be incorporated; these are described in the next

section. Other extensions are suggested in the exercises.

4.7 Inheritance

4.7.1 Inheritance from Supersets

With a representation of the inclusion relation on a set of classes based on the transitive reduction of the inclusion relation (or equally well by the "included by" relation), we can nicely handle additional relations with relatively little effort.

Let us consider the statements "A pheasant is a bird." and "A bird has feathers." From these we normally conclude that "A pheasant has feathers." That is, because the class `pheasant` is included by the class `bird`, certain properties of class `bird` are automatically "inherited" by class `pheasant`.

The general rule is: whenever we have x as a member of a set X that is a subset of a set Y, any property true of any member of Y must also be true of x. The fact that such a property of x can be determined by looking at Y means that the fact that x has this property need not be explicitly represented. As with the transitive reduction of a transitive relation, where it is only necessary to store a covering relation explicitly, we now may store some properties of classes only at "dominant" positions in the inclusion hierarchy.

4.7.2 HAS Links

Like the inclusion relation, the relation we call "HAS" is transitive. Here we use HAS to mean "has as parts." If a person has hands and a hand has fingers, then we can infer that a person has fingers. We might express these relationships as follows

```
"A person has a hand."
"A hand has a finger."
```

therefore,

```
"A person has a finger."
```

By avoiding plural forms here, we also avoid some problems of lexical analysis, which is more a subject in natural language understanding (see Chapter 9) than in the representation of knowledge.

The HAS relation is not only a transitive one by itself, and therefore capable of being efficiently represented by its transitive reduction, but it also may be viewed as a property that can be inherited with respect to the inclusion relation ISA. Let us write $X\ H\ Y$ to denote "an X has a Y"; i.e., members of class X have one or more members of class Y as parts. For example, HAND H FINGER means that a hand has one or more fingers. Then we note:

1. If $X \subseteq Y$ and ZHX, then ZHY.

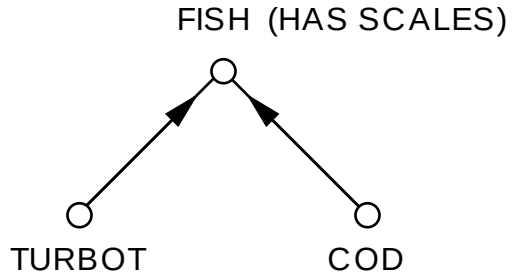

Figure 4.5: Inheritance of HAS relationships.

2. If $X \subseteq Y$ and YHZ, then XHZ.

Rule 1 may be called the rule of "generalizing HAS with respect to ISA," and Rule 2 may be called "inheritance of HAS with respect to ISA."

One can make inferences that involve sequences of these two rules and inferences by transitivity of ISA and HAS. For example, from the list of facts,

```
"A turbot is a fish."
"A fish is an animal."
"An animal has a heart."
"A heart is an organ."
"An organ has a cell."
```

we can infer

```
"A turbot has a cell."
```

Thus ISA and HAS are two partial orders that interact through the rules of generalization and inheritance above.

4.7.3 Multiple Inheritance

When the ISA hierarchy (transitive reduction of the inclusion relation) forms a tree or a forest (collection of trees) such that each class X has at most one immediate superclass, then the test to see whether a class X has some property P is easy to do. Each Y along the path from X to the root of the tree containing X is examined to see if it has property P. If any does, then X also does, by inheritance. Inheritance by TURBOT and COD of the property of having scales, from FISH, is illustrated in Figure 4.5.

A more complicated search is required when each class may have more than one immediate superclass. This is to say, the covering relation branches upward as well as downward in the general case. The search for a property P must

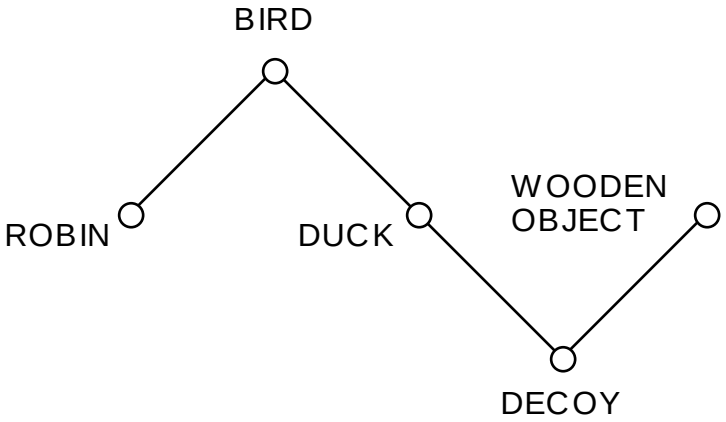

Figure 4.6: Multiple inheritance.

generally follow each upward branch until P is found or all possibilities are exhausted.

The possibility of multiple inheritance increases the potential for conflicting inherited values. Consider the example shown in Figure 4.6. A decoy¡ may be considered to be a kind of duck and, in turn, a bird. It is also a kind of wooden object. As a duck, it has a bill, a head, and a body. This is quite appropriate. As a bird, however, it ought to have a beating heart and be able to fly. But as a wooden object, it should not have a beating heart, nor should it be able to fly. The resolution of conflicts such as these may be difficult. In this particular case it is not; since a decoy is not truly a duck, the link between DECOY and DUCK should not be an inclusion link but a link such as RESEMBLES. A link of resemblance might only be allowed to pass inherited traits of certain types such as traits of appearance.

Generally speaking, properties are inherited from superclasses (i.e., along ISA arcs). However, they are not inherited along HAS arcs. Obviously the following "inference" is invalid:

```
"A person has a hand."
"A hand is-shorter-than one-meter."
```

"therefore"

```
"A person is-shorter-than one-meter."
```

Clearly one cannot treat HAS links in exactly the same manner as ISA links.

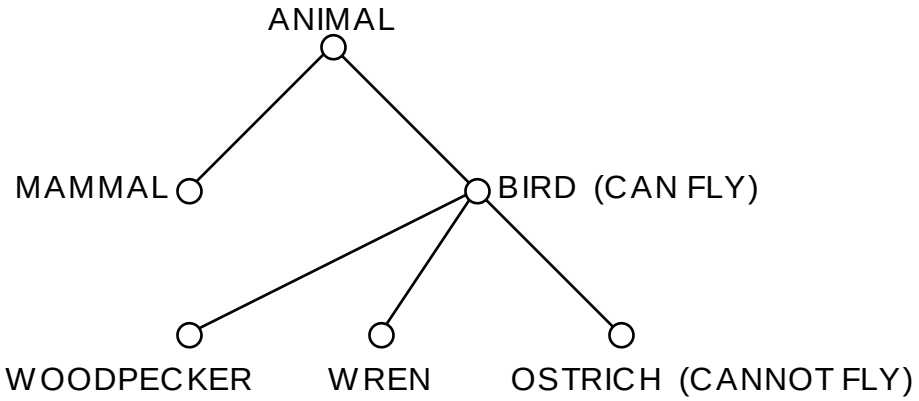

Figure 4.7: Default properties and exceptions.

4.7.4 Default Inheritance

There are some domains of knowledge in which exceptions to general rules exist. For example, it is usually useful to assume that all birds can fly. Certain birds such as the ostrich and the kiwi, however, cannot fly (even though they really are birds, unlike wooden decoys). In such a case it is reasonable to use a representation scheme in which properties associated with symbols in a hierarchy are assumed to be true of all subclasses, unless specifically overridden by a denial or modification associated with the subclass. For example, see Figure 4.7.

Under such a scheme, the fact that a woodpecker can fly is made explicit by following the (short) path from WOODPECKER to BIRD and finding there the property `'can fly'`. On the other hand, starting from OSTRICH, the property `'cannot fly'` is found immediately, and the default, which is further up the tree, is overriden.

Although inclusion (ISA) hierarchies often provide a conceptual organization for a knowledge base, they give immediate support to inferences of only a rather limited sort. The inferences involve either the transitivity of the inclusion relation or the inheritance of properties downward along chains in the hierarchy. An inclusion hierarchy provides a good way to organize many of the objects and concepts in a knowledge base, but it does not provide a representation scheme for noninclusion relationships, for logical or numerical constraints on objects, or for descriptions of the objects.

One way to build an ISA hierarchy into a more powerful structure is to add many different kinds of links to the system. We have already seen ISA, INCLUDES, and HAS links. Some more that can be added include ELEMENT-OF and OWNS. A data structure consisting of nodes that represent concepts or objects, together with labeled arcs representing relationships such as these,

is called a "semantic network" (or "semantic net" for short). Semantic nets are described later in this chapter.

In order to provide a general capability for representing many kinds of relations (rather than just inclusion and other binary relations), we turn to mathematical logic.

4.8 Propositional and Predicate Logic

4.8.1 Remarks

Mathematical logics are appropriate for representing knowledge in some situations. Two logics are commonly used. The propositional calculus is usually used in teaching rather than in actual systems; since it is essentially a greatly simplified version of the predicate calculus, an understanding of propositional calculus is a good first step toward understanding the predicate calculus.

On the other hand predicate calculus (or predicate logic) *is* often used as a means of knowledge representation in AI systems. Predicate logic is the basis for "logic programming" (as permitted by the programming language PROLOG, for example), and many specialists regard it as the single most important knowledge representation method. As we shall see, the predicate calculus is quite powerful but still has some serious shortcomings.

4.8.2 Propositional Calculus Expressions

The propositional calculus is a formal system for representing and manipulating statements according to logical rules. It is fairly simple. Here we present a brief summary of the propositional calculus, its use in representing statements, and some simple ways in which the representations can be manipulated.

Let P represent the statement "it is raining today." Let Q represent the statement "the picnic is canceled." Then the expression

$$P \wedge Q$$

represents the statement "it is raining today and the picnic is canceled." The expression $P \wedge Q$ can be read as "P and Q" and represents the *conjunction* of P and Q.

The expression

$$P \vee Q$$

stands for "it is raining today or the picnic is canceled." The expression $P \vee Q$ can be read as "P or Q" and represents the *disjunction* of P and Q.

The expression

$$P \Rightarrow Q$$

means "if it is raining today then the picnic is canceled." This expression can be read "If P, then Q" and is called the *conditional* of P and Q or the *implication* of Q given P. We can also read $P \Rightarrow Q$ as "Q when P" or, equivalently, "Q whenever P." In the conditional expression $P \Rightarrow Q$, the proposition P is called the "condition" or the *antecedent*, and the proposition Q is called the *consequent*.

The expression

$$P \Leftrightarrow Q$$

means "if it is raining today, then the picnic is canceled, and if the picnic is canceled, then it is raining today." This expression can be read "P if and only if Q" and is called the *biconditional* of P and Q or the *equivalence* of P and Q. We can also read $P \Leftrightarrow Q$ as‘ "If P, then Q, and if Q, then P."

The *negation* of P is written

$$\neg P$$

and means "it is not raining today" or, equivalently, "it is not the case that it is raining today." This expression can be read as "not P" or "it is not the case that P."

The symbols P and Q used here are called *proposition symbols* because each represents some proposition. The symbols

$$\wedge \quad \vee \quad \Rightarrow \quad \Leftrightarrow \quad \neg$$

are called *connectives* because they generally connect pairs of proposition symbols. An exception is $\neg$, which is a unary operator; although it is associated with only one proposition symbol, we still refer to it as a connective. The other connectives are binary operators.

The syntax of propositional calculus expressions can be formally described using Backus-Naur form (which may be regarded as a shorthand way of writing grammar production rules).

⟨exp⟩	::= ⟨prop symbol⟩
	::= ⟨prop symbol⟩
	::= ⟨constant⟩
	::= ¬ ⟨exp⟩
	::= (⟨exp⟩ ⟨binary op⟩ ⟨exp⟩)
⟨prop symbol⟩	::= ⟨prop letter⟩|⟨prop letter⟩⟨number⟩
⟨prop letter⟩	::= $P\|Q\|R\|X\|Y\|Z$
⟨constant⟩	::= **T** | **F**
⟨binary op⟩	::= $\wedge\|\vee\|\Rightarrow\|\Leftrightarrow$
⟨number⟩	::= 0|1|2|3|4|5

Parentheses may be omitted when the association of connectives with subexpressions is clear or is ambiguous but inconsequential.

The above syntax rules state that an expression is either a proposition symbol, a constant, the negation of another expression, or a list of another expression followed by a binary operator followed by one more expression. A proposition symbol is one of the letters P, Q, R, X, Y, or Z, optionally followed by one of the numbers 0 through 5. (Thus P, $P0$, ..., $P5$ are proposition symbols; with a more complex definition, any natural number could be allowed here.) A constant is either T (true) or F (false). A binary operator is either $\wedge, \vee, \Rightarrow$, or $\Leftrightarrow$.

The expressions we have just defined are sometimes called "well-formed expressions" or "well-formed formulas" or "WFFs" (pronounced "woofs") or simply "formulas."

4.8.3 Propositional Calculus Semantics

Given an expression such as $((P \wedge Q) \Rightarrow R)$, if we know the truth values for each of the propositions represented by P, Q, and R, we can mechanically determine the truth value of the whole expression. Suppose that P and Q are each true, and R is false. Then the overall expression's value becomes successively

$$((\mathbf{T} \wedge \mathbf{T}) \Rightarrow \mathbf{F})$$
$$(\mathbf{T} \Rightarrow \mathbf{F})$$
$$\mathbf{F}$$

The whole expression is false in this case. The rules for evaluating expressions are easily given in truth tables:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

4.8.4 Converse and Contrapositive

Given a conditional formula $P \rightarrow Q$, its *converse* is defined as $Q \rightarrow P$. Its *contrapositive* is defined to be $\neg Q \rightarrow \neg P$. If the formula and its converse both hold, then we may write $P \Leftrightarrow Q$. The contrapositive of $P \rightarrow Q$ is true if and only if $P \rightarrow Q$ is true.

4.8.5 Encoding English Sentences

Given some sentences in English describing some simple situation, it is usually easy to represent them in the propositional calculus. Let us describe the necessary steps, considering an example.

> If both networks A and B are up, then John can send a message to the world gateway. If either network C or D is up, then Marie

> can receive a message from the world gateway. If John can send a message to the world gateway and Marie can receive a message from it, then John can send to Marie.

Before we can isolate individual propositions in these sentences, we must expand "factored" phrases, so that each of the implied clauses has its own subject and predicate. We must also expand abbreviated phrases and dereference pronouns, so that clauses with the same meaning can be easily identified. Here is the result:

> If network A is up and network B is up, then John can send a message to the world gateway. If network C is up or network D is up, then Marie can receive a message from the world gateway. If John can send a message to the world gateway and Marie can receive a message from the world gateway, then John can send a message to Marie.

Next, we identify the atomic propositions (those that do not contain logical relations such as conjunction, disjunction, negation, or conditionals). We assign a proposition symbol to each one.

- P1: Network A is up.
- P2: Network B is up.
- P3: Network C is up.
- P4: Network D is up.
- Q1: John can send a message to the world gateway.
- Q2: Marie can receive a message from the world gateway.
- Q3: John can send a message to Marie.

After this, we are ready to represent each of the expanded English sentences using the proposition symbols and logical connectives. The first sentence, "If network A is up and network B is up, then John can send a message to the world gateway" is expressed as $((P1 \wedge P2) \Rightarrow Q1)$. The outer pair of parentheses can be dropped without loss of meaning. Encoding all the sentences results in this list:

1. $(P1 \wedge P2) \Rightarrow Q1$
2. $(P3 \vee P4) \Rightarrow Q2$
3. $(Q1 \wedge Q2) \Rightarrow Q3$

This set of three expressions can be considered the end of the process. However, sometimes we wish to sum up our statements in a single expression. When we have a set of expressions each of which is assumed to be true, there is an implied conjunction of them. Thus the following formula represents the original set of sentences by itself:

$$((P1 \wedge P2) \Rightarrow Q1) \wedge ((P3 \vee P4) \Rightarrow Q2) \wedge ((Q1 \wedge Q2) \Rightarrow Q3).$$

Once a set of statements has been encoded into the propositional calculus, it is easier to make logical inferences from the statements. As we will see later, certain inference methods can be easily performed by computer.

4.8.6 Inference in the Propositional Calculus

There are important things we can do with expressions of propositional calculus without needing to know whether the component propositions are true or false. That is, if we assume that some expressions are true, we can derive new ones which are guaranteed to be true if the assumptions are. To obtain new expressions which logically follow from the starting expressions, we use one or more *rules of inference*. Some important rules of inference are stated below.

1. Modus Ponens:

Assume:	$P \Rightarrow Q$
and	P
Then:	Q

   For example, suppose we know that if it snows today, then school will be canceled, and suppose we also know that it is snowing today. Then, by the rule of modus ponens, we can logically deduce that school will be canceled today.

2. Disjunctive Syllogism

Assume:	P
Then:	$P \vee Q$

   For example, suppose that you own a car. Then you can truthfully say that you either own a car or live in a 17th century castle (regardless of whether or not you live in any castle).

3. Resolution

Assume:	$P \vee Q$
and	$\neg P \vee Z$
Then:	$Y \vee Z$

For example, suppose that either John passes his final or John goes into seclusion. Suppose further that either John flunks his final or he misses Paula's pre-finals party. We can conclude that either John goes into seclusion or he misses Paula's prefinals party.

Resolution is very important in automatic theorem proving and logical reasoning. In Chapter 6 we will see a more flexible kind of resolution in the predicate calculus.

Certain kinds of propositional calculus expressions deserve special names. An expression that is always true, no matter what assumptions one may make about the propositions represented, is a *tautology*. The expression $P \vee \neg P$ is a tautology. An expression that is always false (i.e., can never be true) is a *contradiction*. For example, $P \wedge \neg P$ is a contradiction. An expression that is not a contradiction is said to be *satisfiable*.

The propositional calculus is very limited as a method of knowledge representation. Perhaps its primary use is in studying some aspects of the predicate calculus, which is much more powerful.

4.8.7 Predicate Calculus Motivation

One generally needs more expressive power in a knowledge representation language than is offered by the propositional calculus. There, one must build on propositions, and one cannot "get inside" a proposition and describe the objects that make up the proposition. The predicate calculus, on the other hand, does allow one to work with objects as well as with propositions.

Suppose that we wish to represent the following knowledge in a formal system:

Every fish can swim. Bubbles is a fish.

With the propositional calculus, we could assign one proposition symbol to each of these two statements.

- P: Every fish can swim.
- Q: Bubbles is a fish.

However, there would be no way to make the obvious inference from them (that Bubbles can swim). An alternative would be to specialize the first axiom ("Every fish can swim") to get "If Bubbles is a fish, then Bubbles can swim." We could then work with the following atomic propositions:

- P: Bubbles is a fish.

- Q: Bubbles can swim.

Then our knowledge is represented by the pair of propositional calculus formulas: $P \Rightarrow Q$, and P. This at least allows us to make the obvious inference (Q) by applying the inference rule *modus ponens*. However, as a representation of the original statements, it is poor, since the generality of the rule "every fish can swim" has been lost and only the special case for Bubbles has been kept.

We might attempt to remedy the situation by embedding a variable in our English-language clauses:

- P: X is a fish.
- Q: X can swim.
- R: Bubbles is a fish.

Then we could encode our statements as $P \Rightarrow Q$, and R. After all, English-speaking people sometimes speak this way, using symbols like X as variables. But now we have several problems. If X is a variable, must it refer to the same object in Q as it does in P? The propositional calculus doesn't provide any guidance on this. Secondly, we can no longer make the inference that Bubbles can swim. (We can't even represent it.) If we could bind Bubbles to X, then we might have some means to make progress, but again, the propositional calculus does not provide any binding mechanism whatsoever, let alone anything about consistency of bindings. What we need is a system of knowledge representation that provides mechanisms for writing general statements using variables, guidelines for making substitutions (bindings) for those variables, and generally a means for representing the details of individual statements. The predicate calculus is such a system.

Because of its generality and the direct way in which it can support automatic inference, the predicate calculus is probably the single most important method for knowledge representation. Here we present the basic syntax and semantics of the predicate calculus. This form of knowledge representation will be used in Chapter 6 in the discussions there of theorem proving and logic programming.

Before discussing the detailed syntax and semantics of the predicate calculus, let us briefly consider a possible representation of the above example in the predicate calculus.

- Fish(x): A predicate stating that x is a fish.
- Swims(x): A predicate stating that x can swim.
- Bubbles: A constant referring to a particular individual.

The original statements are represented as follows:

1. $(\forall x)(\text{Fish}(x) \Rightarrow \text{Swims}(x))$.

2. Fish(Bubbles).

The symbol $\forall$ is called the universal quantifier and is read as "for all" or "for each." This representation of the original statements is sufficient to make the inference that Bubbles can swim, that is, Swims(Bubbles). (Such inferences are discussed in Chapter 6).

4.8.8 Predicate Calculus Syntax

An expression in the predicate calculus is much like one of propositional calculus to which more detailed descriptions have been added. Where one might use the symbol P in the propositional calculus to represent the statement "The apple is red," in the predicate calculus one separates the predicate (quality of being red) from the objects (or subjects, here the apple) and writes:

$$R(a)$$

or, more explicitly,

$$\text{Red(Apple)}$$

Here, the symbol "Red" is a predicate and "Apple" is a constant that represents a particular object in a domain or universe of objects. The use of standard predicate symbols $P, Q, R, P_0, P_1, \ldots$ is common. If we have a particular domain in mind, then it is often convenient to use more meaningful names for predicates, like "Red" instead of P or R.

As another example, the propositional calculus doesn't provide a way to refer to specific objects within statements such as "the golden egg" in "The golden egg is heavy." On the other hand, the *predicate* calculus does provide for symbols to represent objects and then allows these to be used as components of statements. For example, the constant symbol a may refer to a particular golden egg and a predicate symbol P may assert that something is heavy. The statement $P(a)$ then could state that the golden egg is heavy. The *constant symbols* a, b, c, ... are standard symbols used in the predicate calculus to denote particular objects in some domain. (As with predicate symbols, we are free to use more meaningful names when convenient.) The predicate symbols $P, Q, R, \ldots$ (or more meaningful names) are used to denote qualities or attributes of objects or relationships among objects that are either true or false. For example, $Q(x, y)$ might assert that x is less than y in some domain of numbers such as the reals. *Function symbols* such as the standard ones $f, g, h, f_0, f_1, \ldots$ denote mappings from elements of the domain (or tuples of elements from the domain) to elements of the domain. For example, $P(a, f(b))$ asserts that predicate P is true on the argument pair a and the value of the function f applied to b. We may sometimes use more meaningful function symbols, like "square(x)." A predicate generally takes arguments, and the arguments are represented by expressions called "terms." A function also takes arguments represented by terms. The

number of arguments to a predicate or function is called the *arity* of the predicate or function. For example, the arity of predicate P in $P(x, y, z)$ is 3. A predicate of arity n is also called an n-place predicate. Also, a function of arity n is called an n-place function. Terms are defined below.

Logical connectives such as $\neg, \wedge$, and $\vee$ are the same as in the propositional calculus. *Variable symbols* x, y, z, x_1, x_2, and so on represent potentially any element of the domain, and they allow the formulation of general statements about many elements of the domain at a time. These standard variable symbols are used almost exclusively; it is not common to use longer symbols to represent variables.

The predicate calculus provides two *quantifiers*, $\forall$ and $\exists$, which may be used to build new formulas from old. For example, $\exists x P(x)$ expresses that there exists at least one element of the domain that makes $P(x)$ true. Also, $\forall x P(X)$ expresses that $P(x)$ is true for every member of the domain.

The rules for building up syntactically correct formulas are as follows:

1. Any constant or variable taken by itself is a *term*.
2. Any n-place function applied to n terms is a *term*.
3. Any n-place predicate applied to n terms is a *well-formed formula*.
4. Any well-formed expression of the propositional calculus (see page 117) is a *well-formed formula* (of the predicate calculus).
5. Any logical combination of well-formed formulas is also a *well-formed formula*. (All the logical connectives of the propositional calculus may be used.)
6. Any well-formed formula F may be made into another *well-formed formula* by prefixing it with a quantifier and an individual variable, e.g., $(\forall x)(F)$. Parentheses should be used when necessary to make the scope of the quantifier clear. $\forall x F$ is acceptable when unambiguous.

Some examples of terms using standard symbols are a, x, a_2, y_0, $f(a)$, $f(x, y)$, and $f_2(f_1(b, x_0), g(x))$. Some terms using nonstandard symbols are $Bubbles$ (see earlier example), $\sqrt{2}$, $\sqrt{x}$, 17, and $\sin(x + \sqrt{y})$.

Some examples of well-formed formulas using standard symbols are P, T, $P \vee F$, $P \Rightarrow Q(x)$, $P(x)$, $\forall x P(x)$, $\exists y Q(y)$, $P_1(a)$, and $\forall x \exists y(\neg Q(x, y) \wedge R(a, y))$. An example of a well-formed formula using nonstandard symbols is $(\forall x)(\text{Fish}(x) \Rightarrow \text{Swims}(x))$.

4.8.9 Predicate Calculus Semantics

In order to assign meaning to a predicate calculus formula, we must first identify the "domain" of objects that it relates to. The domain is a set of objects, either

finite or infinite, that are of interest in the problem or the statements. For example, in our earlier example about fish's being able to swim, the predicate calculus formulas representing the knowledge were $(\forall x)(\text{Fish}(x) \Rightarrow \text{Swims}(x))$, and Fish(Bubbles). The domain here might be the set of all animals. It could be a smaller set, such as the set of all aquatic animals, or a larger set, such as the set of all biological organisms.

A smaller and more accurately defined example domain for a formula is the finite set of integers from 0 to 3. Let us use this domain to discuss the business of interpreting a formula. For a formula $P(a, f(b, c))$ we can make an assignment between each constant symbol and an element of the domain. For example, we might take a to be 0, b to be 2, and c to be 3. Next, let us associate with each n-place function symbol some mapping from n-tuples of elements of the domain to elements of the domain. For example, we can take $f(x, y) = (x+y) \bmod 4$. With these assignments $f(b, c) = 1$. Next let us associate with each n-place predicate symbol an n-place relation on the domain. (An n-place relation on a set S is a set of n-tuples of elements of S.) For example, let us associate with the 2-place predicate symbol P the set of 2-tuples $\{(0, 1), (1, 2), (2, 3)\}$. The predicate in the formula is considered to be true if the tuple of its arguments occur in the set. Thus, for our example, we take $P(x, y)$ to be true when $y = x + 1$, that is, when y is the successor of x. For our example, these assignments make the formula $P(a, f(b, c))$ true. The same assignments make the formula $P(a, b)$ false, since 2 is not the successor of 0.

Given a formula F (or set of formulas S), the combination of (1) a domain, (2) an assignment of constant symbols to members of the domain, (3) an assignment of function symbols to mappings on the domain, and (4) an assignment of predicate symbols to relations on the domain is called an "interpretation" of F (or S). In mathematical logic, an interpretation for a formula is often called a *model* for the formula provided the formula is true for that interpretation.

To summarize the above model for $P(a, f(b, c))$, we have

1. domain: $\{0, 1, 2, 3\}$
2. assignment of constant symbols: $a = 0, b = 2, c = 3$
3. assignment of function symbols: $f(x, y) = (x + y) \bmod 4$
4. assignment of predicate symbols: $P : \{(0, 1), (1, 2), (2, 3)\}$

If a formula F does not contain any variables, then given an interpretation, we can figure out whether F is true or false. We do this by first finding for each predicate whether it is true or false under the interpretation. Then we use the logical rules of the propositional calculus (for $\neg$, $\wedge$, $\vee$, and so on) to combine the truth values into an overall truth value.

If the formula contains quantifiers and variables, then it can be more complicated to determine whether the formula is true or false. For example, to verify the formula $\forall x P(x)$, we may have to test each element of the domain to see if it

satisfies P. If we discover an element that fails to satisfy P, we have our answer — the formula is false. However, if we do not discover such an element quickly and the domain is infinite, we could have a difficult time. This issue is taken up in detail in Chapter 6.

4.8.10 Inference with the Predicate Calculus

The predicate calculus can be a convenient representation for facts and rules of inference provided that a suitable set of functions and predicates is available with which to build formulas.

Predicates readily represent relations such as inclusion. For example,

Isa(Bear, Mammal).

The predicate calculus is an attractive representation mechanism for knowledge in AI systems because well-known techniques of logical inference can easily be applied to such a representation. One thing the predicate calculus does *not* provide is any particular set of given predicates with meanings, or functions or domain. These must be provided by the knowledge engineer in developing predicate calculus representations for knowledge.

The predicate calculus can be used to formalize the rules for inheritance that were discussed earlier in the chapter. Let us reconsider the specific syllogism: "A pheasant is a bird, and a bird has feathers implies that a pheasant has feathers." In general, when we have $X \subseteq Y$ and $\forall y \in Y$, $P(y)$, then we can infer $\forall x \in X$, $P(x)$. Since $\forall x \in X$, $P(x)$ is derivable from $X \subseteq Y$ and $\forall y \in Y$, $P(y)$, it need not be explicitly represented. (Logical inference techniques are discussed in Chapter 6.)

4.8.11 Encoding into Predicate Calculus

Given some English sentences, the job of representing them in the predicate calculus may either be easy or not, depending on the complexity of the meaning to be captured and explicitly represented. As we have shown earlier, it is possible to represent the sentence "Every fish can swim" as a single proposition P, and technically, this is also a well-formed formula of the predicate calculus. However, the range of inferences we can make is then very limited. What we usually try to do is to represent the given statement at a level of detail so as to capture the principal objects and relationships expressed by the sentence as distinct symbols in the representation. Let us first consider a simple example.

Every dictator is cruel.

We first surmise that the domain of interest is some set that includes dictators and at least some cruel people. We might therefore take the domain to be all heads of state (presidents, kings, queens, dictators, etc.). Within this domain

are the subsets "dictators" and "cruel heads of state." We can create predicate symbols to assert membership in each of these subsets: Dictator(x) means x is a dictator, and Cruel(x) means x is a cruel head of state. The word "every" immediately suggests using the universal quantifier $\forall$. The verb "is" does not convert as obviously. It is used here to assert that the "cruel" property is true of the sentence subject. The way to represent this assertion about all members of a subset of the domain is to use the conditional connective $\Rightarrow$ between the predicates. The result is the formula $\forall x(\text{Dictator}(x) \Rightarrow \text{Cruel}(x))$.

Now let us consider a more complex example.

> Any normal bird can fly. An ostrich is an abnormal bird. A whirlybird is not a normal bird, either. A whirlybird is not even a bird. Anything that's not an abnormal bird must either be a normal bird or a whirlybird. Tweety is a bird, but not a normal bird. Tweety has her wings clipped. Tweety has two parents. They are normal birds.

This example will help to bring out some of the important things to keep in mind when translating English into predicate calculus.

Here are the main steps: (1) refining the English into separate, expanded statements, (2) identifying the domain, (3) identifying constants, (4) structuring the domain — finding important subsets of the domain, (5) identifying and expressing functional relationships, (6) writing predicates to represent relationships, (7) identifying and expressing logical relationships, and (8) identifying and expressing quantification over parts of the domain.

Let us begin our example by rewriting the English to emphasize quantified propositions and omit rhetorical features irrelevant to the logical meaning of the text:

> Every normal bird can fly. An ostrich is a bird, and not a normal bird. A whirlybird is not a normal bird. A whirlybird is not a bird. Every object that is not a bird that is not normal is either a normal bird or a whirlybird. Tweety is a bird. Tweety is not a normal bird. Tweety has her wings clipped. There exist precisely two individuals that are parents of tweety, and they are both normal birds.

An alternative to this last sentence is the pair of sentences, "Tweety's mother is a normal bird. Tweety's father is a normal bird." In general, one has to make an interpretation of the given sentences. Since there were no explicit mentions of mothers or fathers, this alternative seems less appropriate. An alternative to "Tweety has her wings clipped" is the pair of sentences "Tweety has clipped wings. Tweety is a female." Decisions must be made about how much implied information should be represented, i.e., about what is relevant.

The next step is to identify the domain. The domain should encompass all the objects and classes of objects mentioned in the statements. Here, the class

of birds must be included. Also, the class of whirlybirds (a term for helicopters) must be included. Perhaps we could take some much larger set (such as the set of all animals and all machines) as our domain. If we did, then the following sentence from the original set of statements would not be true: "Anything that's not an abnormal bird must either be a normal bird or a whirlybird." For this to be true, an object cannot be a dog or a car or something else. We must restrict the domain to the union of the set of birds and the set of whirlybirds.

Next, we identify constants. There is just one in our example, the bird named Tweety.

Then, we structure the domain. We have already done part of this. The important subsets of the domain are the birds, the normal birds, the abnormal birds (birds that are not normal), the female birds, the parents of Tweety, and the whirlybirds.

Then we can identify functional relationships. If we consider "parents of" to be a function of a bird, then the set of parents would have to be an object in the domain. We have already limited the domain to birds and whirlybirds, and not *sets* of those objects. Therefore, we should not attempt to describe this relationship this way. Instead, we can either use other functions (mother and father, each of which is a function that maps Tweety to a single domain element) or a predicate. This brings us to predicates.

We need predicates that express relationships, such as Parent(x, Tweety), and predicates that express properties or belonging to a subset of the domain, such as Bird(x), Normal(x), Female(x), Whirybird(x), and Clipped-wings(x).

Next, we should identify the quantifications in the statements and over what parts of the domain the quantifications are valid. "Every normal bird . . . " can be written $\forall x[(\text{Bird}(x) \wedge \text{Normal}(x)) \Rightarrow \ldots]$. The statement "There exist precisely two individuals that are parents of Tweety. . . " should be represented using two existential quantifiers — one for each parent. In order to express that there are precisely *two* parents, we will need an additional predicate, Equals(x, y). The negation of Equals(x, y) expresses that x and y are distinct, and we will use this to help make sure that our representation specifies that there are at least two parents of Tweety. We will also use Equals(x, y) to help specify that there are no more than two parents of Tweety.

Combining the above techniques with the use of the logical connectives $\neg, \wedge, \vee, \Rightarrow$, and $\Leftrightarrow$, we can complete our example, and come up with the following set of formulas in the predicate calculus:

1. $\forall x[(\text{Bird}(x) \wedge \text{Normal}(x)) \Rightarrow \text{Can-fly}(x)]$
2. $\forall x[(\text{Ostrich}(x) \Rightarrow (\text{Bird}(x) \wedge \neg \text{Normal}(x))]$
3. $\forall x[(\text{Whirlybird}(x) \Rightarrow \neg(\text{Bird}(x) \wedge \text{Normal}(x))]$
4. $\forall x[(\text{Whirlybird}(x) \Rightarrow \neg\text{Bird}(x)]$
5. $\forall x[\neg(\text{Bird}(x) \wedge \neg \text{Normal}(x)) \Rightarrow ((\text{Normal}(x) \wedge \text{Bird}(x)) \vee \text{Whirlybird}(x))]$

6. Bird(Tweety)
7. $\neg$(Normal(Tweety) $\wedge$ Bird(Tweety))
8. Clipped-wings(Tweety)
9. $\exists x \exists y[\neg$ Equals$(x, y) \wedge$Parent$(x,$ Tweety$) \wedge$Parent$(y,$ Tweety$) \wedge [\forall z($Parent$(z,$ Tweety$) \Rightarrow ($Equals$(z, x) \vee$ Equals$(z, y)))]]$
10. $\forall x[($Parent$(x,$ Tweety$) \Rightarrow ($Normal$(x) \wedge$ Bird$(x))]$

Expressing the fact that Tweety has exactly two parents is done in a slightly roundabout manner. First we assert the existence of an x and a y that are distinct and that are parents of Tweety. Then we assert that any z that is a parent of Tweety must be equal to either x or y. In this manner, we assert that there are at least two parents and at most two parents. This is equivalent to asserting that there are exactly two parents. We use both existential quantifiers and a universal quantifier to make this assertion.

This more complicated example illustrates not only the process of converting statements in English to statements in the predicate calculus, but also some of the sorts of situations where the predicate logic formulations are awkward. If we had to express the fact that there are exactly five platonic solids or exactly ten sides to a decagon, the resulting expression with lots of existential quantifiers and not-equals predicates would be very messy. In spite of its limitations, however, the predicate calculus is a very important knowledge representation system for artificial intelligence. Its limitations can sometimes be overcome by extending it or combining it with other methods.

4.9 Frames of Context

Another problem with the predicate calculus as a representation scheme is that it does not provide a means to group facts and rules together that are relevant in similar contexts. Such groupings may not be necessary in small knowledge bases. However, a lack of overall organization in large bases can have costly consequences. In order to provide organizational structure, various methods have been proposed including "partitioned semantic networks" (described later) and "frames."

By providing the knowledge in modules called "frames," the designer makes life easier for the algorithms that will access the knowledge. A frame is a collection of knowledge relevant to a particular object, situation, or concept. Generally there are many pieces of information in each frame, and there are many frames in a knowledge base. Some frames may be permanent in the system; others may be created and destroyed during the course of problem solving. The term "frame" appears to be borrowed from physics, where it usually refers to a coordinate

Slot Name	*Filler*
FRAME NAME	KITCHEN-FRAME
FRIDGE-LOC	(3 5)
DISHWASH-LOC	(4 5)
STOVE-LOC	(5 4)
PANTRY-LOC	NIL

Figure 4.8: Frame representing a kitchen and its attributes.

frame or frame of reference in three-dimensional space. It suggests a concern with a subset of the universe, from a particular point of view.

A frame provides a representation for an object, situation, or class in terms of a set of attribute names and values. A frame is analogous to a Python atom with its property list, to a Python structure, or to a "record" data type in Pascal.

4.9.1 A "Kitchen" Frame

Let us suppose that we are designing a household robot. This robot should do useful things such as vacuum the living room, prepare meals, and offer drinks to the guests. If we ignore the mechanical aspects and consider only the problem of designing the knowledge base for this robot, we must find an overall organization for it. The robot should know about the living room and the things that are likely to be found there, such as the living room furniture. It should also know about the kitchen and all the key appliances there: stove, fridge, garbage disposal, dishwasher, and possibly fire alarm. Since our robot is to be designed not for one particular house but many, its knowledge base should not presuppose exact locations for these things. The exact coordinates for each item could be established at the time the robot is installed or delivered. A reasonable way of organizing such a knowledge base is according to the rooms of the house. We set up one module (frame) for each room. There can be a frame for the living room, a frame for the bathroom, a frame for the kitchen, etc. Our next step is to design each frame.

4.9.2 Slots and Fillers

A frame commonly consists of two parts: a name and a list of attribute-value pairs. The attributes are sometimes called "slot names" and the values "fillers." Therefore a frame is a named collection of slots and the fillers associated with the slots. A frame can easily be represented in Python using a symbol for the frame name and part or all of its property list to hold the attribute-value pairs. For a kitchen we might have a frame such as shown in Figure 4.8.

It may be that a slot is to be filled with the name of another frame or with a list of other frames. If we add a slot named "ADJACENT-ROOMS" to the KITCHEN frame, it might get as value (DINING-ROOM BACK-HALL CELLAR-STAIRS). The interlinking of frames to one another creates a network that can be viewed as a semantic network. (However, the term "semantic network" is applied to a large variety of relational knowledge bases.)

4.9.3 Schemata

Often a frame is associated with a class of objects or a category of situations. For example, a frame for "vacations" may provide slots for all the usual important features of a vacation: where, when, principal activities, and cost. Frames for particular vacations are created by *instantiating* this general frame. Such a general frame is sometimes called a *schema*, and the frames produced by instantiating the schema are called *instances*. The process of instantiating the schema involves (1) creating a new frame by allocating memory for it, (2) linking it to the schema, for example by filling in a special "instance-of" slot, and (3) filling in the remaining slots with particular information for the vacation in question. It is not necessary that all the slots be filled. The relationship between a schema and two instances of it is shown in Figure 4.9.

Each schema in a collection of schemata gives the general characteristics that pertain to a concept, class of objects, or class of situations. Therefore, a schema acts as a template or plan for the construction of frames for particular objects or situations.

4.9.4 Attachments to Slots

A slot may be provided with a default value and/or information related to the slot. Since such information is neither the value of the slot (which is "filled in") nor the name of the slot, the associated information is said to be *attached*. Attached information may be of such kinds as the following:

1. a constraint that must be satisfied by the filled-in value for the slot

2. a procedure that may be used to determine the value for the slot if the value is needed (this is called an *if-needed* procedural attachment)

3. a procedure that is to be executed after a value is filled in for the slot (this is called an *if-added* procedural attachment)

By attaching procedures or constraints to slots, frames can be made to represent many more of the details of knowledge relevant to a problem, without losing their organizational effectiveness.

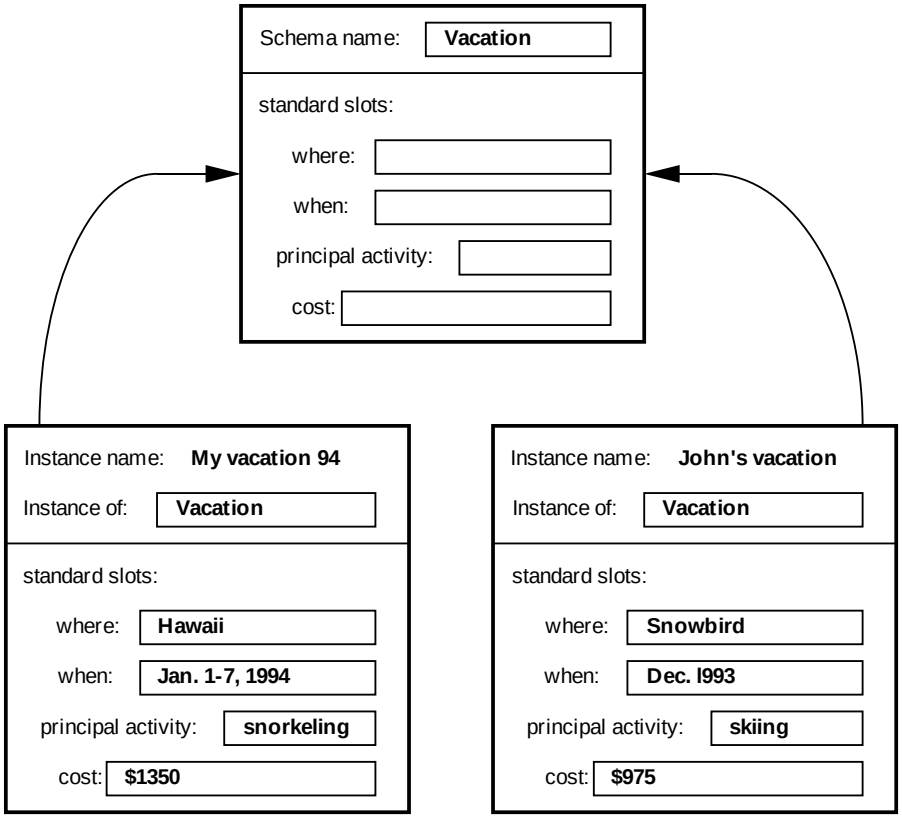

Figure 4.9: Schema and two instances of it.

4.10 Semantic Networks

4.10.1 Motivation

Earlier in this chapter it was suggested that ISA hierarchies could be extended into more general semantic networks by adding additional kinds of links and nodes. In fact, such linked data structures have been used often to represent certain kinds of knowledge in AI systems. In this section, we present the rationale, methods, and an evaluation of semantic networks as an approach to knowledge representation.

Semantic networks were first developed in order to represent the meanings of English sentences in terms of objects and relationships among them. The neural interconnections of the brain are clearly arranged in some type of network (apparently one with a highly complex structure), and the rough similarity

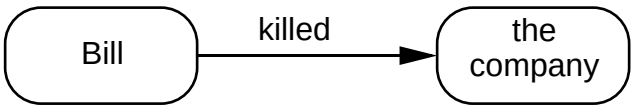

Figure 4.10: Simple semantic net for "Bill killed the company."

between the artificial semantic nets and the natural brain helped to encourage the development of semantic nets. The notion of accessing semantic information through a kind of "spreading activation" of the network, analogous to brain activity spreading via neurons, is still an appealing notion.

There are some more practical aspects to semantic nets, also. There is an efficiency to be gained by representing each object or concept once and using pointers for cross references, rather than naming an object explicitly every time it is involved in a relation (as must be done with the predicate calculus, for example). Thus it is possible to have very little redundancy in a semantic net. Not only can we get an efficiency in space, but search time may be faster as well; because the associations between nodes are represented as arcs in a graph, it is possible to use efficient graph-search methods to locate desired information. If the network structure is implemented with an adjacency list scheme, a search is likely to be much faster than if a long list of relationships has to be scanned every time an association is followed.

Semantic networks can provide a very general capability for knowledge representation. As we shall see, they can handle not only binary relations but unary and higher-order relations as well and so are, in theory, as powerful as the predicates of the predicate calculus.

Unlike the predicate calculus, however, there is no standard semantic network, reasoning methods are not provided by the techniques themselves, and semantic net support for universally or existentially quantified statements is either not provided, nonstandard, or messy. On the positive side, the semantic net approach is clearly valuable for providing a graphical way for the AI researcher or system designer to view knowledge, and it often suggests a practical way of implementing knowledge representations.

4.10.2 Representing Sentence Semantics

Since semantic nets were originally developed for representing the meanings of ordinary sentences, it is appropriate to consider an example in that vein.

Perhaps the simplest way to design semantic networks to represent sentences is first to restrict the allowable sentences to certain kinds that use only nouns, verbs, and articles. Then one can set up a network node for each noun (including its article, if any) and a link for the verb. Such a net for the sentence "Bill killed the company" is shown in Figure 4.10.

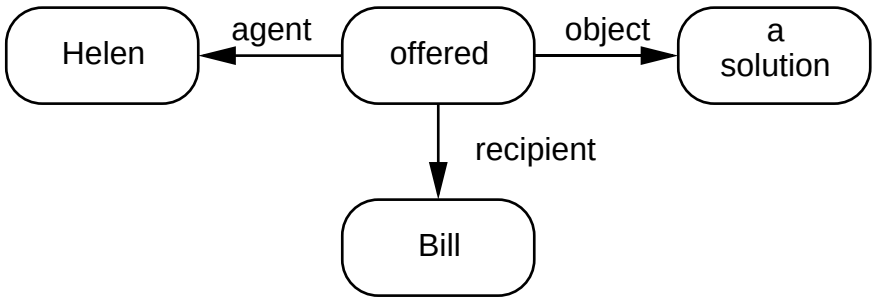

Figure 4.11: Semantic net for "Helen offered Bill a solution."

It is true that much can be done with such representations, as the Linneus program illustrates. Unfortunately, however, relatively few of the sentences we use are simple enough to be represented this way. Many verbs, for example, take both a direct object and an indirect object. Consider the sentence "Helen offered Bill a solution." Here the direct object is "a solution" and the indirect object is "Bill." In order to associate all three nouns and the verb, it is appropriate to create a node in a semantic net for the verb as well as each noun, and then to link the verb to each noun with an arc labeled with the relationship of the noun to the verb. In this case, the indirect object, Bill, plays the role of recipient of the offer (see Figure 4.11).

If this fragment of a semantic net is to be a part of a large one representing the many aspects of a complicated story or situation, the nodes of the fragment are likely to duplicate existing nodes. If the repeated nodes are simply merged, there may be problems. For example, if the larger net also contains a similar representation of the sentence "David offered Bob a ride home," then merging the "offered" nodes would confuse the two offering events and possibly allow the erroneous inference of "Helen offered Bob a ride home."

The situation is much improved if the specific offering events are represented as separate nodes, each of which is an instance of a general node representing the class of all offering events. Similarly, if there is another "solution" node in the network, it probably represents a different solution from the one Helen offered. Thus the noun phrase "a solution" should also be represented as an instance node linked to a node representing some class. It would be desirable for the sake of consistency for each particular nominal in the sentence to be represented as an instance. This leads to the net in Figure 4.12.

Representing adjectives and prepositional phrases can be handled with additional nodes and links. For the example with an adjective, "The armagnac is excellent," a node for the attribute "excellent" is set up and a link labeled "quality" may be used (as in Figure 4.13).

A prepositional phrase modifying a noun may be represented by a node for

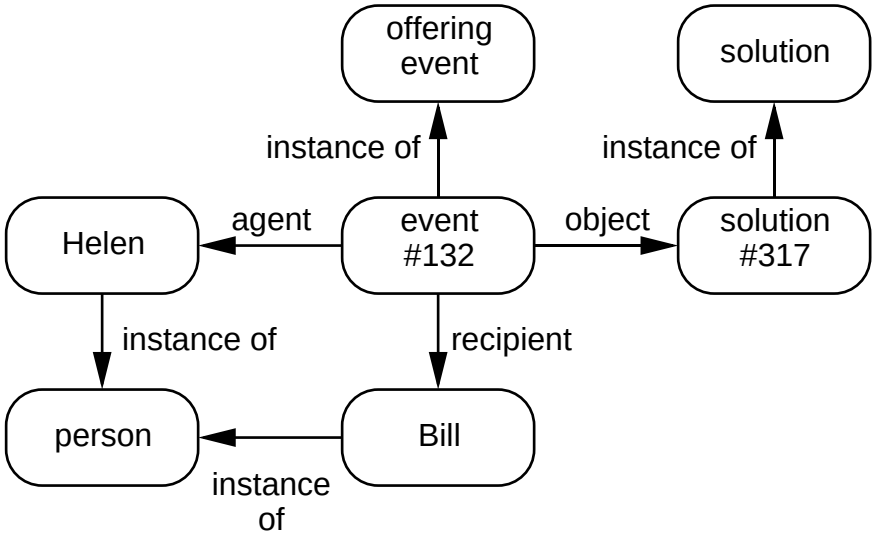

Figure 4.12: A net with class and instance nodes for "Helen offered Bill a solution."

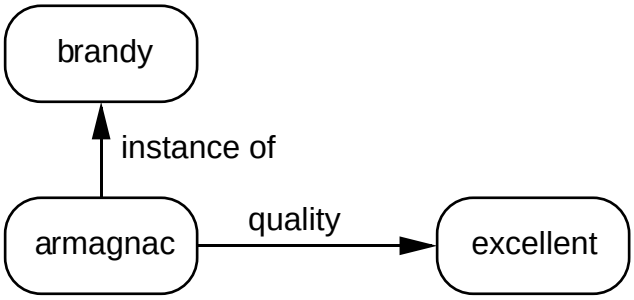

Figure 4.13: Net with an attributive node.

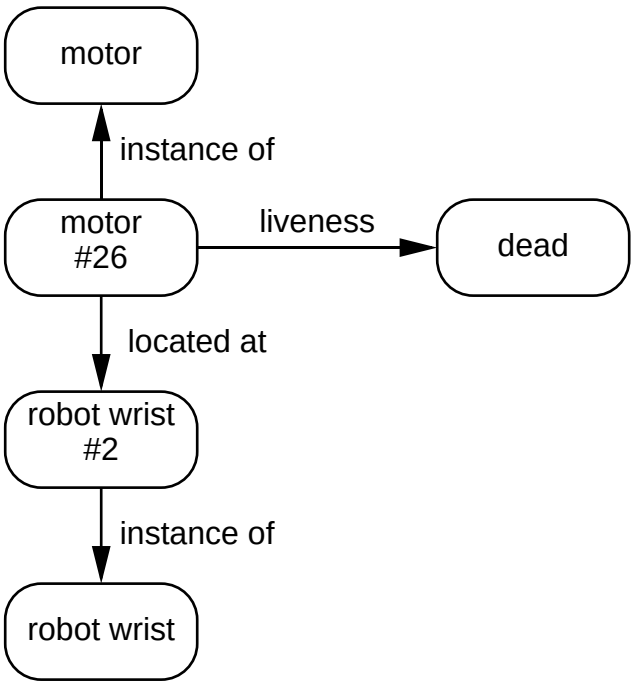

Figure 4.14: Net with a representation of a prepositional phrase.

the nominal object of the preposition, pointed to by an arc from the noun that is modified; the arc is labeled with the relationship specified by the preposition. Thus for "the motor in the robot's wrist is dead," we have the net of Figure 4.14.

It is not necessarily easy to build a useful semantic net representation for a sentence. Even when one is provided with a good set of class nodes and arc labels, it can be unclear which nodes and arc types to use and how much of a sentence's meaning should be represented explicitly. Let us now consider a slightly more complicated sentence: "Laura traded her car for Paul's graphics board." A net for this is shown in Figure 4.15.

The sentence suggests that Laura took the initiative in the trade. However, it is usual for a trade to be a cooperative activity, so that it would make sense to have an additional link from "event#793" to "Paul" with the label "co-agent." But, since the sentence does not begin with "Laura traded *with* Paul," it appears to be safer not to infer that Paul was an active participant in the event.

This example contains another case that is difficult to decide. The sentence makes clear that Laura was the owner of a car. Should an "owned" link be established from "Laura" to "car"? One problem with putting in such a link is the time-dependent nature of the truth of the fact it represents. After the trade was completed, Laura no longer owned the car. Anyway, the ownership

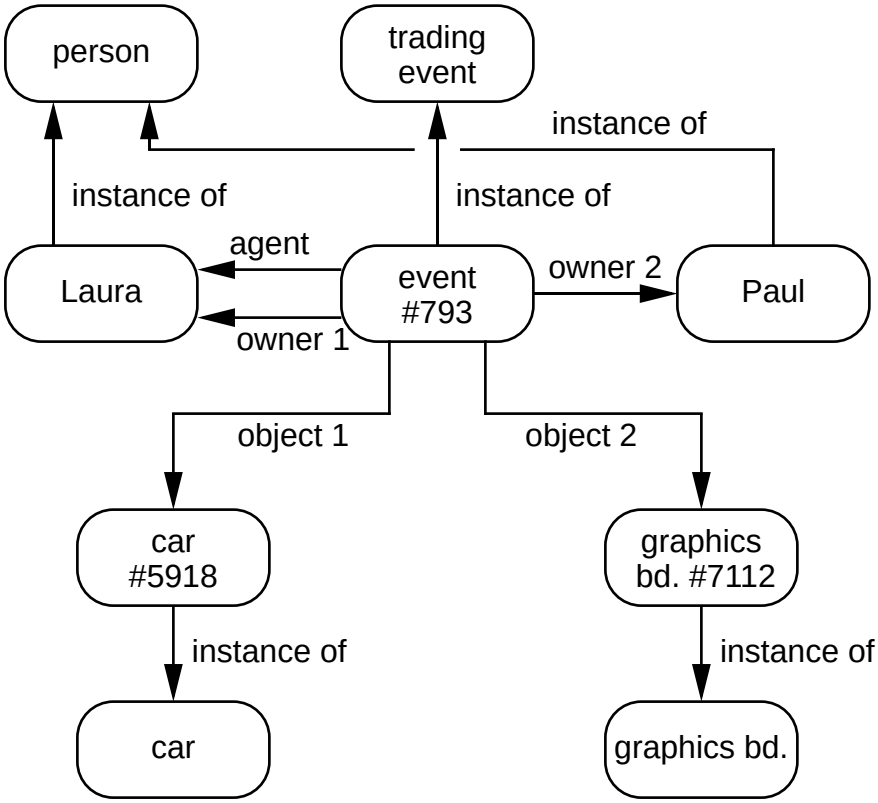

Figure 4.15: Net for "Laura traded her car for Paul's graphics board."

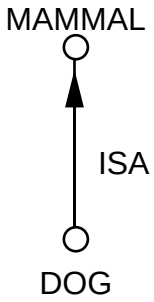

Figure 4.16: Semantic net for a named binary relationship.

information is implicit in the net because the node for the particular event is linked both to "Laura" and to "car" with appropriate labels on the arcs.

One thing that should be clear from this example is that there is a problem concerning time. In the representation of an event, one generally cannot represent the state of things before the event and the state of things after the event consistently without some kind of separation of representations. One way of maintaining consistency without physically separating the representations is to add temporal information to some or all of the links in a net. Then one could put a link between "Laura" and "car" with the label "owned before event#793" and a link between "Laura" and "graphics board" with the label "owned after event#793." This certainly complicates the representation of the links and is likely to slow down some inferences.

Should a semantic net represent the current states of the relevant objects or their histories or both? This depends on the kinds of inferences a system is supposed to make. If a system is to be able to answer questions such as, "What was the relationship between the defendant and the victim at the time of the crime?" then clearly temporal information must be incorporated. On the other hand, if a robot is expected only to be able to navigate through a room and see where all the obstacles are, it probably doesn't have to keep track of the history of its environment; it only needs to represent the current state of the environment it finds itself in.

4.10.3 Representing Nonbinary Relations

At first glance, semantic networks appear to be more effective in representing named binary relations (i.e., two-place predicates) than other kinds of relations. For example, Isa(dog, mammal) is represented as in Figure 4.16. It should be made clear, however, that semantic networks are not limited in this respect; they can represent an n-ary relation with no loss of information. For example, consider the quaternary relationship expressed by the four-place predicate gives(John,

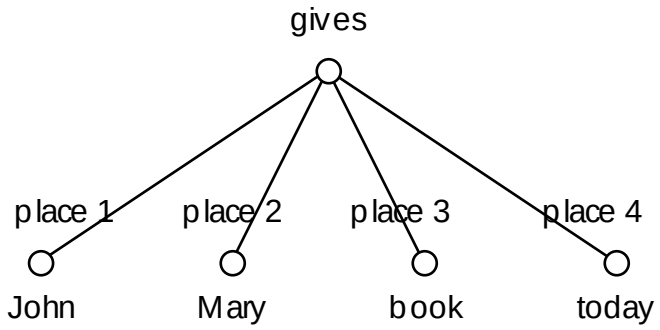

Figure 4.17: Semantic net for a four-place relationship.

Mary, book, today). A net representing this can be constructed with a node for the predicate symbol, a node for each argument, and an arc from the predicate node to each argument node labeled with the place number of the argument (as in Figure 4.17). Of course, there may be more appropriate names for the arc labels than "place 1," etc. In this case, a better set of labels would be agent, recipient, object, when. The one disadvantage of using semantic nets to represent n-ary relations is that there is some overhead that results from the need to create these new arc labels.

4.10.4 Semantic Primitives and Combining Forms

The basic concepts necessary to represent everyday experiences are called "semantic primitives." Semantic primitives must be adequate for representing such things as events, stories, and situations. Systems of such primitives have been proposed by R. Schank and by Y. Wilks. Typically, each primitive is either an entity, such as a person, a thing, or a part of another entity; an action, such as to fly, to be, or to want; a case, such as "on behalf of," "surrounding" or "toward"; a qualifier such as "good," "much," or "unfortunate"; or a type indicator, such as "how," which indicates that a related phrase modifies an action, or "kind," which indicates that a related phrase modifies an entity. A deeper treatment of semantic primitives is given in Chapter 9 as a basis for natural language understanding.

4.11 Constraints

A method for representing knowledge that is based on the predicates of predicate calculus but that is augmented with procedural information is "constraints." A constraint is a relationship between two or more items that the system, in the course of solving a problem, must attempt to satisfy or keep satisfied.

4.11.1 Constraint Schemata

A constraint may be represented simply as a predicate of predicate calculus. However, it has proved useful to represent constraints as instances of "generalized constraints" or *constraint schemata*. A constraint schema may be represented by giving it a name, listing the formal parameters that represent the parts of each constraint modeled by the schema, and listing rules that allow any one of the parameters to be computed when the others are known.

An example of a constraint schema is the following one, which could be used to represent Ohm's law in an electronics problem-solving system.

```
('constraints',
  ('parts', ['voltage', 'current', 'resistance']),
  ('rules', [('take', 'voltage', '(current * resistance)'),
             ('take', 'current', '(voltage / reseistance)'),
             ('take', 'resistance', '(voltage / current)')]))
```

This constraint would make it easy for the current to be computed in a circuit if the voltage and resistance were known. Note that the constraint provides the knowledge in a form that can be used not just in updating a predetermined variable when the others change, but for whichever variable may have an unknown value at some time when the other variables have known values.

It is possible to make a constraint representing Ohm's law that is yet more useful by adding rules that allow updating with knowledge of only one variable, when that variable has the value zero and is either CURRENT or RESISTANCE.

```
('constraints', 'ohms-law',
  ('parts', ['voltage', 'current', 'resistance']),
  ('rules', [('if', 'current = 0', ('take', 'voltage', 0)),
             ('if', 'resistance = 0', ('take', 'voltage', 0)),
             ('take', 'voltage', '(current * resistance)'),
             ('take', 'current', '(voltage / reseistance)'),
             ('take', 'resistance', '(voltage / current)')]))
```

Both representations suffer from the problem that division by zero is not prevented. This could be fixed by modifying the last two rules, and is left as an exercise for the reader.

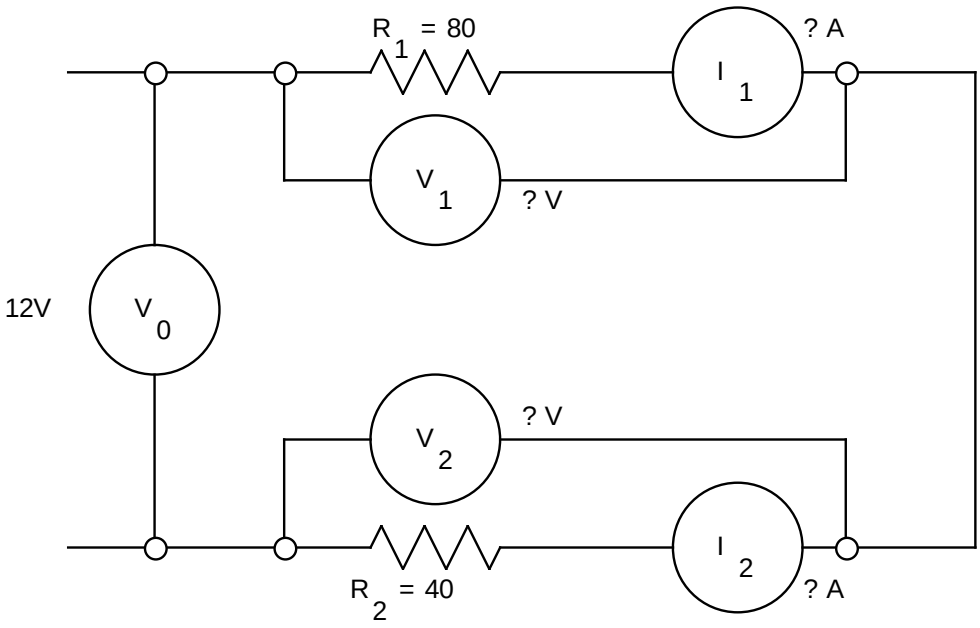

Figure 4.18: Constraint network for an electronic circuit.

4.11.2 Using Constraints

In order to use a constraint schema such as this one in representing a complex situation, one or more instances of it may be used in conjunction with instances of other such schemata, and the instances may form a "constraint network." An example is shown in Figure 4.18.

This diagram represents an electronic-circuit problem in which one voltage and two resistance values are given, and the object is to determine the voltages V_1 and V_2, and currents I_1 and I_2. The constraints for this problem, expressed as ordinary equations, are

$$\begin{aligned} V_0 &= 12 \\ R_2 &= 40 \\ V_1 &= I_1 R_1 \\ I_0 &= I_1 \\ R_0 &= R_1 + R_2 \end{aligned} \qquad \begin{aligned} R_1 &= 80 \\ V_0 &= I_0 R_0 \\ V_2 &= I_2 R_2 \\ I_1 &= I_2 \end{aligned}$$

As instances of constraint schemata, the constraints are

```
('initial', 'v0', 12)
('initial', 'r1', 80)
('initial', 'r2', 40)
```

```
('ohms_law', 'v0', 'i0', 'r0')
('ohms_law', 'v1', 'i1', 'r1',)
('ohms_law', 'v2', 'i2', 'r2')
('series_current', 'i0', 'i1')
('series_current', 'i1', 'i2',)
('series_resistance', 'r0', 'r1', 'r2')
```

The designing of suitable representations for the schemata INITIAL, SERIES_CURRENT, and SERIES_RESISTANCE is left as a series of exercises for the reader.

4.11.3 Satisfying Constraints

Constraints are most frequently used as part of the representation of a problem. In the example of Figure 4.18, the problem of finding the current through R1 may be solved by an iterative constraint-satisfaction procedure. Such a procedure repeatedly finds a variable for which the variables on which it depends have defined or updated values, and it computes a new value. If all constraints are satisfied, the procedure halts, and if the procedure ever detects that it is making no progress, it also stops. Sets of constraints may be inconsistent and thus have no solution. In some cases, a set of constraints may have a solution, but existing methods of constraint satisfaction are inadequate for finding it. Constraint-satisfaction procedures can be quite involved, and they continue to be a subject of active research.

Constraints have been most successful in representing numeric relationships. However, they have also been used successfully in combinatorial relationships involving finite sets of objects or labels. In such a situation, they may be used to filter out particular combinations of labels or partial states to arrive at a solution. Visual scene analysis and natural language understanding are two areas where combinatorial constraints have been useful.

4.12 Relational Databases

4.12.1 Remarks

It would do justice neither to AI nor to the field of database systems to omit relational databases from a serious survey of knowledge representation methods. Database techniques, while they have not been widely used in AI experiments, are fairly mature, well understood, and being brought into AI systems now. The relational database approach is particularly good at handling large, regularly structured collections of information. As with other representation methods, relational databases are set up to store some relationships explicitly and to permit implicit relationships to be recovered through computation. Capability for certain useful transformations is generally provided by relational database systems;

ANGIOSPERMS			
Plant name	*General form*	*Seed body*	*Products*
Wheat	Grass	Grain	Bread
Corn	Grass	Kernel	Meal
Potato	Tuber	Eye	Fries
Oak	Tree	Acorn	Floors
Oak	Tree	Acorn	Desks

Figure 4.19: Relation in a relational database.

selection, projection, and joining, for example, are common. These can be used both for access to, and to a limited degree, for inference on, the database. However, these operations may be used in connection with more powerful inference methods (such as resolution in the predicate calculus) to attain a combination of intelligence and efficiency in a knowledge-based system.

4.12.2 n-ary Relations

Database management systems frequently are based on the "relational approach." A relation in the database sense is more general than the binary relations that we discussed in Section 4.4 on concept hierarchies. Rather than a set of ordered pairs, when talking about relational databases, the term "relation" refers to a set of ordered n-tuples, where n may vary from one relation to the next. For example, a the following is ternary relation:

$$\{(a, a, b), (a, c, d), (d, c, d), (d, c, e)\}.$$

It is customary to display such relations in tabular form:

a	a	b
a	c	d
d	c	d
d	c	e

In an n-ary relation there are n "fields." Each field has a name and a "domain." The *domain* is the set of values from which elements in the field may be drawn. The first field is sometimes called the primary key of the relation. Not only do the fields have names, but the entire relation usually has a name also. For example, consider the relation "angiosperms" shown in Figure 4.19.

The relational method is convenient insofar as certain standard operations on relations tend to be supported by database management systems. The operations are useful not only for querying and updating the database, but also for extracting subrelations and merging relations to form composites.

R2	
Plant name	*General form*
Wheat	Grass
Corn	Grass
Potato	Tuber
Oak	Tree

Figure 4.20: Projection of "angiosperms" onto "plant name" and "general form."

It is interesting to note that binary and ternary relations can be easily represented in Python using the property lists of symbols. Hash tables could also be used. Relations of higher order can also be represented in Python as lists of tuples that are themselves lists.

4.12.3 Selection

With the relation in Figure 4.19 we ought to be able to find the answer to a question such as: "What is the name of each angiosperm that is a grass?" The procedure is simply to scan top to bottom looking at the "general form" attribute of each tuple, and whenever the value "grass" is found, output the value of the "plant name" field in the same row. A somewhat more general formulation of this kind of process is the following: the *selection* from an n-ary relation R according to a predicate $P(x_1, \ldots, x_n)$, is a new relation R' that is a subset of R, each of whose tuples satisfies P. The effect of a selection, therefore, is to extract some (and possibly none or possibly all) of the rows of R. Of course, the predicate P can be designed to ignore most of its arguments, and if it is understood which argument a unary predicate is to be applied to, it is not necessary to specify an n-ary one.

4.12.4 Projection

In a relation having n fields, it may be the case that only k of them are relevant to a particular application. A new relation, generally smaller than the original, can be obtained by making a copy of the original, but deleting the fields that are not wanted. At the same time, any duplications in the set of k-tuples thus formed are removed. For example, projecting the relation "angiosperms" (Figure 4.19) with respect to the first two fields yields the new relation R2 shown in Figure 4.20.

The effect of projection is to extract one or more columns of the table representing the relation, and then to remove any redundant rows. Projection is

R3	
General form	*Size*
Grass	Small
Tree	Large
Bush	Medium

Figure 4.21: Two-place relation containing a field "size."

R4		
Plant name	*General form*	*Size*
Wheat	Grass	Small
Corn	Grass	Small
Oak	Tree	Large

Figure 4.22: Join of R2 and R3.

analogous to selection, except in this possibility of having to remove redundant rows.

4.12.5 Joins

Two relations can be combined by the *join* operation if they share one or more common domains; that is, one can find a column in one relation whose elements are drawn from the same set as those in some column of the other relation. In such a case, the join is obtained by finding all "compatible pairs" of tuples and merging such pairs into longer tuples. A tuple from one relation is *compatible* with a tuple of the other relation if for each shared domain the first tuple's value matches the second tuple's value. In the tuples of the join, each shared field is represented once, so that the length of a tuple of the join is strictly less than the sum of the lengths of the tuples used to produce it. Note that any tuple in one of the two initial relations does not participate in the join if its values in the common fields do not match those of some tuple in the other relation. Consider the relation R3 shown in Figure 4.21. The join of R2 with R3 is the relation R4, shown in Figure 4.22. If one starts only with R2 and R3, then the join of R2 and R3 is required before selection can be applied for answering the query: "What are the names of the small plants." This is because selection must be applied to a relation that contains both the "plant name" field and the "size" field in order to obtain the answer to the question.

The relational approach to knowledge representation does not seem as appropriate for complicated semantic knowledge of the sort that could support dialogs in natural language, as other schemes such as those organized by frames or by class hierarchies. Although the relational approach is general enough to represent anything, the operations typically available (projection, join, and some others for updating relations) are not very helpful for solving problems and making inferences.

On the other hand, relational databases do well at handling bulky bodies of information that have regular, homogeneous structures, and they can be useful as components of intelligent information systems.

4.13 Problems of Knowledge Representation

Some of the problems of knowledge representation have already been mentioned: handling defaults and exceptions and explicit versus implicit representations (e.g., transitive closures vs. transitive reductions). Here, some additional problems are discussed. These problems are related to the quality, completeness, and acquisition of knowledge.

4.13.1 The Closed-World Assumption

It is difficult to believe that a doctor knows everything there is to know about treating a common cold. There are lots of aspects of viruses that scientists, let alone doctors, do not understand that might be relevant to treating colds. Similarly, in playing a game of chess, a player may see very well where all the pieces are and be able to foresee various possible unfoldings of the game, but he or she probably does not know his or her opponent well enough to predict the reply to each move. The opponent may be thinking about his or her love life and suddenly make some unexpected move.

Except in very artificial situations, a person or machine doesn't have all the knowledge needed to guarantee a perfect performance. As a result, the system or the designer of the system needs to recognize the limits of the system's knowledge and avoid costly errors that might result from assuming that all there was to know was known.

The Linneus program stays within its limits when it responds negatively to a question such as "Is a house an animal?" Its reply is "SORRY NOT THAT I KNOW OF." This points up a limitation of Linneus; it does not provide a way to represent negative information, e.g., "a house is not an animal." The program does, however, avoid concluding falsely that, for our example, "a house is not a building," but only because it hadn't been told that a house is a building.

Nonetheless, there are times when it is reasonable to assume that the system knows everything there is to know about a problem. For example, if the classical problem of missionaries and cannibals crossing a river is posed, it would be

"cheating" to propose a solution using a bridge, since no bridge is specifically mentioned in the problem.[2] Thus, it can be appropriate to make use of the *closed-world assumption* that anything that cannot be derived from the given facts is either irrelevant or false.

When the closed-world assumption can be made, that is very nice, because it implicitly represents a whole lot of things that might otherwise have to be explicitly stated (e.g., "You may not use a bridge," "You may not drug the cannibals," etc.).

There is another way, though, that the closed-world assumption can lead to trouble. Its use could imply that "if you can't find a solution to a problem then there is no solution." This may often be true, but may be false even more often. For example, if a system were inefficient at finding solutions and didn't try long enough to find one, it could mistakenly infer that no solution exists. Even if the system is a good one, it might not be able to verify a true statement, as K. Gödel showed in his famous work on the incompleteness of arithmetic.[3]

In designing a system for representing knowledge, one should decide whether a closed-world assumption can be used. If not, then it may be necessary to provide ways to represent negative information (e.g., "A house is not an animal"). Alternatively, compromises are possible where the absence of an explicit or derivable fact *suggests* that the negation may be true but does not assure it. Information obtained from suggestions would always be qualified when reported to the user, but could be used freely by the system for the purpose of directing search where it would be helpful if true but have little effect if false.

4.13.2 Knowledge Acquisition

The question of how knowledge should be represented is related to the questions of where the knowledge comes from and how it is acquired. Here are three reasons why these questions are related:

1. The representation chosen may affect the acquisition process (this is discussed further in Chapter 8).

2. The acquisition process can suggest useful representations (tools exist that build up knowledge structures from dialogs with human experts).

[2]The missionaries and cannibals problem is stated as follows: There are three missionaries and three cannibals on one bank of a river that they must cross. There is a rowboat there that can carry up to two people, including the one who rows. If there ever are more cannibals than missionaries on one side of the river, then the missionaries on that side (if any) will be eaten. Otherwise, all will cooperate in peaceful transport. What is the plan by which the entire party of six can cross the river uneaten?

[3]The original title of Gödel's paper is Die Vollständigkeit der Axiome des logischen Funktionenkalküls, and it was published in *Monatshefte für Mathematik und Physik*, Vol 37, pp. 349-360 in 1930.

3. It is possible that some of the knowledge that a system is to use should stay in the form in which it is available (e.g., text files representing books and reports).

Methods for building knowledge structures automatically or interactively are discussed in Chapter 8.

4.14 Large Knowledge Bases: Cyc

4.14.1 Motivation for Cyc

A notable project to develop a very large base of common-sense knowledge is the Cyc project. Begun in late 1984 at the Microelectronics and Computer Technology Corporation, this project, under the direction of Douglas Lenat, set out to build a knowledge base that would provide computer systems with enough "common sense" knowledge to permit them to (1) avoid "brittleness" in expert systems (i.e., not fail miserably as soon as they must deal with an issue outside of a very narrow domain of expertise) and (2) acquire new knowledge through natural language understanding. The name "Cyc" comes from the core of "encyclopedia"; the Cyc knowledge base should provide a complement to the knowledge in an encyclopedia. Cyc would provide a computer system enough common sense to be able to understand an encyclopedia article. Without common-sense knowledge, it is far too difficult for a computer system to be able to understand such articles.

The development of Cyc is particularly interesting from the perspective of knowledge representation. The large volume of information — thousands of frames and tens of thousands of logical assertions — poses problems in itself. The organization of the knowledge base to support common-sense reasoning is a challenge as well.

4.14.2 Epistemological and Heuristic Levels

It is worth mentioning some of the mechanisms that Cyc uses. The knowledge base contains two connected representation systems, one based on predicate calculus and the other based on a collection of templates, frames, and efficient inference mechanisms. The logic component is called the "epistemological level," and the computationally efficient level is called the "heuristic level." Each item of knowledge is represented at both levels. The epistemological level presents a standard interface between Cyc and the outside world (knowledge engineers, applications programs). However, inference at the epistemological level is not particularly efficient, and so the heuristic level is used internally by most of the inference procedures. A special representation language, CycL, was developed for the project. CycL supports both the epistemological level and the heurstic

level, and in this way overcomes many of the limitations of the predicate calculus or a frame system taken individually.

4.14.3 Ontologies

The general selection of what sort of objects and concepts Cyc knows about is also of interest. Cyc's domain and the domain's structure constitute its "ontology." Before discussing Cyc's ontology in particular, let us elaborate a bit on the concept of ontology itself.

Webster's New World Dictionary of the American Language defines "ontology" as "1. the branch of metaphysics dealing with the nature of being, reality, or ultimate substance: cf. PHENOMENOLOGY 2. *pl.* **-gies** a particular theory about being or reality." Speaking about an artificial intelligence system, we generally mean the second of these. That is, the ontology of a system is what it assumes to exist and what it assumes to be the nature of those entities and their relationships to each other. With regard to a set of formulas in the predicate calculus, the ontology of the system comprises the domain, the structure of the domain in terms of subdomains, and the particular predicates and functions involved in the formulas. In a system to represent common-sense knowledge, it is important that an ontology include notions of time, space, events, intangible objects and thought, as well as material objects.

Cyc's ontology includes a number of interesting categories, the most general of which is "Thing." Every object in Cyc is an instance of Thing. One pair of broad categories is InternalMachineThing and RepresentedThing. These partition the class Thing into things like the number 9 and the string "computer" on the one hand and things like chair or George Washington on the other. Another categorization of things is IndividualObject versus Collection. Cyc also provides for classification as a *substance* (e.g., wood, water, mayonnaise) or as an *individual* (e.g., John Hancock, the Empire State Building). There are also novel subsystems in Cyc for reasoning about time and causality. Much of Cyc's knowledge is represented as a collection of "microtheories." Each microtheory describes the classes of objects and events relevant to one subject, such as taking a taxi, illnesses, or buying clothes.

4.14.4 Other Components of Cyc

Besides being a large knowledge base, Cyc is a system that includes the following components: a knowledge server that processes operations on the knowledge base; a user interface including browsers, query processors, and editors; a machine learning module that combs through the knowledge base, typically at night, looking for unusual relationships (some of which turn out to be bugs!); and nonpropositional knowledge components including digital images and neural networks.

Method	*Relations handled*	*Inference mechanisms*	*Strong organization?*	*Principal shortcomings*
Proposition-al logic	Boolean truth functions	Modus ponens, etc.	No	Models only boolean truth relationships but not the statements themselves
Concept hierarchy	ISA	Graph search and transitive closure	Yes	Limited to one relation
Predicate logic	Any predicate	Resolution & others	No	Lacks facilities for organizing knowledge; awkward for control information
Frames	Binary or ternary	Not provided	Yes	Only a methodology; not an actual rep. system
Semantic nets	Binary or ternary	Not provided	No (except with partitioning)	No standard
Constraints	Any predicates	Propagation; satisfaction	No	No standard
Production rules	If-then	Rule activation	No	Awkward for nonprocedural knowledge
Relational database	n-ary	Selection, projection, join	Somewhat	Awkward for control information

Figure 4.23: Summary and rough evaluation of eight methods for representing knowledge.

As of this writing, the Cyc research and development project was nearing the end of its planned 10-year lifetime. It will be very interesting to see its various applications.

4.15 Summary of Representation Schemes

Eight of the methods discussed in this chapter are compared in Figure 4.23.

A serious system for knowledge-based reasoning must combine two or more of the basic approaches. For example, a frame system may be organized as an ISA hierarchy whose nodes are schemata with instance frames linked to them. The slots of the frames may be considered to represent predicates, and the filled-in values and frame names may be viewed as arguments (terms) of the predicates, so that logical inference may use the knowledge in the frames. At

the same time, a base of production rules may encode the procedural knowledge and heuristics that use the knowledge base to manipulate the state information to solve particular problems. Thus, in this example, four of the basic knowledge representations are used: frames, ISA hierarchies, predicate logic, and production rules. This chapter has provided an introduction to knowledge representation, but it should be clear that there is more to this subject than can be expressed in one chapter of this length. In particular, inference methods as described in following chapters add much to a knowledge representation system. For example, the rules for making inferences using assertions to which degrees of uncertainty have been associated can be an important representation of knowledge, as is described in Chapter 7.

4.16 Bibliographical Information

A recent analysis of the knowledge representation issue can be found in [Davis et al. 1993] where five roles are identified that knowledge representations play: surrogates for real-world situations and events, a set of ontological commitments (assumptions about what is relevant), partial theories of reasoning, media for efficient computation, and media of human expression.

A very readable introductory article on knowledge representation is [McCalla and Cercone 1983]. That article is also an introduction to a special issue of *IEEE Computer* devoted to knowledge representation. The issue contains 15 additional articles that collectively present a good survey of knowledge representation. Another good collection of papers is [Brachman and Levesque 1985].

Elementary properties of binary relations (reflexiveness, symmetry, antisymmetry, and transitivity, for example) are treated by many texts on discrete mathematics, such as [Tremblay and Manohar 1975]. An algorithm for computing the transitive closure of a relation was developed by Warshall and is given in [Aho, Hopcroft, and Ullman 1974]. For an intriguing treatment of the semantics of ISA see [Brachman 1983]. For a more general treatment of inheritance see [Touretsky 1986].

A collection of research papers that address the issue of representation of knowledge is [Bobrow and Collins 1975]. One of those papers is particularly good as an introduction to the problems of representing the kinds of knowledge that can support dialogs in natural language [Woods 1975].

The use of the predicate calculus as a knowledge representation method is described in [Nilsson 1981]. The frames approach to knowledge organization was presented in [Minsky 1975]. A formalism called KRL, that stands for "knowledge representation language," was developed for expressing knowledge in a frame-like way; see [Bobrow and Winograd 1977]

Constraints were used extensively as a means of knowledge representation in [Borning 1979]. Good overviews of the use of constraints are [Deutsch 1981] and [Tsang 1993].

Relational databases, developed in large part by E. Codd [Codd 1970], are introduced in [Date 1976] and [Ullman 1982]. A thorough theoretical treatment is provided in [Maier 1983]. Many of the issues common to database systems and knowledge representation are treated in papers that were presented at a workshop sponsored by three ACM special interest groups and published in [Brodie and Zilles 1981]. A collection of readings on the interrelationships between AI and databases is [Mylopoulos and Brodie 1988]. Readings on object-oriented databases are given in [Zdonik and Maier 1989].

The Cyc project, whose goal is to represent and make available commonsense knowledge to computer systems, is described in [Guha and Lenat 1990] and [Lenat et al. 1990]. A discussion of Cyc's use of contexts for microtheory representation can be found in [Guha and Lenat 1992].

Recent research in approximate representations of knowledge about the physical world has developed new methods of qualitative reasoning such as one based on exaggeration (see [Weld 1988]). An overview of qualitative reasoning in the domain of physics is [Forbus 1988]. A collection of readings on this subject is [Weld and de Kleer 1989]. A related area is qualitative simulation (see [Kuipers 1986]).

References

1. Aho, A., Hopcroft, J. E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley.

2. Bobrow, D. G., and Collins, A. (eds.) 1975. *Representation and Understanding: Studies in Cognitive Science*. New York: McGraw-Hill.

3. Bobrow, D. G., and Winograd, T. 1977. An overview of KRL: A knowledge representation language. *Cognitive Science*, Vol. 1, No. 1., pp. 3-46.

4. Borning, A. 1979. ThingLab: A constraint-based simulation laboratory. Ph.D. dissertation, Dept. of Computer Science, Stanford University. Stanford, CA.

5. Brachman, R. J. 1983. What IS-A is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, Vol. 16, No. 10 (October), pp. 30-36.

6. Brachman, R. J., and Levesque, H. J. (eds.) 1985. *Readings in Knowledge Representation*. San Mateo, CA: Morgan Kaufman.

7. Brodie, M. L., and Zilles, S. N. (eds.) 1981. *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling*, Pingree Park, CO. Published jointly 1981 as *ACM SIGART Newsletter*, No. 74 (January), *SIGMOD Record*, Vol. 11, No. 2 (February), and *SIGPLAN Notices*, Vol. 16, No. 1 (January).

8. Codd, E. F. 1970. A relational model for large shared data banks. *Communications of the ACM*, Vol. 13, No. 6 (June), pp. 377-387.

9. Date, C. J. 1976. *An Introduction to Database Systems*. Reading, MA: Addison-Wesley.

10. Davis, R., Shrobe, H., and Szolovits, P. 1993. What is a knowledge representation? *AI Magazine*, Vol. 14, No. 1, pp. 17-33.

11. Deutsch, P. 1981. Constraints: A uniform model for data and control. In Brodie, M. L., and Zilles, S. N. (eds.) *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling*, Pingree Park, CO, pp. 118-120.

12. Forbus, K. 1988. Qualitative physics: Past, present and future. In Shrobe, H. E. (ed.), *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, San Mateo, CA: Morgan Kaufman, pp. 239-296.

13. Guha, R. V., and Lenat, D. B. 1990. Cyc: A midterm report. *AI Magazine*, Vol. 11, No. 3 (Fall), pp. 32-59.

14. Guha, R. V., and Lenat, D. B. 1992. Language representation and contexts. *Journal of Information Processing*, Vol. 15, No. 3, pp. 340-349.

15. Lenat, D. B., Guha, R. V., Pittman, K., Pratt, D., and Shepherd, M. 1990. Cyc: Toward programs with common sense. *Communications of the ACM*, Vol. 33, No. 8 (August), pp. 30-49.

16. Maier, D. 1983. *The Theory of Relational Database Systems*. New York: Computer Science Press.

17. McCalla, G., and Cercone, N. 1983. Approaches to knowledge representation. *IEEE Computer*, Vol. 16, No. 10 (October), pp. 12-18.

18. Minsky, M. 1975. A framework for representing knowledge. In Winston, P. H. (ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, pp. 211-277.

19. Mylopoulos, J., and Brodie, M. (eds.) 1988. *Readings in Artificial Intelligence and Databases*. San Mateo, CA: Morgan Kaufman.

20. Nilsson, N. J. 1981. *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Press; also Los Altos, CA: William Kaufman, 1983.

21. Quillian, M. R. 1968. Semantic memory. In Minsky, M. (ed.), *Semantic Information Processing*. Cambridge, MA: MIT Press, pp. 27-70.

22. Raphael, B. 1976. *The Thinking Computer: Mind Inside Matter*. San Francisco: W. H. Freeman and Company.

23. Touretsky, D. S. 1986. *The Mathematics of Inheritance Systems*. San Mateo, CA: Morgan Kaufman.

24. Tremblay, J.-P., and Manohar, R. P. 1975. *Discrete Mathematical Structures with Applications to Computer Science*. New York: McGraw-Hill.

25. Tsang, E. 1993. *Foundations of Constraint Satisfaction*. London: Academic Press.

26. Ullman, J. D. 1982. *Principles of Database Systems*, 2d ed. New York: Computer Science Press.

27. Weld, D. 1988. Comparative analysis. *Artificial Intelligence*, Vol. 36, pp. 333-373.

28. Weld, D., and de Kleer, J. (eds.) 1989. *Readings in Qualitative Reasoning about Physical Systems*. San Mateo, CA: Morgan Kaufman.

29. Woods, W. A. 1975. What's in a link: Foundations for semantic networks. In

30. Bobrow, D. G., and Collins, A. (eds.) *Representation and Understanding: Studies in Cognitive Science*. New York: McGraw-Hill, pp. 35-82.

31. Zdonik, S., and Maier, D. (eds.) 1989. *Readings in Object-Oriented Databases*. San Mateo, CA: Morgan Kaufman.

Exercises

1. Imagine a computer network of the future in a large hospital, that includes patient-monitoring devices, medical records databanks, and physicians' workstations. Explain some possible uses of data, information, and knowledge in this environment.

2. Describe how knowledge may be represented in each of the three parts of a production system.

3. For each of the following, determine which of the two relations "subset-of" or "element-of" is being represented, and reformulate the statement to make this clearer. The first one is done for you. If you find genuine ambiguity in a statement, justify each of the possible interpretations.

   (a) Fido is a dog. Fido $\in$ dogs

(b) A parrot is a bird.

(c) Polly is a parrot.

(d) David Jones is a Jones.

(e) "George Washington" is a great name.

(f) Artificial intelligence is a state of mind.

4. For each of the following relations, state whether or not it is reflexive, whether or not it is symmetric, whether or not it is transitive, whether or not it is antisymmetric, and whether or not it is a partial order. For each example, let the set S on which the relation is defined be the set of elements mentioned in that example.

   (a) {(a, a)}

   (b) {(a, b), (a, c), (b, c)}

   (c) {(a, a), (a, b), (b, b), (b, c), (a, c), (c, c)}

   (d) {(a, b), (b, c)}

   (e) {}

5. Let R be the relation {(a, b), (a, c), (b, c)}. Draw the graph of this relation. Draw the Hasse diagram for this relation.

6. Give an example of a transitive relation on a set of people. Is the "ancestor-of" relation transitive? How about "parent-of," "cousin-of," "sister-of," and "sibling-of?" Assume that these are "blood-relative" relations rather than the more general ones that include adoptions, etc.

7. Write a Python program that takes a binary relation and computes its transitive reduction.

8. Write a Python function that determines whether the input relation is reflexive or not.

9. Let us assume that "A house has a roof." means that a house has a roof as a part. Suppose we want to extend Linneus to know about and reason with HAS links as well as ISA links.

   (a) Let Isa(x, y) mean "an x is a y," and let Has(x, y) mean "an x has a y as a part." New HAS links may be inferred from combinations of existing ISA and HAS links. For example,

   $$\text{Has}(x, y) \wedge \text{Isa}(y, z) \Rightarrow \text{Has}(x, z).$$

   Complete the predicate calculus formulation of the rules for inferring HAS relationships described on page 113.

(b) Just as `isa_test(x, y, n)` succeeds if there is a path from `x` to `y` of length `n` or less, following only ISA links, one can imagine a function `hasatest(x, y, n)` that tests for an implied HAS relationship between `x` and `y`. Exactly what kind of path between `x` and `y` implies that the test should succeed?

(c) Extend the Linneus program to properly handle HAS links. Allowable inputs should include expressions such as

A dog has a snout.

which expresses the fact that a dog has a snout,

A dog has a leg.

which says a dog has a (at least one) leg, and

Has a dog a paw?

which asks, "Does a dog have a paw?" or equivalently, "Do dogs have paws?"

10. Extend the Linneus program to automatically maintain its inclusion hierarchy in transitive reduction (Hasse diagram) form. (That means that no link is stored if it can be inferred by transitivity or reflexiveness.) In connection with this, the conversational front end should handle the following new kinds of responses:

```
I already know that by inference.
I have been told that before.
Your earlier statement that a mouse is an animal is now redundant.
```

You may name your new program whatever you like. Suppose it is called "SMARTY." If you tell SMARTY that "A dog is a mammal." and then later tell it exactly the same thing, it should respond "I have been told that before.". If you tell it something that it can already deduce, it should respond "I already know that by inference.", and if you tell it something new (not already implied) that makes a previously input fact redundant, SMARTY should reply with a statement such as "Your earlier statement that a mouse is an animal is now redundant.". Furthermore, the redundant link should then be removed, so that the internal data structure is kept nonredundant. Test your program on the following sequence of facts plus another sequence of your own creation.

```
A lion is a carnivore.
A lion is a carnivore.
A carnivore is an animal.
A lion is an animal.
```

```
A lion is a thing.
A dog is a thing.
A mammal is a thing.
A dog is an animal.
An animal is a thing.
A dog is a mammal.
```

11. By including more kinds of links in an ISA hierarchy, we can obtain a more general kind of semantic network.

    (a) Extend the program Linneus to properly handle the ELEMENT-OF relation expressed by user inputs such as "Janet is a woman." and "Larry is a lobster.". Your program should correctly handle questions such as

        i. Who is Janet?
        ii. What is Larry?
        iii. Is Larry an animal?
        iv. Is a larry an animal?
        v. What is a janet?

        — especially the last two. These two types should be answered with a message that indicates that they contain a false presupposition.

    (b) Further extend the program to handle the ownership relation as in "Larry owns a clam." or "Janet owns a Porsche.". Allow the user to type appropriate statements and questions and get answers that are reasonable. Note that if Janet owns a Porsche and a Porsche is a car, then Janet owns a car.

12. Draw a semantic network representing the sentence "Artificial Thought, Inc., bought a controlling interest in Natural Ideas, Inc., for the sum of $23 million." Include nodes for each object or event and the class to which it belongs.

13. Suppose that we wish to represent some notion of the ISA hierarchy of Figure 4.4 using propositional calculus. It is very difficult to represent the hierarchy in a way that would let us do reasoning based on inheritance of properties, for example. However, consider the following statements:

    P1: "Larry is a lobster." (I.e., Larry is a member of the class lobster.)
    P2: "Larry is a crustacean."
    P3: "Larry is an arthropod."
    etc.

Class inclusion may be represented (in this very specific case of Larry) by the expressions:

P1 $\Rightarrow$ P2
P2 $\Rightarrow$ P3
etc.

This knowledge can be used to infer "Larry is an arthropod" from the statement "Larry is a lobster."

(a) If we add the statement P4, "Louise is a lobster," what can be inferred about Louise?

(b) Give additional propositions to support inferences about Louise.

(c) Give a set of *predicate* calculus expressions for the knowledge that allows some inferences about both Larry and Louise.

14. Encode the following statements in the propositional calculus. State what clauses are represented by each of your proposition symbols.

> If the home team wins then the picnic will be in Victory Garden; otherwise it will be at Consolation Springs. If it rains, the sheltered barbeque will be used. Whenever it rains, the Joneses stay home. The Smiths only attend if it rains.

15. The predicate calculus supports certain kinds of quantification quite nicely, but not all kinds.

(a) For each of the statements below, give a predicate calculus formula that represents its meaning.

i. There exists a white elephant.
ii. There uniquely exists a white elephant.
iii. There are at least two white elephants.
iv. There exist exactly two white elephants.
v. There exist precisely three white elephants.

(b) Describe a scheme that, for any given n, can be used to create a formula to represent the statement "There exist exactly n white elephants."

(c) What does your answer to part (b) suggest about the predicate calculus in representing numerically quantified statements?

16. By using the symbol P to represent the statement "It is raining today," so much of the detail of the sentence has been lost in abstraction that nothing is left that represents the objects or actions that make up the

statement. In the predicate calculus, on the other hand, we might represent the statement with "Weather(Today, Raining)", which provides representational components for important parts of the statement. However, if the symbol P were replaced by the identifier "Raining" and the predicate "Weather(Today, Raining)" were redescribed as $P(a, b)$, then the predicate calculus representation would seem less informative than the propositional one. Explain the cause of this apparent paradox and which representational scheme provides a more informative representation.

17. Encode the following statements into the predicate calculus, capturing the main relationships being expressed. In each case, specify an appropriate domain.

    (a) There are no birds named Tweety in the Woodland Park Zoo.

    (b) The tallest mountain in Washington State is Mount Rainier, but the tallest mountain in the world is Mount Everest.

    (c) The number 2 is the largest positive integer n for which there exist integers x, y, and z that satisfy the equation $x^n + y^n = z^n$.

18. Attempt to represent the following sentence in the predicate calculus. What problems do you run into?

    > It is uncertain whether Fermat would have been able to understand Wile's proof of his famous theorem.

19. Consider the domain $D = \{a, b, c, d, e\}$, and let I assign the relation $\{(a, b), (b, a), (a, c), (c, a)\}$ to the predicate $P(x, y)$. Then D and I are a model for which one of the following formulas?
    [A] $\exists x P(x, x)$, [B] $\forall x \forall y (P(x, y) \rightarrow P(y, x))$, [C] $\forall x \exists y P(x, y)$,
    [D] $\forall x \forall y (P(x, y) \wedge P(y, x))$, or [E] $\exists x \forall y P(x, y)$
    Explain your answer.

20. Describe how the knowledge necessary to drive a car might be organized in a collection of frames.

21. Write representations for the following constraint schemata mentioned on page 142:

    (a) INITIAL

    (b) SERIES_CURRENT

    (c) SERIES_RESISTANCE

22. Design, implement, and test a Python program that accepts a list of constraint schemata and a list of constraints and determines the values of uninitialized variables by applying the constraints in a systematic fashion. Demonstrate your procedure on the circuit problem described in Figure 4.18.

23. Consider the following problem. Let $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ be the set of vertices of a triangle whose perimeter is P and whose area is A. Suppose $x_1 = 0, y_1 = 0, x_2 = 6, y_2 = 0$, and $A = 30$.

    (a) Develop constraint schemata and instances to represent the problem.

    (b) Which variables are forced to particular values? Which are not forced? What, if anything, can be said about the ranges of possible values for the variables that are not forced?

24. Relational databases are frequently incorporated into intelligent systems.

    (a) Project the relation ANGIOSPERMS (on page 143) to obtain a new relation employing only the attributes "general form" and "seed body."

    (b) Compute the join of the relation you just found in part (a) with the relation R3 on page 145.

    (c) What is the sequence of selections, projections, joins, and counting operations necessary to process the query "How many products are made from small plants?" using the relations "angiosperms" and R3?

25. Relational database operations can be coded in Python to demonstrate question-answering capabilities.

    (a) Design a Python scheme for representing n-ary relations.

    (b) Write one or a collection of Python functions to compute the projection of a relation (onto any given list of attributes).

    (c) Write a Python function to compute the join of two relations.

    (d) Add whatever functions and other representations may be necessary to automatically handle queries such as that of the previous problem, part (c). Assume these queries are presented in the form (A1 A2 V2) which means "How many different values of A1 can be found in tuples whose A2 value is V2?"

26. Suppose a system uses a small knowledge base consisting of the following statements in the predicate calculus, and suppose it is capable of making logical deductions, but does notq know anything special about any of the predicates such as "Color" or "Equal" that might appear in its knowledge base.

    - Color(Apple1, Red)
    - Color(Apple2, Green)
    - Fruit(Apple1)
    - Fruit(Apple2)

- $\forall x\{[\text{Fruit}(x) \wedge \text{Color}(x,\text{Red})] \Rightarrow \text{Ripe}(x)\}$

Assume the system makes the closed-world assumption. For each of the predicate calculus statements (a) through (h) tell whether the system would assign to it a value of *true*, *false*, or *unknown*:

(a)	Color(Apple3, Red)	(e)	¬Color(Apple1, Red)
(b)	Fruit(Apple2)	(f)	Color(Apple1, Blue)
(c)	Ripe(Apple1)	(g)	NotEqual(Red, Blue)
(d)	Ripe(Apple2)	(h)	Equal(Red, Blue)

(i) To what extent does the system make a consistent interpretation of the statements (a) through (h)?

(j) Is there any inconsistency?